

**Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596**

**ARCHITECTURAL AND COMPILER SUPPORT FOR
EFFICIENT FUNCTION CALLS**

Miquel Huguet

**September 1989
CSD-890051**

UNIVERSITY OF CALIFORNIA

Los Angeles

Architectural and Compiler Support for Efficient Function Calls

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Computer Science

by

Miquel Huguet

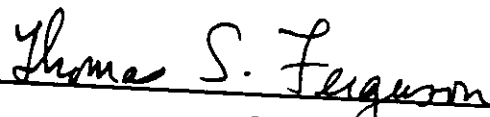
1989

© Copyright by

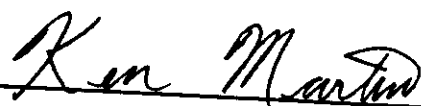
Miquel Huguet

1989

The dissertation of Miquel Huguet is approved.




Thomas S. Ferguson



Kenneth W. Martin



Yuval Tamir



Richard R. Muntz



Tomás Lang, Committee Chair

University of California, Los Angeles

1989

*In the memory of my father and my Aunt Mercè
who left us during this journey,
and to my mother and my sister,
for all their love, support,
and encouragement.*

*To Laia and Robert Michael
who have just started
their journey in life.*

Contents

1	Introduction	1
1.1	The Problem and Our Research Contributions	1
1.2	The Block-and-Actions Generator	6
1.3	Programs Measured	10
1.4	Organization of this Dissertation	13
2	Previous and Related Work	15
2.1	Previous Work	15
2.1.1	Multiple-Window Architectures	15
2.1.2	Single-Window Architectures	21
2.1.3	Register Allocation and Assignment	22
2.1.4	Performance Studies on Register Allocation	35
2.1.5	Performance Evaluation of New Architectures	39
2.2	Related Work	43
2.2.1	Cache versus Registers	43
2.2.2	Instruction Format (RISC versus CISC)	45
3	RSR Architectural Support for Single-Window Register Files	47
3.1	Static Policies A and B	54
3.2	Policies C and D	55
3.3	Policies E, F, G, and H	60

3.4	An Example for the Eight RSR Policies	66
3.5	On the Implementation of Policy G	66
3.6	Summary	73
4	RSR Intra-Procedural Optimizations	75
4.1	Compiler Support to Reduce the RSR Traffic	77
4.1.1	Policy A-live	77
4.1.2	Policy A-live Optimized (A-lvOpt)	80
4.1.3	Policies B, C, and G with Leaf Functions	82
4.1.4	Policies C-live and G-live with Leaf Functions	86
4.1.5	Summary	91
4.2	A Comparison with Other Register Allocators	93
4.3	A Comparison with SPICE	98
4.4	A Comparison with Multiple-Window Register Files	101
4.5	Overall Data Memory Traffic Generated	106
4.6	Speed-Up	111
4.7	Summary	114
5	Inter-Procedural Optimizations	117
5.1	The Inter-Procedural Optimizer	120
5.2	Register Allocation for Global Scalar Variables	123
5.3	Return-Address Inter-Procedural Optimization	127
5.4	RSR Inter-Procedural Optimizations	136
5.4.1	Global Policy A-live	137
5.4.2	Global Policy A-lvOpt	148
5.4.3	Global Policy B-lf	156
5.4.4	Global Policy G-lf	166
5.4.5	Summary	175

5.5	General-Purpose Register Set Partition	177
5.5.1	Overall Data Memory Traffic Reduction for the Four Programs . .	178
5.5.2	Overall Data Memory Traffic Reduction for SPICE	181
5.5.3	Speed-Up	183
5.6	Summary	185
6	Conclusions and Future Work	189
6.1	Conclusions	189
6.2	Suggestions for Future Work	191
A	On Intra-Procedural Register Allocation	195
A.1	Selection Mechanism for Local Scalars	196
A.2	Dedicated versus Shared Registers for Local Scalars	199
	Bibliography	203

List of Figures

1.1	Operations to Be Performed on a Function Call	2
1.2	Gathering Measurements for the PM on the EM	6
1.3	One-to-One Block Mapping	7
1.4	PM to EM Block Translation	8
1.5	Gathering Type II Measurements with BKGEM	10
2.1	Multiple-Window Register Files	16
2.2	64-Shift-Register File with Three-Size Windows	19
2.3	A Global View of the Compilation Process	24
2.4	Basic Blocks	25
2.5	Intra-Procedural Register Allocation and Assignment	27
2.6	General-Purpose Register Set Partition	30
2.7	Gathering Measurements with the Step-by-Step Instruction Tracer	41
3.1	Relation among the Register Saving/Restoring Policies	49
3.2	Activation Records where Registers Are Saved	51
3.3	RSR Traffic per Function Call with Fixed Cost	53
3.4	Policy A	54
3.5	Policy B	55
3.6	Policy C	56
3.7	Policy D	59
3.8	Policy D versus Policy C	59

3.9	Policy E versus Policy D	60
3.10	Register Pointers	61
3.11	Policy E	62
3.12	Policy F	63
3.13	Policy G	64
3.14	Policy H	64
3.15	Example of RSR Traffic Caused by the Different Policies	67
3.16	Implementation of Policy G	69
3.17	RISC-like Pipeline without Restoring	70
3.18	RISC-like Pipeline with Restoring	71
3.19	Critical Path for a Restoring Operation	72
3.20	Critical Path for a Saving Operation	72
4.1	Policy A-live versus Policy A	78
4.2	Policy A-lvOpt versus Policy A-live	80
4.3	RSR Traffic for Policies A-live and A-lvOpt	82
4.4	Leaf-Function Optimization for the Return Address	83
4.5	RSR Traffic for Leaf-Function Optimization (Policies B and G)	85
4.6	RSR Traffic for Leaf-Function Optimization (Policies B and C)	85
4.7	Example for Policy G-live	86
4.8	RSR Traffic for Policy G-live	89
4.9	RSR Traffic for Policy G-live with <i>Clear-TBS</i> Instruction Traffic	90
4.10	RSR Traffic for Policy C-live	91
4.11	Average Number of Registers To Be Saved During Context Switching	92
4.12	RSR Traffic for Optimized Policies	92
4.13	ASM RSR Traffic for Optimized Policies	93
4.14	RSR Traffic for Policy B-lf	96
4.15	RSR Traffic for Policy G-lf	97

4.16	RSR Traffic for Policies A-live and A-lvOpt	97
4.17	RSR Traffic for GNU Compiler	98
4.18	VAX-11 Instructions per Executed Basic Block	98
4.19	Data Memory Traffic per Executed Function	99
4.20	To-Be-Preserved Registers Required per Executed Function	99
4.21	RSR Traffic Generated by the GNU C Compiler for SPICE	100
4.22	RSR Traffic for Multiple-Window Register Files	103
4.23	Average Number of Registers To Be Saved During Context Switching . . .	105
4.24	Distribution of Data Memory Traffic	106
4.25	Overall Data Memory Traffic with PCC for the Four Programs	107
4.26	Overall Data Memory Traffic with GNU for the Four Programs	108
4.27	Overall Data Memory Traffic with GNU for SPICE	108
4.28	Overall Data Memory Traffic Distribution	110
4.29	Speed-Up for Intra-Procedural Optimizations w.r.t. 0 TBP registers . . .	112
4.30	Speed-Up for Intra-Procedural Optimizations w.r.t. Policy B-lf	113
4.31	GNU Speed-Up for 12, 16, and 24 TBP registers w.r.t. Policy B-lf	113
5.1	Scalar Data Memory Traffic To Be Eliminated	118
5.2	The Inter-Procedural Optimizer in the Compilation Process	122
5.3	Global Scalar Traffic for the Four Programs	126
5.4	Global Scalar Traffic for SPICE	126
5.5	Function Execution Frequency in a Call Graph	128
5.6	Register Assignment with Three-Link Registers	129
5.7	A Better Register Assignment with Three-Link Registers	130
5.8	Distribution of Functions by Height	132
5.9	RA Traffic Reduction for the Four Programs	132
5.10	RA Traffic Reduction for ASM	134
5.11	An Example of Register Assignment for Policy A ^g -live	138

5.12	Registers Assigned to Functions f_2 , f_3 , and f_4	138
5.13	Variable-to-Register Assignment for Function f_1	139
5.14	Distribution of the RSR Traffic Caused by Policy A-live	139
5.15	Variable-to-Register Assignment for Function f_1 when $K = 11$	140
5.16	Register Assignment for Policy A ^g -live	142
5.17	Wrong Computation for $K[i]$ when Values Are Added	143
5.18	K Values with Two Passes	147
5.19	Cases for which Policy A ^g -lvOpt Cannot Be Optimized	150
5.20	Policy A ^g -lvOpt versus Policy A-lvOpt	151
5.21	Register Assignment for Policy A ^g -lvOpt	152
5.22	Distribution of the RSR Traffic Caused by Policy A-lvOpt	154
5.23	RSR Traffic Generated by Policies A ^g -live and A ^g -lvOpt	155
5.24	Distribution of Functions by Depth	157
5.25	Distribution of the RSR Traffic Caused by Policy B-lf	157
5.26	An Example of Register Assignment for Policy B ^g -lf	158
5.27	Alternative Register Assignments for Function f_4	159
5.28	Optimal Variable-to-Register Assignment for Function f_4	160
5.29	Register Assignment for Policy B ^g -lf	161
5.30	Alternative Register Assignments for Algorithm 3	162
5.31	RSR Traffic for Policy B ^g -lf	165
5.32	Disjoint Register Assignment for Policy G ^g -lf	167
5.33	An Example of Register Assignment for Policy G ^g -lf	168
5.34	Variable-to-Register Assignment	168
5.35	Register Assignment for Policy G ^g -lf	170
5.36	RSR Traffic for Policy G ^g -lf	172
5.37	Inter-Procedural Optimized RSR Traffic for the Four Programs	172
5.38	Inter-Procedural Optimized RSR Traffic for SPICE	175

5.39	Data Memory Traffic for the Four Programs with the Best Static Global Policy	179
5.40	Data Memory Traffic for the Four Programs with Policy G ^s -lf	179
5.41	Remaining Data Memory Traffic for the Four Programs	181
5.42	Data Memory Traffic for SPICE with the Best Static Global Policy	182
5.43	Data Memory Traffic for SPICE with Policy G ^s -lf	182
5.44	Remaining Data Memory Traffic for SPICE	183
5.45	Speed-Up for Inter-Procedural Optimizations w.r.t. Policy B-lf	184
5.46	Speed-Up for Inter-Procedural Optimizations w.r.t. Policy G-lf	185
A.1	Local Scalar Traffic for the Four Programs	198
A.2	Local Scalar Traffic for SPICE	199
A.3	Dedicated versus Shared Registers for the Four Programs	201
A.4	Dedicated versus Shared Registers for SPICE	202

List of Tables

1.1	Large Programs Measured by Some Other Studies	12
2.1	Multiple-Window Processors	17
2.2	Single-Window Processors	22
2.3	Features for Several Register Allocators	36
3.1	Register Saving/Restoring Policies	48
3.2	RSR Traffic Relative to Policy B	52
3.3	Hardware Implementation for Set Operations	56
4.1	RSR Traffic for Policy A Optimizations Relative to Policy B	79
4.2	Percentage of Consecutive Calls	81
4.3	Percentage of Leaf Functions	83
4.4	RSR Traffic for Leaf-Function Optimization Relative to Policy B	84
4.5	RSR Traffic for Policies C-live and G-live Relative to Policy G-lf	87
4.6	Percentage of Calls which Issue a <i>Clear-TBS</i> Instruction	88
4.7	RSR Traffic Generated by ACK and GNU Relative to PCC Policy B-lf	95
4.8	RSR Traffic for Multiple-Window Register Files	102
4.9	Percentage of Functions which Must Expand the Window Size	103
4.10	Percentage of Overflows in Multiple-Window Register Files	104
4.11	Average Number of Registers To Be Saved During Context Switching	105
5.1	Input Data for Profiling the Measured Programs	124

5.2	Global Scalar Traffic Reduction	125
5.3	Percentage of Functions in a Cycle	130
5.4	RA Traffic Reduction with Inter-Procedural Optimization	133
5.5	RA Traffic Reduction with Multiple PCs	135
5.6	RSR Traffic Reduction for Policy A ^g -live	145
5.7	RSR Traffic Reduction for Policy A ^g -lvOpt	153
5.8	RSR Traffic Reduction for Policy B ^g -lf	164
5.9	RSR Traffic Reduction for Policy G ^g -lf	173
5.10	RSR Traffic Reduction for the Global Policies	174
A.1	Local Scalar Traffic Reduction	197
A.2	Local Scalar Traffic with Dedicated and Shared Registers	200

ACKNOWLEDGMENTS

Many people have helped me during my 7-year journey at UCLA. I would like to thank them, because without the positive learning environment which they helped create, this dissertation would not have been finished.

Tomás has been an excellent mentor. He has been a role model for both my teaching and research, has always pushed me to find better and more efficient algorithms, and has offered innumerable suggestions to improve the readability and the content of my drafts. Tomás and Diana were marvelous hosts upon my arrival to Los Angeles, making my transition smooth and happy.

Dick, Miloš, and Dave Martin have also been terrific teachers and have encouraged me to improve my teaching skills. Dick has also offered advice on linear optimization. Yuval has provided suggestions on how to perform architecture measurements and positive criticism on the implementation of one of the architectural policies proposed in this dissertation. Mateo and Jordi Cortadella have also commented on previous drafts.

Maria Eugenia and Hector, Marisa and Jaime, Ravi, Brett, Leon, Martine and Pak, Marc, Frank, Doris, Art, Verra, Jeong-A, and Paul have helped to create a warm and stimulating environment in the Computer Science Department where research and personal growth have been possible.

Marisol and Robert, Josefina, Olga, Pilar, Maribel and Jordi, Luisa and Edward, Magda, Andolin, Jon, Neil, Chris, Darren, and Tim have been my adopted family in Los Angeles, have made life in the city more bearable and lively, and provided me with an emotional support during the period of time that we have enjoyed together. Also, Neil has proofread this manuscript.

FULBRIGHT/MEC, CIRIT, and GTE provided financial support during my studies at UCLA.

To all, thank you very much!

VITA

- March 27, 1956 Born in Alforja, *El Baix Camp de Catalunya*, Spain
- 1979 *Licenciado en Informàtica*
Universitat Politècnica de Catalunya, Barcelona
- 1979 National Award for Best Student in Computer Science
- 1980–1982 Ministry of Education, Young Researchers Fellowship
- 1980–1982 Lecturer
Universitat Politècnica de Catalunya, Barcelona
- 1981 *Licenciado con Grado en Informàtica*
Universitat Politècnica de Catalunya, Barcelona
- 1982–1984 FULBRIGHT/MEC Fellowship
- 1984–1989 Teaching Assistant, Associate, and Fellow
Computer Science Department
University of California, Los Angeles
- 1984–1986 CIRIT (*Generalitat de Catalunya*) Fellowship
- 1985 Master of Science in Computer Science
University of California, Los Angeles
- 1987–1988 GTE Fellowship
- 1989 Distinguished Teaching Award
Computer Science Department
University of California, Los Angeles

PUBLICATIONS AND PRESENTATIONS

1. M. Huguet and T. Lang, "Reduced Register Saving/Restoring for Small Register Files," UCLA Technical Report CSD-880066, August 1988.
2. M. Huguet, "Architectural and Compiler Support for Efficient Function Calls: A Proposal," UCLA Technical Report CSD-880030, June 1988.
3. M. Huguet, T. Lang, and Y. Tamir, "A Block-and-Actions Generator as an Al-

- ternative to a Simulator for Collecting Architecture Measurements," *Proc. of the SIGPLAN'87 Symp. on Interpreters and Interpretive Techniques*, June 1987, pp. 14–25.
4. T. Lang and M. Huguet, "Reduced Register Saving/Restoring in Single-Window Register Files," *Computer Architecture News*, Vol. 14, No. 3, June 1986, pp. 17–26.
 5. M. Huguet and T. Lang, "A Reduced Register File for RISC Architectures," *Computer Architecture News*, Vol. 13, No. 4, September 1985, pp. 22–31.
 6. M. Huguet and T. Lang, "A C-Oriented Register Set Design," *Proc. of the 29th Symposium on Mini and Microcomputers and Their Applications*, June 1985, pp. 182–189.
 7. M. Huguet, "A C-Oriented Register Set Design," Master's Thesis, University of California, Los Angeles, June 1985.
 8. M. Huguet, "The Protection of the Processor Status Word of the PDP-11/60," *Computer Architecture News*, Vol. 10, No. 4, June 1982, pp. 27–30.
 9. M. Huguet, "Monitorització del Sistema Operatiu RSX-11M," Minor Thesis, Facultat d'Informàtica, Universitat Politècnica de Catalunya, December 1981.

ABSTRACT OF THE DISSERTATION

Architectural and Compiler Support for Efficient Function Calls

by

Miquel Huguet

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1989

Professor Tomás Lang, Chair

With the current trend towards VLSI load/store architectures, registers have become one of the critical resources for increasing processor performance. One of the important factors which influences the design and use of a register file is the overhead produced by the *register saving and restoring* (RSR). This dissertation investigates the support that the architecture can provide to implement efficient function calls for *single-window architectures* combined with the support that can be provided by the compiler. Several architectural policies which reduce the RSR overhead for single-window architectures are presented. These policies make use of *dynamic* information to determine which registers have been used during program execution. They are evaluated and compared to the conventional *static* policies for single-window architectures (to save/restore registers at the caller or at the callee) and to the already-existing schemes for *multiple-window architectures* (fixed-size windows, variable-size windows, and multi-size windows). We show that one of our dynamic policies reduces significantly the RSR traffic with respect to the conventional static policies and that most of its activities are performed in parallel with the main CPU activities.

Six new compiler optimizations to reduce the RSR traffic are also presented. Two of them are *intra-procedural* and are based on live-variable analysis. The other four are *inter-procedural* and their goal is to find a *register assignment* so that unnecessary RSR can be eliminated. These optimizations are also evaluated and compared to the already-existing intra-procedural optimizations (leaf functions and live-variable analysis) and to

other inter-procedural register allocation schemes. The inter-procedural optimizer not only reduces the RSR traffic, but also the global scalar traffic and the return-address traffic.

We show that a combination of both architectural and compiler support, i.e., our dynamic policy with either intra-procedural or inter-procedural optimizations, is the best approach; it generates less data memory traffic than when only either architectural or compiler support is provided. Also, it simplifies the compilation process.

Both the architectural and the compiler policies have been evaluated with a new tool, the *Block-and-Actions Generator*, which substantially reduces the overhead introduced by a conventional simulator, the tool traditionally used to obtain these types of measurements. The Block-and-Actions Generator's main advantage is that, since the execution time is substantially reduced, *large typical* programs can be measured—the NROFF word processor, the SORT program, the VAX-11 assembler, the Portable C Compiler, and SPICE, a VLSI circuit simulator.

Chapter 1

Introduction

In this introductory chapter we offer first a description of the problem that this dissertation addresses and a summary of our contributions towards its solution (Section 1.1). Second, we present our tool to perform the measurements discussed in this work (Section 1.2) and the programs measured (Section 1.3). Finally, we give the general outline of this dissertation (Section 1.4).

1.1 The Problem and Our Research Contributions

With the current trend towards VLSI load/store architectures, registers have become one of the critical resources to increase processor performance [Patt82b, Radi82, Henn84, Patt85a]. Recent advances in compiler technology, especially in *register allocation* optimization [Chai82, Ankl82, Chow83, Wall86, Stee87, Chow88], have resulted in better use of the register set because the compiler can allocate more *scalar* variables to registers. The use of registers reduces the data memory traffic, because whenever operands are already available in processor registers, no memory address has to be calculated and no memory access has to be generated. The use of registers also reduces the instruction memory traffic because shorter addresses are used, and for load/store architectures no load (store) instruction has to be generated to fetch (transfer) the operand from (to) memory. However, registers have to be saved every time a *procedure* or *function* is called, and restored when the function returns (see Figure 1.1). It has been observed that one of the important factors which influences the design and use of the register set is the overhead produced by the register saving and restoring (RSR).

For instance, Lunde [Lund77] affirms that the BLISS compiler spends 25% of its execution time compiling the program “Treesort” in call administration. Patterson and Séquin [Patt82b] have weighted the relative frequency of C-language [Kern78] statements to conclude that 12% of the statements are calls and that they correspond to 33% of the machine instructions and to 45% of the memory references. (These numbers have been computed from the average number of instructions and references per statement

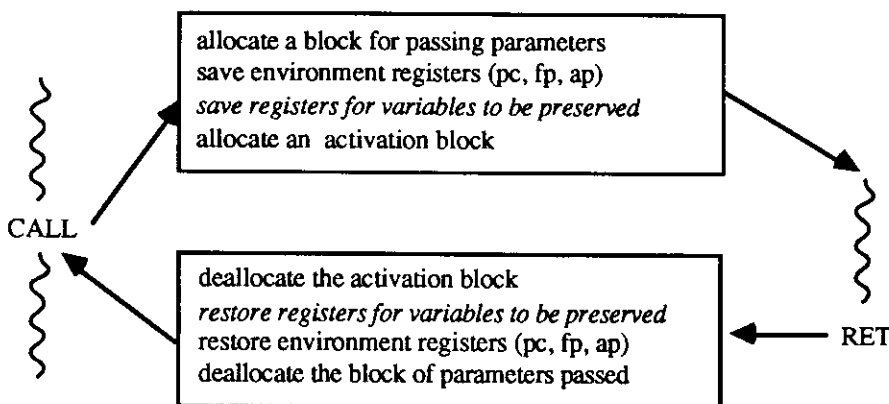


Figure 1.1: Operations to Be Performed on a Function Call

generated by the C compilers for VAX-11, PDP-11,¹ and MC68000.) The RSR overhead becomes more significant for languages such as C, which encourage the use of functions.

To reduce the RSR overhead some current processors rely on hardware support, such as multiple-window register files [Patt82b, Kate83, Unga84, Atki87, Ditz87c], intra-procedural optimizations [Radi82, Chow84], and/or inter-procedural optimizations [Wall86, Stee87, Chow88, Rich89].

Multiple-window architectures have divided the register file into a set of banks or *windows*. When a function is called, a new window of registers is made active [Site79a, Dann79, Lamp82], so that registers have to be saved only when no more free windows are available in the register file. In contrast, processors with the conventional general-purpose register file directly addressable by each function are classified as *single-window architectures*.

Although multiple-window architectures produce a large reduction in the RSR traffic [Patt82a, Patt82b, Hugu85a, Flyn87], they have three main drawbacks: they use a large chip area in a VLSI implementation or a large number of chips in a MSI/LSI implementation; they increase the amount of processor context to be saved on context switching; and they increase the processor cycle-time due to the long data busses [Henn84, Sher84]. These drawbacks are discussed in Subsection 2.1.1.

Due to these drawbacks, several processors prefer to have the conventional single-window register file and rely on compiler optimizations to reduce the RSR overhead, such as live-variable analysis [Hech77, Aho86] and leaf-function optimization (as used in the compilers for MIPS [Chow86] and for HP-Spectrum [Cout86b]). These optimizations are discussed in Subsection 2.1.3.4. However, the compiler complexity is increased. To reduce the cost of developing an optimizing compiler for every language and every machine [John81], current research on optimizing compilers pursues the independence of the optimizations from both the source language and the target machine

¹VAX-11 and PDP-11 are trademarks of Digital Equipment Corporation.

[Perk79, Alle80, Ausl82, Tane83, Powe84]. This is also true for the algorithms which perform register allocation [John75, Site79b, Leve83, Chow83].

This dissertation investigates both the support that the architecture can provide to implement efficient function calls in single-window architectures and the support that can be provided by the compiler. By *architectural support* we mean the hardware included in the processor to implement a specific RSR operation which would be executed much more slowly if it was implemented by the standard set of machine instructions provided. The algorithms which describe the operations to be performed in hardware are called *architectural policies*. Similarly, *compiler support* refers to the (software) algorithms provided by the compiler to reduce the number of RSR operations to be performed during execution.

The Measurements

Both the architectural and the compiler policies have been evaluated with a new tool: the *Block-and-Actions Generator (BKGEN)*. BKGEN drastically reduces the overhead introduced by a conventional simulator—the tool traditionally used to obtain these types of measurements. Section 1.2 presents BKGEN and Subsection 2.1.5 comments on the overhead introduced by conventional simulators. The main advantage of BKGEN is that, since the execution time is substantially reduced, *large typical* programs can be measured. The typical UNIX² programs measured are: the NROFF word processor, the SORT program, the VAX-11 assembler (ASM), the Portable C Compiler for VAX-11 (VPCC), and SPICE, a VLSI circuit simulator. More information on these programs is given in Section 1.3. Since SPICE has different characteristics than the other four programs (as we mention in Section 1.3 and show in Section 4.3), the measurements presented in this dissertation are usually given separately for SPICE and for the average of the other four programs.

Three compilers have been used in this dissertation to evaluate the data memory traffic generated and the traffic reduction obtained by our new architectural and compiler policies.³ These compilers are: the Portable C Compiler, PCC [John79], the Amsterdam Compiler Kit, ACK [Tane83], and the GNU C Compiler [Stal88]. These compilers have been modified to generate code for six different register configurations. These configurations correspond to when the register allocator has 6, 8, 12, 16, 24, and 32 registers available for variables to be preserved across function calls (e.g., local scalar variables). The *overall* data memory traffic generated by these compilers for the above-mentioned programs is divided into the following six categories: local scalar traffic, RSR traffic, global scalar traffic, return-address traffic, parameter-passing traffic, and non-scalar traffic. The traffic caused by expression evaluation is ignored because expressions are evaluated with a small number of registers (see Subsection 2.1.4).

²UNIX is a registered trademark of AT&T.

³Each policy has *not* been evaluated with every compiler, although we expect that similar results would be obtained for any compiler (see Sections 4.2 and 4.5).

RSR Architectural Support

This dissertation presents six new architectural policies to reduce the RSR overhead for single-window architectures. These policies make use of *dynamic* information to know which registers have been used during program execution. They are evaluated and compared to the conventional *static* policies for single-window architectures (to save/restore registers at the caller—named **Policy A**—or at the callee—named **Policy B**) and to the already-existing schemes for multiple-window architectures (fixed-size windows [Kate83], variable-size windows [Ditz82], and multi-size windows [Hugu85a]).

We show that one of the dynamic policies, **Policy G**, is our best candidate for implementation since it is the one that generates the least RSR traffic. Policy G has between 12% (when 24 registers are available to the allocator) and 31% (for 6) of the RSR traffic generated by Policy B and between 5% (for 24) and 20% (for 6) of Policy A. In addition to reducing the RSR traffic, Policy G also reduces the number of registers to be saved/restored during context switching. We also show that when the register set size is increased, the static Policies A and B generate even more RSR traffic, while this is not the case for the dynamic Policy G. The implementation of Policy G is sketched to show that most of its activities are performed in parallel with the main CPU activities.

RSR Intra-Procedural Optimizations

Two new *intra-procedural* (i.e., performed per function) compiler optimizations are also presented to reduce the RSR traffic. These are based on live-variable analysis: one is for Policy A and the second for Policy G. These optimizations are also evaluated and compared to the already-existing compiler optimizations: live-variable analysis and leaf functions. When intra-procedural optimizations to reduce the RSR traffic are performed, we conclude the following:

1. Policy G with leaf-function optimization (**Policy G-If**) is the policy which generates the least traffic for any program and for any of the three compilers used in this dissertation. For instance, with the GNU C Compiler, Policy G-If has between 13% and 42% of the RSR traffic generated by the best intra-procedural static policy (**Policy B-If**) for the average of the four programs and between 31% and 48% for SPICE.
2. When only compiler support is provided, our measurements have indicated that there is no best approach for performing the register saving and restoring, either at the caller or at the callee. Depending on the program characteristics, one approach can perform better than the other. Thus, the compiler should let the programmer decide which of the two approaches is the most suitable for a specific application.
3. Our measurements have not shown the necessity for having more than 12 registers available to the allocator for non-numeric applications because the reduction in local scalar traffic obtained with a larger number of registers is balanced with the

increase in RSR traffic. This is not the case for numeric applications, like SPICE, because of its heavier register use.

4. When 12 registers are available to the allocator, the overall data memory traffic reduction, when both architectural and compiler support are provided, is 15% for the average of the four programs and 8% for SPICE with respect to when only compiler support is provided.

Inter-Procedural Optimizations

To reduce the RSR traffic further, four new *inter-procedural* (i.e., applied to the whole program) compiler optimizations are also presented. One inter-procedural optimization is based on the dynamic Policy G, the other three on static intra-procedural policies. Their goal is to find a *register assignment* such that unnecessary RSR can be eliminated. Register assignment decides the registers to be used by the variables that have been selected for allocation in a previous phase by the intra-procedural register allocator.

Our inter-procedural optimizer not only reduces the RSR traffic, but also the global scalar traffic and the traffic caused by the return address. The inter-procedural optimizer uses a *dynamic execution profile* of the programs to obtain a better overall data memory traffic reduction. When inter-procedural optimizations are performed, we conclude the following:

1. The Global Policy G with leaf-function optimization (**Policy G^s-lf**) is the policy which generates less RSR traffic than any of the three proposed static inter-procedural policies. For instance, with the GNU C Compiler, the best inter-procedural static policy has between 445% (when 12 registers are available to the allocator) and 816% (for 32) of the RSR traffic generated by Policy G^s-lf for the average of the four programs and between 103% and 155% for SPICE.
2. When only compiler support is provided, our measurements have indicated that there is no best approach for performing the register saving and restoring among the three static inter-procedural policies. Thus, the inter-procedural optimizer should have all of the policies available to select the most suitable for a specific application. Still, this increases the complexity of the inter-procedural optimizer. On the other hand, Policy G^s-lf not only generates the least RSR traffic, but also is the easiest inter-procedural policy to implement.
3. The RSR traffic becomes smaller for larger register sets for all four inter-procedural RSR policies. Thus, the inter-procedural optimizer can efficiently use a larger register set. This is not the case when only intra-procedural optimizations are performed.
4. For a 32-general-purpose-register file and for the average of the four programs, the overall data memory traffic reduction when both architectural and compiler

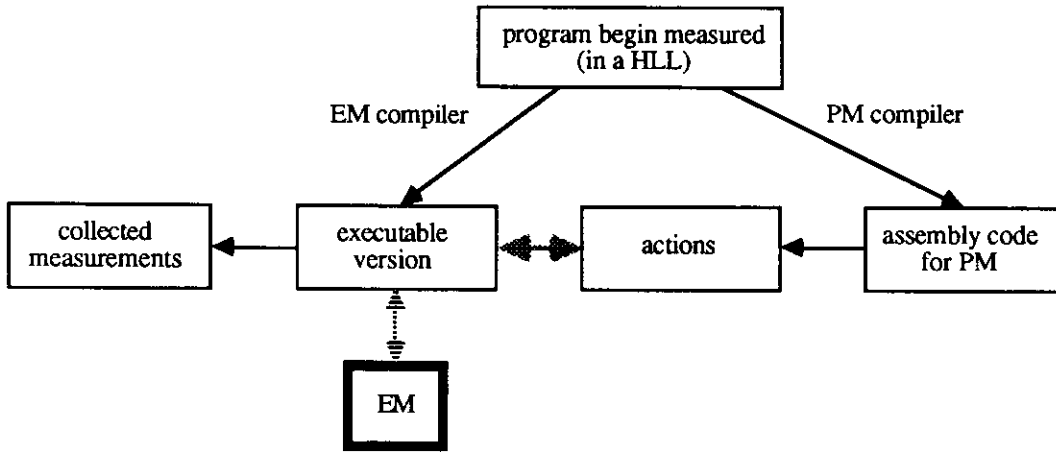


Figure 1.2: Gathering Measurements for the PM on the EM

support are provided, versus when only inter-procedural compiler support is available, is 14%. The overall data memory traffic reduction with respect to the intra-procedural optimizer with Policy G-1f is 21%. For SPICE, the overall data memory traffic generated when both architectural and compiler support is provided is the same as the one generated when only compiler support is provided. The reason for this will be discussed in Section 5.5.

Therefore, since Policy G^s-1f usually generates the least data memory than any static inter-procedural policy (with the exception of SPICE), both architectural and compiler support (i.e., Policy G^s-1f) is the best approach to reduce the overall data memory traffic and to simplify the compilation process.

1.2 The Block-and-Actions Generator

Conventional simulators used for collecting dynamic measurements are limited by their execution speed because several hundred instructions are required to decode, interpret, and measure each simulated instruction (see Subsection 2.1.5). To be able to measure *large typical* programs we have designed a new tool for collecting architectural measurements: the Block-and-Actions Generator (**BKGEN**).

Given a program to be measured (in a high-level language), the goal of BKGEN is to run directly an executable version of this program on an *existing machine* (EM) while collecting measurements for the *proposed machine* (PM). This executable version is obtained directly either with the EM compiler or with a combination of the PM compiler and an assembly-to-assembly translator. The choice between these alternatives depends on the EM and PM compiler technology and the type of measurements to be obtained (as we will discuss below). BKGEN also collects the PM events to be measured (called *actions*). Each EM *basic block of instructions* is associated with a PM *block of actions*

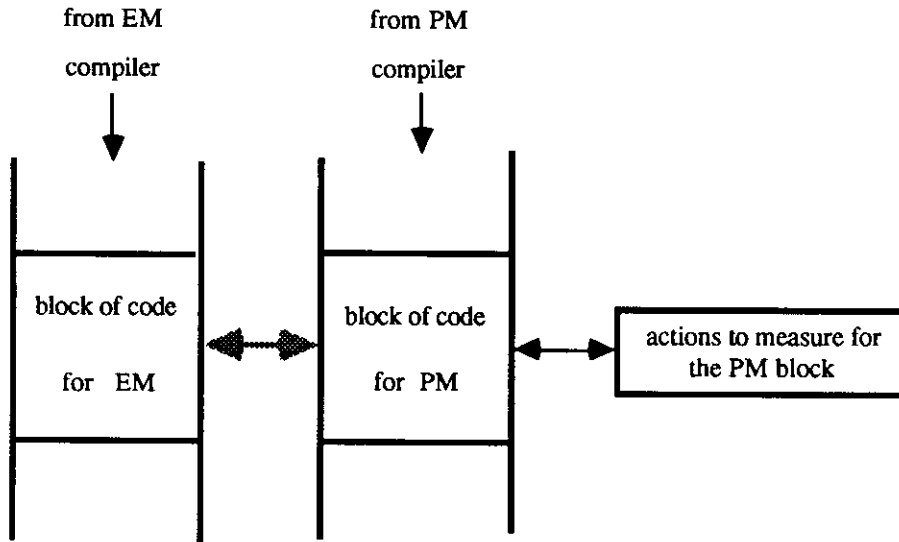


Figure 1.3: One-to-One Block Mapping

so that when the program is executed, it collects the measurements associated with the PM (see Figure 1.2).

To present how BKGGEN is implemented we explain first how to associate EM blocks to PM blocks and actions; afterwards, we discuss the size of a block and the interception code that has to be added to each EM block to perform the measurements.

If a number is associated with each block, then the execution flow of a program can be described by the sequence of block numbers. This flow depends mainly on the original program structure, the transformations performed by the compiler, and the input data, but usually not on the processor architecture (some exceptions are discussed in [Hugu87]). Thus, the flow for both the EM and the PM is generally the same. In this case, each block of code for the EM is *mapped* to a block of code for the PM. This mapping can be performed as follows:

1. If the compiler technology used for the PM is the same as (or similar to) the one used for the EM, then it is probable that both compilers generate the same number of blocks and equivalent control flow instructions. If both programs were executed and the trace of executed blocks was identical, we would say that we have a *valid mapping* (see Figure 1.3). In this case, it is possible to execute machine language instructions of the EM directly and measure the architecture characteristics of the PM.
2. Otherwise, we have an *invalid mapping*. In this case, the assembly code of the PM has to be translated to the assembly code of the EM (see Figure 1.4). The complexity of the translation depends on similarities between instruction sets. Thus, additional overhead is introduced which depends on the number of EM instructions that have to be executed per PM instruction. However, the overhead is

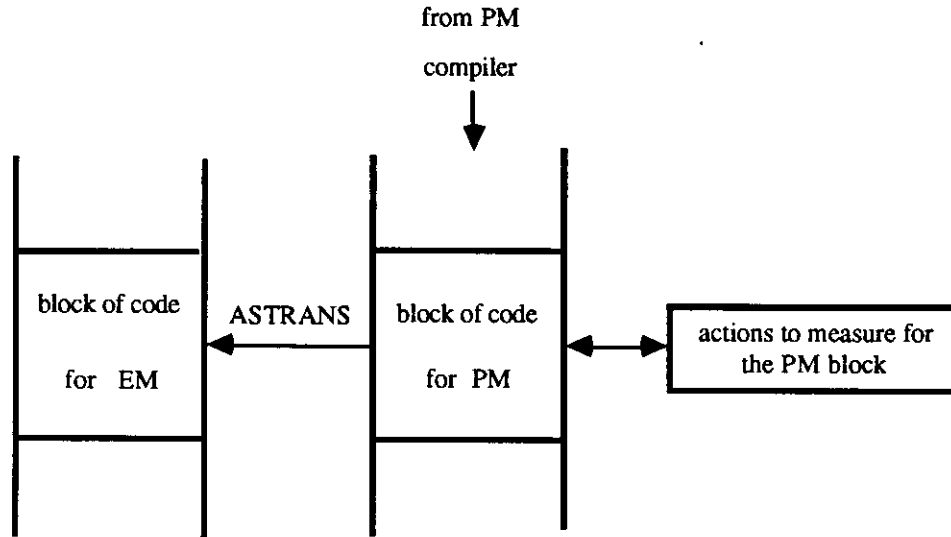


Figure 1.4: PM to EM Block Translation

always smaller than for a simulator because only a few instructions have to be executed here per *simulated* (PM) instruction. This is the approach taken by [Chow86, Cort87b] (see Subsection 2.1.5).

Since we have always been able to find a valid mapping for the measurements that we have taken so far, the reader is referred to [Hugu87] for more information on the second alternative.

The size of the block and the interception code depends on the type of measurements that are being performed. Measurements can be classified into the following three types:

- I Measurements that are *independent of the order of execution* of the instructions. Examples are: opcode frequency, addressing modes distribution, register usage, memory traffic caused by local scalar variables, traffic caused by register saving and restoring with static policies.
- II Measurements that are *dependent on the sequence* of instructions executed, but *independent of the state* of the machine. In this case, the measurements do not depend on the values that are being computed during program execution. Examples are: frequency of pairs of opcodes, nested stack depth, branches taken or not taken, number of instructions executed between branches, traffic caused by register saving and restoring with dynamic policies, traces of instruction addresses.
- III Measurements that are *dependent on the state* of the machine. In this case, the function that is collecting the measurements needs to access the values computed during execution. Examples are: memory addresses generated for data, distribution of operand values for the multiplier, branch distances.

For measurements of Type III, each EM block corresponds to only one PM instruction because the PM state has to be updated after the EM instructions (associated with a PM instruction) have been executed. Since our measurements are of Types I and II and the complexity of BKGGEN for measurements of Type III is greater than the one for the former types, we concentrate our attention on BKGGEN for measurements of Types I and II. The reader is referred once more to [Hugu87] to find more information on BKGGEN for measurements of Type III.

For measurements of Types I and II, the block corresponds to a basic block (as defined in Subsection 2.1.3). The advantage of grouping the PM instructions in a block is that the overhead caused by the EM instructions which are collecting the measurements is reduced since these EM instructions have to be executed once per PM block rather than once per PM instruction.

For measurements of Type I, the interception code consists of incrementing a counter associated with the block number, because it is only necessary to know how many times each block has been executed. (The order of instructions or blocks is not significant.) When the program finishes its execution, the number of occurrences of each event is computed as the product of the number of executions of each block by the number of event occurrences per block (given by the actions file). The use of this scheme for measuring the frequency of executed instructions for new machines was proposed by Weinberger [Wein84].

For measurements of Type II, the interception code consists of a call to a *measurement function*. This function can either (a) produce an execution trace of the blocks or (b) compute the measurements during program execution. The selection between these alternatives depends on the specific measurements and the overhead associated with their processing per block. Our experience with BKGGEN has shown that it is more efficient to perform the measurements during program execution (*in-line*) due to the large overhead for writing a file of several Mbytes and the amount of disk space required for it; this case is shown in Figure 1.5.

To estimate the overhead introduced by the different alternatives we have counted the number of executed instructions in the SORT program.⁴ As we will show in Subsection 2.1.5, this can be considered the simplest type of measurement and, therefore, the observed overhead is the minimum overhead introduced by the different measurement techniques. The overhead generated is 1.5 times its normal execution time when the interception code only counts the number of times that a block has been executed, 8.4 times when the measurements are performed in-line, and 11.5 times when the sequence of block numbers is written to a file [Hugu87]. In contrast, the same measurement using the UNIX `ptrace` system call and implementing the tracer as an exception handler produces an overhead of 1,540 and 540 times, respectively.

To have a correct and successful execution of the measured programs, the interception code must not change the state of the EM (i.e., the registers, the condition codes, etc.).

⁴Note that this is a Type I measurement. The reason for performing this measurement as Type II is only to estimate the overhead introduced.

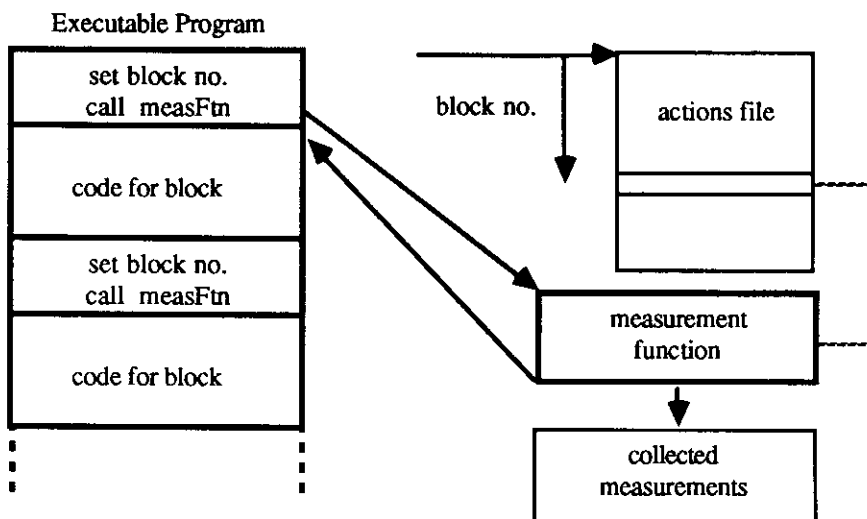


Figure 1.5: Gathering Type II Measurements with BKGGEN

The reader is referred to [Hugu87] to see how this is done on VAX-11.

The overhead introduced for the Type I measurements is very small—only a few instructions to increment a counter while preserving the machine state per block. For Type II, the overhead depends on the measurements that are being performed. However, since the measurement function is called only once per block and the events to be measured are given directly by the actions file, the overhead is always substantially smaller than for a simulator. For instance, the overhead to measure the RSR traffic generated by the static and the dynamic RSR policies is, on the average, 30 times the normal program execution.

In conclusion, the block-and-actions generator lets the designer measure many of the same events that can be measured by a simulator, an emulator, or a step-by-step instruction tracer. The effort to develop this new tool is not greater than the effort to develop a simulator, an emulator, or an instruction tracer. Since BKGGEN directly executes machine code for the machine on which the measurements are performed, the execution time for obtaining measurements that do not depend on the machine state is significantly smaller than the execution time required to simulate the same program. Thus, for these types of measurements the designer can obtain meaningful dynamic characteristics of typical programs.

1.3 Programs Measured

In this section we first review some of the *standard benchmarks* that have been used to perform architectural evaluations and comment on why we do not use them. Afterwards, we present the *large typical programs* that we have selected to perform our evaluations.

In many cases the programs used as benchmarks are either *small specific programs*, such as the Ackermann's function [Wich76], LINPACK [Garb77], the Hanoi Towers, the Quicksort, the Fibonacci Numbers, the Erasthenes Sieve [Gilb81], and the Puzzle Program [Beel84], or *synthetic programs* that either reflect some high-level language characteristics, such as the CFA benchmarks [Full77] and Dhrystone [Weic84], or measure some architecture characteristics, such as Whetstone [Curn76]. Programs of the first type are too small to be considered typical of any real system behavior, while those of the second type can only be representative of the characteristics considered in their design.

Our first measurements on several of the above-mentioned programs show that the results obtained from benchmarks might not be representative of real system behavior [Hugu85a]. For instance, the RSR traffic caused by Policy G for some of these benchmarks either is zero or has little variation from the traffic caused by Policy B. This is not the case when the large typical programs are measured. Some other authors [Levy82, Chow83, Colw85, Patt85b, Hitc86, Laru86, Wall88, Wong88] have also pointed out the limitations of these benchmarks for evaluating certain architecture features.

Table 1.1 shows some large programs used by some other studies. Small programs are enclosed in parentheses. Since only [Cort87a, Cort88] have used a fast simulator to perform their measurements, we have to assume that the simulation time required for the other studies was large since this is not reported in the papers and/or the number of different alternatives measured was small (to reduce the overall simulation time).

The large typical programs that we have selected to perform our measurements are written in C [Kern78]. C has been selected because it is widely used for system programming [Feue82], it has a type flexibility that allows to program system functions traditionally coded in assembler [Kern81, Post83], it has been used to implement UNIX, it can be used as an intermediate language for other block-structured languages such as ADA [Hill83], and it has been used to implement other languages such as LISP and PROLOG. No attempt is made in this work to generalize the measurements obtained for the C programs. The reader is referred to [Weic84] or [Huck83] for a comparison of the characteristics of different programming languages to C. The programs are:

ASM. The UNIX assembler for VAX-11 (*as*), assembling one of the machine-independent Portable C Compiler modules (*allo.s*). ASM has 177 functions defined and executes 29,453 function calls.

NROFF. The UNIX *nroff* word processor, formatting one third of the manual page entry for the FORTRAN 77 compiler. NROFF has 300 functions defined and executes 379,831 function calls.

SORT. The UNIX *sort* program, sorting a file with 2,250 numbers. SORT has 68 functions defined and executes 142,448 function calls.

VPCC. The UNIX Portable C Compiler (*ccom*) for VAX-11, compiling one of its machine-independent modules (*allo.c*). VPCC has 337 functions defined and executes 154,071 function calls.

Reference	Description
[Ditz82]	<ul style="list-style-type: none"> • First Pass of the Portable C Compiler • UNIX word processor • Second Pass of the PCC (generating PDP-11 code) • VLSI design ruler checker
[Blom83]	<ul style="list-style-type: none"> • UNIX Portable C Compiler • (Puzzle and TAK—heavy recursive function)
[Tami83]	<ul style="list-style-type: none"> • UNIX Portable C Compiler • (Puzzle and Towers of Hanoi)
[Laru86]	<ul style="list-style-type: none"> • LISP—Spice Lisp system from CMU • SLC—SPUR Lisp Compiler (based on Spice Lisp) • RSIM—circuit simulator
[Band87]	<ul style="list-style-type: none"> • UNIX system programs: cat, comm, diff, echo, mv, nroff, pr, rm, and wc
[Cort87a] [Cort88]	<ul style="list-style-type: none"> • Portable C Compiler for RISC • NROFF—UNIX word processor • YACC—parser generator • LEX—lexical analyzer
[Eick87]	<ul style="list-style-type: none"> • AS—UNIX assembler for VAX-11 • CCOM—main part of the UNIX Portable C Compiler • COMPACT—adaptive Huffman code file compressor • EQN,TBL,NROFF—UNIX equation, table, and word processors • INDENT—C source program indenting program • PI—Pascal interpreter code translator • SORT—UNIX sort program • YACC—parser generator
[Flyn87]	<ul style="list-style-type: none"> • CCAL—emulates a desk calculator • Compare—compares 2 text files and indicates differences • PCOMP—compiles PASCAL programs and generates P-code output • PASM—assembles the P-code output • Macro—macro processor for SCALD

Table 1.1: Large Programs Measured by Some Other Studies

SPICE. A VLSI circuit simulator (“translated” to C from FORTRAN), simulating a unidirectional ratioless shift cell. SPICE has 1,171 functions defined and executes 175,567 function calls.

SPICE is a program that has different characteristics than the previous ones because it was originally written in FORTRAN. SPICE has fewer function calls, larger basic blocks, and heavy floating-point variable usage (see Section 4.3). Thus, we prefer to isolate this program from the rest. For this reason, in Section 1.1 we commented on the results for the average of the four programs (ASM, NROFF, SORT, VPCC) and for SPICE.

In some parts of this dissertation we use averages from *our previous measurements*. These are reported in [Hugu85a] and correspond to only three of the programs (NROFF, SORT, and VPCC). These measurements do not include the library functions which these programs use. However, we expect that the conclusions obtained would be the same if the library functions were included.

1.4 Organization of this Dissertation

The outline of the dissertation is the following: Chapter 2 presents a detailed summary of the work performed in this area (multiple-window architectures, register allocation and assignment, performance evaluation of new architectures, etc.) and of some related work that is not directly discussed in this dissertation (CISCs versus RISCs and cache memories versus registers).

Chapter 3 presents the six architectural policies to reduce the RSR traffic as well as the implementation sketch for Policy G. The RSR traffic generated by these policies is compared with the RSR traffic generated by the standard Policies A and B when no optimizations are performed. Only one compiler is used to perform the evaluation, the Portable C Compiler, because similar results would be obtained with the other two.

Chapter 4 presents the two new intra-procedural RSR policies and compares them with the existing RSR policies (live-variable analysis and leaf-function optimization). All three compilers are used to evaluate these policies. The RSR traffic generated is also compared to the RSR traffic produced by multiple-window architectures.

Chapter 5 introduces our inter-procedural optimizer. First, the traffic reduction obtained by each optimization (for global scalar traffic, for the return-address traffic, and for RSR traffic) is discussed separately. Afterwards, the independent optimizations are put together and the optimal partition of the general-purpose register set is evaluated. Only one compiler is used to perform the inter-procedural evaluations, the GNU C Compiler, since it is the one which generates the least overall data memory traffic when intra-procedural optimizations are performed.

Finally, Chapter 6 summarizes the contributions of this dissertation and suggests areas for future work.

Chapter 2

Previous and Related Work

2.1 Previous Work

This section presents the previous work that has been performed in the following areas:

- Multiple-window architectures.
- Single-window architectures.
- Register allocation and assignment and RSR policies.
- Performance studies on register allocation.
- Performance evaluation of new architectures.

These are discussed in turn.

2.1.1 Multiple-Window Architectures

With recent technological advancements it is possible to have a large number of registers on a single chip [Site79a]. To reduce the register saving and restoring traffic *multiple-window architectures* have the large register file divided into *register windows*. Every time that a function is called, a new window is made available [Site79a, Dann79, Lamp82] so that no RSR has to be performed. When there are no more windows available (i.e., an *overflow* condition is detected), one or more windows must be transferred to memory [Tami83]. Similarly, when a function returns and there are no more windows in the register file (i.e., an *underflow* condition is detected), one or more windows have to be transferred back from memory. Windows might overlap to pass parameters through registers so that data memory traffic is further reduced [Patt82b].

In this section, we first describe the general organization of a register file with multiple windows. Second, we present three schemes to divide the register file into windows: fixed-size windows [Kate83], variable-size windows [Ditz82], and multi-size windows [Hugu85a].

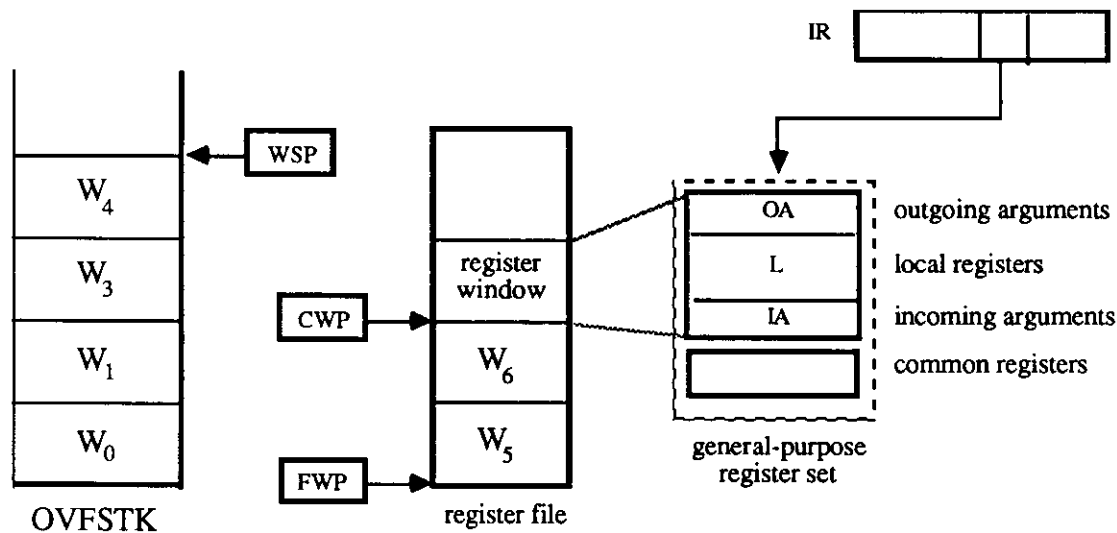


Figure 2.1: Multiple-Window Register Files

We comment on the drawbacks that fixed-size and variable-size windows have and how they are reduced by the use of multi-size windows and of a shift-register file. We also mention a different approach to reduce these drawbacks: the *dribble-back register file* [Site79a]. Finally, we discuss some evaluation studies that have been performed on multiple-window architectures.

Figure 2.1 shows the organization of a register file with multiple windows. The programmer (i.e., the compiler) sees the general-purpose register set divided into the following groups:

- A group of *common* registers to all the functions. The compiler has to preserve (if necessary) the contents of these registers to prevent them from being destroyed across function calls.
- A group of registers for the *incoming arguments* (IA) of the function. One of these registers contains the return address.
- A group of *local* registers. These registers are preserved across function calls (as well as the incoming registers).
- A group of registers for the *outgoing arguments* (OA) for the function that might be called.

The *window size* is defined as the number of new registers that are allocated per call. Thus, the size of the window is the number of local registers plus the number of overlapped registers (i.e., OA registers). In the figure we can also see the overflow stack, *OVFSTK*, where windows are transferred on overflow.

Most register file designs use *fixed-size windows* because of the simplicity in the implementation: C/70, RISC I and II, PYRAMID 90x, C1200, SOAR, SPUR, the LISP-

Processor	Description	References
C/70 (BBN Computer Co.)	<ul style="list-style-type: none"> • 8 general-purpose registers no overlapped/common registers • register file size = 1024 • number of windows = 128 • NO overflow exception detection 	[Kral80, BBN81]
RISC I & II (Univ. of California, Berkeley—UCB)	<ul style="list-style-type: none"> • 32 general-purpose registers window size = 16 • overlapped registers = 6 • common registers = 10 ($r0 \equiv 0$) • register file size = 128 + 10 • number of windows = 7+ • circular buffer organization 	[Piep81, Fitz82] [Laru82, Patt82a] [Patt82b, Séqu82] [Blom83, Kate83] [Patt83, Peek83] [Pond83, Tami83] [Patt84, Sher84]
CRISP (AT&T Bell Labs.)	<ul style="list-style-type: none"> • variable-size windows • registers mapped to memory • register file size = 32 • 7 special registers (<i>psw</i>, <i>pc</i>, ...) 	[Ditz82, Bere87a] [Bere87b, Ditz87a] [Ditz87b, Ditz87c]
Pyramid 90x (Pyramid Technology)	<ul style="list-style-type: none"> • 64 general-purpose registers window size = 32 • overlapped registers = 16 • common registers = 16 • register file size = 512 + 16 • number of windows = 15+ • circular buffer organization 	[PTC83, Raga83]
C1200 (Celerity Computing)	<ul style="list-style-type: none"> • As Pyramid, but with 8 register files for fast context switching • 4096 registers 	[CEL85, Olle85]
SOAR (UCB)	<ul style="list-style-type: none"> • 32 general-purpose registers window size = 8 (all overlap) • 8 special regs. ($r16 \equiv 0$; $r17-r23$ for OS) • 8 common registers • register file size = 64 + 16 	[Unga84, Samp85] [Samp86, Unga86] [Adam85, Chen85]
SPUR (UCB)	<ul style="list-style-type: none"> • general-purpose registers as RISC II • 15 floating-point registers • 7 special registers (<i>psw</i>'s, <i>pc</i>'s, reg. window ptrs.) 	[Gibs85, Hans85] [Katz85, Rite85] [Tayl85, Vill85] [Hill86, Tayl86] [Wood86, Borr87]
Dragon (Xerox)	<ul style="list-style-type: none"> • variable-size windows • registers in the Execution Unit: 128-stack regs, 16 auxiliary, 12 constant • registers in the Instruction Fetch Unit: 15-element stack of <i>pc</i>'s and context ptrs. 	[Atki87]
LISP-Machine (Univ. of Miami)	<ul style="list-style-type: none"> • register file size = 64 • window size = 8 • overlapped registers = 4 • circular buffer organization 	[Aboa87]
SPARC (Sun)	<ul style="list-style-type: none"> • variable register file size = 40–520 • 32-general purpose registers • window size = 16 (8 overlapped) 	[Garn88]

Table 2.1: Multiple-Window Processors

Machine, and SPARC (see Table 2.1;¹ this table also gives a detailed list of references for the reader interested in knowing more about these systems). The main drawback with fixed-size windows is the number of *unused* registers in the file. For instance, we have measured three C programs (the NROFF word processor, the Portable C Compiler for VAX-11, and the SORT program) and have shown that, on the average, RISC has 10.9 registers unused per function (for a window size of 16) and Pyramid has 26.8 (for a window size of 32) [Hugu85c]. These unused registers not only waste area in the chip, but also have to be saved and restored on overflow, on underflow, and on context switch.

Variable-size windows [Ditz82] assign per function the exact number of registers required for the variables selected for allocation. Thus, no unused registers are present in the file. Only two of the machines in Table 2.1 (CRISP and Dragon) use variable-size windows. Although variable-size windows utilize the register file more efficiently because the exact number of registers required is allocated, their implementation significantly increases the complexity of the processor: large register addresses are required in the instruction, two more machine instructions are required to execute a function call (one to allocate a window at function entry and a second to detect underflow after return), and more overhead is incurred to map a virtual register address to a physical register address [Hugu85a].

Although a multiple-window register file has been shown to be effective in reducing the memory traffic due to saving and restoring of registers in function calls/returns [Patt82a], the resulting register files have the following three drawbacks:

1. They use a large chip area in VLSI implementations [Henn84]. For instance, for RISC II [Kate83] the percentage of the chip area dedicated to register storage is 27.5% and to the decoders is 5.8%.
2. They increase the size of the context to be saved during context switching, since all the windows in use have to be saved [Henn84]. For instance, for the programs mentioned above, we have shown that, on the average, 70 registers have to be saved on context switching for RISC II and 211 for Pyramid [Hugu85c].
3. They increase the processor cycle-time due to the increase in the length of the data busses [Henn84, Patt85a]. This length is proportional to the size of the register file. For instance,
 - Sherburne [Sher84] claims that the datapath cycle time increases with the square root of the register-file size.
 - Ditzel *et al.* [Ditz87b] say that the access time for register files larger than 16 or 32 increases about 30% when the register-file size is doubled.

¹The calculation of the number of windows per register file requires some explanation. Let us consider, for instance, the 128-register file for RISC. Although the register file has exactly 8 windows (128 registers ÷ 16 registers per window), the table indicates that it has only 7⁺. This is because the first window in the file has 22 registers allocated (16 locals + 6 incoming registers) and, therefore, the eighth window is going to cause an overflow condition.

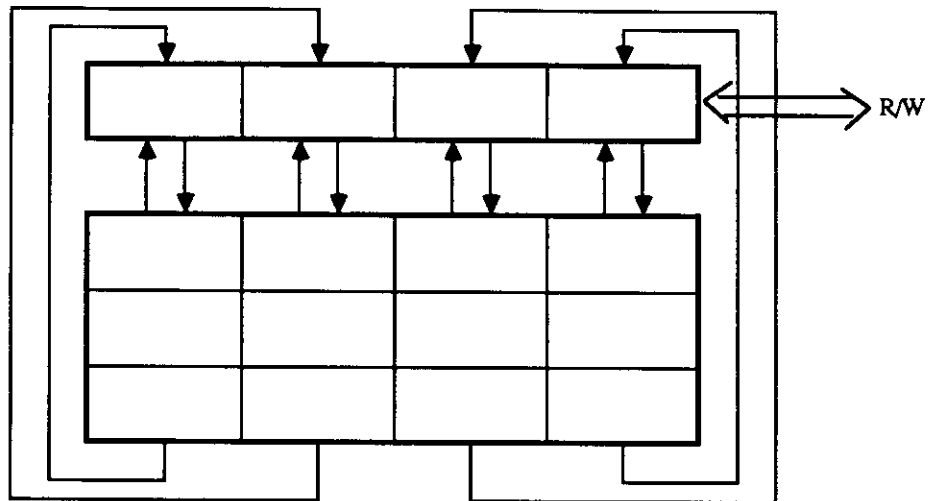


Figure 2.2: 64-Shift-Register File with Three-Size Windows

That is, for small register files (16 or 32) the other components in the datapath determine the basic processor cycle time. However, for larger ones the access time to the register file is critical in determining the processor cycle time.

To reduce these drawbacks we have previously proposed the combination of two approaches [Hugu85a, Hugu85b, Hugu85c]:

Multi-size windows: Instead of having only one window size (i.e., a fixed-size window), we have several sizes so that the best suited window is selected for the registers required by the function. *Two-size windows* (i.e., to have two window sizes) have also been proposed by Furht [Furh85, Furh88].

Shift-register file: This is a modification of the implementation of the circular-buffer register file [Kate83], which is used in most of the processors indicated in Table 2.1. Read and write operations are only performed on the registers of the top window. Registers are pushed (shifted down) when a function is called and popped (shifted up) when a function returns. In this case, the data busses are connected only to the top window so that the access time depends only on the register set, and not on the number of windows. For instance, Figure 2.2 shows a 64-register file with three-size windows of 4, 8, and 12 registers.

Multi-size windows differ from variable-size windows in that, by default, the smallest window is always allocated when the call is performed. Thus, if the callee does not require any more registers, no specific machine instruction is required at the callee to allocate a window. We have shown that when a 4-register window is allocated by default, 75% of the executed functions do not have to issue an instruction to increase the window size [Hugu85a]. Moreover, no explicit machine instruction is required to detect underflow after return. The standard return instruction detects an underflow condition when the largest window is not present in the register file.

Multi-size windows have essentially the same implementation complexity as fixed-size windows, but provide a better register utilization. For instance, a *three-size* window (4, 8, and 12) has, on the average for the three programs mentioned above (NROFF, SORT, and VPCC), 1.1 registers being unused; on the other hand, a fixed-sized window with 12 registers has 7.9. Moreover, for a 64-register file with three-size windows, the average number of registers saved per function call (0.06) doubles the average saved for a 512-register file with fixed-size windows of 32 (0.03) as used in PYRAMID and CELERITY. The implementation of multi-size windows produces, for the same overhead in register saving and restoring, a reduction in both the number of registers (windows) in the file and the size of the context to be saved. Therefore, two of the drawbacks (area required and context size) given by Hennessy [Henn84] are reduced.

Multi-size windows are also more suitable for general-purpose processors. Since different programming languages might use different window sizes, multi-size windows offer the flexibility of selecting the most appropriate one. For instance, measurements on C and PASCAL programs taken for RISC have shown that the most appropriate fixed-size window has 16 registers (with 10 local registers and 6 overlapped registers) [Patt82b], while measurements on Smalltalk programs taken for SOAR have shown that a window of 8 registers (with no local registers and 8 overlapped registers) is the most appropriate [Unga84]. Moreover, different application programs might also use different window sizes. For instance, our measurements have shown that a window size of 4 is large enough for 98% of the executed functions in NROFF, but for only 58% in the Portable C Compiler.

When the register file is implemented with shift registers, the area is further reduced and the access time becomes almost independent of the number of windows in the register file. It has been shown [Trem87] that the area for a 128-register file implemented as a circular buffer is 25% larger than the area required with shift registers and that the time of a READ operation has been reduced by 35%.

While multi-size windows reduce the three mentioned drawbacks, they do not completely eliminate these drawbacks. Thus, it is necessary to investigate alternative approaches to reduce the RSR overhead for the conventional single-window architectures. We will turn to these approaches in Chapter 3.

A different approach to reduce the above-mentioned drawbacks has been mentioned by some other authors [Site79a, Kate83, Good85, Stan87]: the *dribble-back register file*. The dribble-back register file has a smaller number of windows than the conventional register file with the circular-buffer organization. The windows which are not currently used are saved/restored in the background. That is, these windows are saved/restored not by explicit instructions on overflow/underflow, but by the processor when it is executing register-to-register operations and, thus, the memory port is idle. In this case, more data memory traffic might be generated, but this is not significant since it uses cycles on which the memory would not be used. Frazier [Fraz87] has shown that a dribble-back register file (he calls it a *trickle-back register file*) can also be implemented with multi-size windows. The advantage is that the register file can be even smaller. However, at this moment it is not clear that a dribble-back register file will improve processor performance, because of the extra hardware required for its implementation

and because no dynamic measurements are available (to the knowledge of the author) to show that the window saving/restoring can be performed in parallel with program execution.

Finally, let us comment on some other performance studies made on multiple-window architectures. These can be classified into three categories:

1. Studies that compare the execution speed of different processors [Piep81, Patt82a, Patt82b]. These comparisons are done across different machines. The machines have different instruction sets, different numbers of registers, different cycle times, different hardware technology, and the programs measured were compiled by different compilers. Therefore, from the results obtained (execution time) we cannot determine the performance benefit provided by multiple windows.
2. Studies that compare the data memory reduction obtained by a processor with and without multiple windows using a non-optimizing compiler [Hitc85, Eick87]. These comparisons give an estimation of the data memory traffic reduction obtained for a given processor with a single-window register file and with a multiple-window register file (with fixed-size windows) when a non-optimizing compiler is used. Both papers conclude that multiple-window architectures generate less memory traffic than single-window architectures. However, non-optimizing compilers usually perform an efficient register allocation for multiple-window architectures with hardware support to eliminate the alias problem [Kate83], but not for single-window architectures. The reason for this is discussed in Subsection 2.1.3.1. Thus, the combination of a single-window architecture and an optimizing compiler provides a different conclusion as we will see in Chapter 4.
3. Studies that compare the data memory reduction obtained by a processor with and without multiple windows using an optimizing compiler [Wall88]. The discussion of this approach is postponed until Subsection 2.1.4, after our discussion of interprocedural optimizers.

2.1.2 Single-Window Architectures

In spite of the RSR traffic reduction that multiple-window architectures offer, several designers prefer to have a conventional general-purpose register set (see Table 2.2²). The motivation for this is the drawbacks mentioned earlier in Subsection 2.1.1 for multiple-window architectures.

Traditionally, single-window processors rely on compiler optimizations (such as live-variable analysis and leaf functions) to reduce the register saving and restoring overhead.

²Notice that the CLIPPER processor has been classified as a single-window architecture although its authors claim that it is a multiple-window architecture. The reason is that each *bank* of registers is completely independent of the others. The usage of each bank is determined by the CPU operation mode (system or user) and the type of arithmetic operation to be performed (integer or floating point). Thus, the register file has been partitioned in three independent general-purpose (single-window) register sets.

Processor	Description	References
IBM 801	• 32 general-purpose registers	[Radi82, Chan88]
RIDGE 32 (Ridge Computers)	• 16 general-purpose registers	[Basa83, Folg83] [RID83a]
MIPS (Stanford Univ.)	• 16 general-purpose registers • 6 special regs.: <i>pc</i> , <i>su</i> (<i>psw</i>), <i>Ma</i> (process id.), <i>Ap</i> (addr. mapping), <i>Lo</i> (shift amount), <i>Hi</i> (for * and ÷)	[Henn82, Gros83] [Henn83a, Henn83b] [Gros84, Przy84] [DeMo86, Mous86]
MIPS-X (Stanford Univ.)	• 32 general-purpose registers	[Chow87b]
Spectrum (Hewlett-Packard)	• 32 general-purpose registers ($r0 \equiv 0$) • 25 control regs. (system state information) • 16 floating point regs. (FP coprocessor) • 8 space regs. (to form virtual addresses) • 3 special regs.: <i>pc</i> , <i>psw</i> , instr. addr.	[Birn85, Maho86] [Fotl87]
CLIPPER (Fairchild)	• 3 banks of 16 registers each: 16 for user & 16 for system (32 bits) 8 for floating point (64 bits) • 3 special regs.: <i>pc</i> , <i>psw</i> , system status	[Sach85, Neff86]
Titan (DEC)	• 63 general-purpose registers	[Wall87]

Table 2.2: Single-Window Processors

These are discussed in Subsection 2.1.3.4. To our knowledge, the research presented in Chapter 3 is the first work on architectural support to reduce this overhead. In this section we comment on the only architectural support being offered nowadays for single-window architectures: the grouping of several of the operations mentioned in Figure 1.1 in a few machine instructions.

Some authors [Stre78, Bere82] have claimed that some relatively recent architectures have an efficient calling mechanism: instead of having to specify instructions to perform register saving and restoring explicitly, it is done implicitly by the `call` and `return` instructions. Both the program size and the instruction memory traffic have been reduced because fewer instructions are needed to code high-level-function calls and returns, and fewer instructions have to be fetched. However, the data memory traffic is not reduced because the registers must still be saved and restored by the `call` and `return` instructions in the same way that they were saved and restored by simpler instructions. However, with this approach the total number of cycles required to perform the register saving and restoring might be reduced. For instance, the instruction can pipeline the reading from the register file and the writing to memory (and vice versa).

2.1.3 Register Allocation and Assignment

For load/store architectures the efficient use of the register set is important to obtain a high performance. For these architectures, register-to-register instructions are executed in one machine cycle while load/store instructions require more than one. Thus,

load/store architectures expect the compiler to allocate as many operands as possible in registers so that loads and stores are minimized. The responsibility for this relies on the *register allocator*.

The register allocator is divided into two phases. In the first one, the *register allocation* phase, the allocator selects the *scalar* variables to be allocated to registers. Scalar variables with an alias problem (as it will be defined in Subsection 2.1.3.1) or scalar variables of a certain type (for instance, **double** floating-point variables if the general-purpose register set does not have 64-bit registers) are discarded by the allocator. All scalar variables which can be stored in registers rather than in memory are selected as candidates for allocation. The number of candidates is independent of the number of registers available to the allocator. During this phase, the allocator also keeps track of the variable usage. This information is used to select among the candidates the ones which will be assigned to physical registers. This assignment is performed in a second phase, the *register assignment* phase. The advantage of having two phases instead of only one is that register allocation can be done before code generation and, therefore, can be machine-independent [Site79a, Chow83, McKu84].

Both phases, register allocation and assignment, are traditionally performed at the beginning of the compilation process when the high-level source language is translated into assembly code (see Figure 2.3). Thus, the register allocator does not have a global view of the program, but a partial one limited to the module being compiled. This is something that we have to keep in mind during this discussion to understand the lack of “global” knowledge of the allocator.

In this section we discuss the problem of allocating programmer-defined scalar variables to registers, we introduce the previous work performed in both phases (register allocation and assignment), we comment on the partition of the register set usually done by the compiler, we present the conventional register saving and restoring policies (to save/restore registers at the caller and to save/restore registers at the callee), and we examine the implementation characteristics of some register allocators for load/store architectures.

2.1.3.1 Register Allocation and the Alias Problem

In this subsection we discuss the problem of allocating programmer-defined scalar variables to registers when only one pass is performed by the compiler through the source code. We also mention how this allocation problem is solved either when more than one pass is performed or when some hardware support is provided.

One-pass compilers [Aho86] cannot allocate programmer-defined scalar variables to registers because of the *alias problem*. This occurs when the memory address of a variable is loaded into a pointer or passed as a parameter to a function. In this case, this variable might be referred either by its name or through its address. As a consequence, if the compiler assigns this variable to a register, two copies will be available and the compiler could not guarantee its consistency. The reason for this is that the compiler only performs

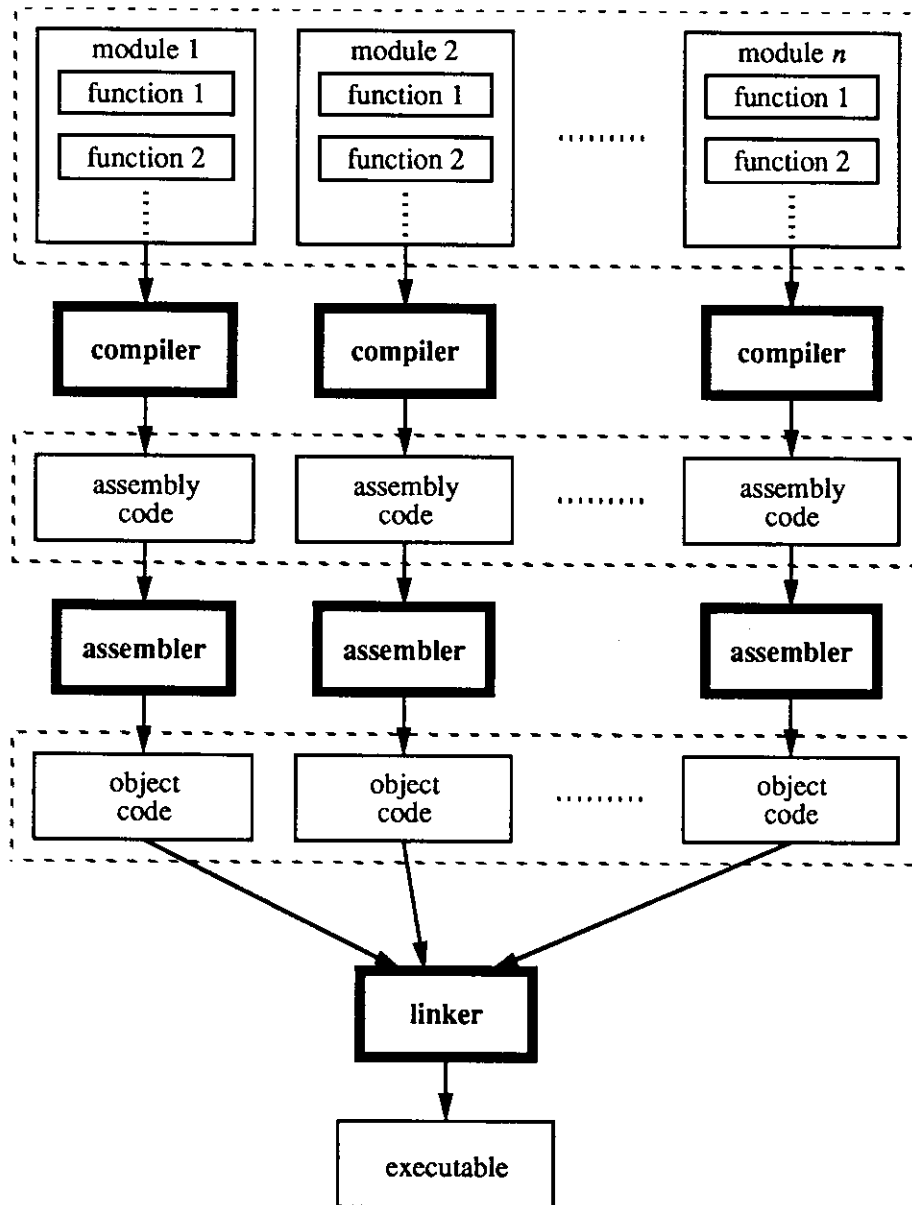


Figure 2.3: A Global View of the Compilation Process

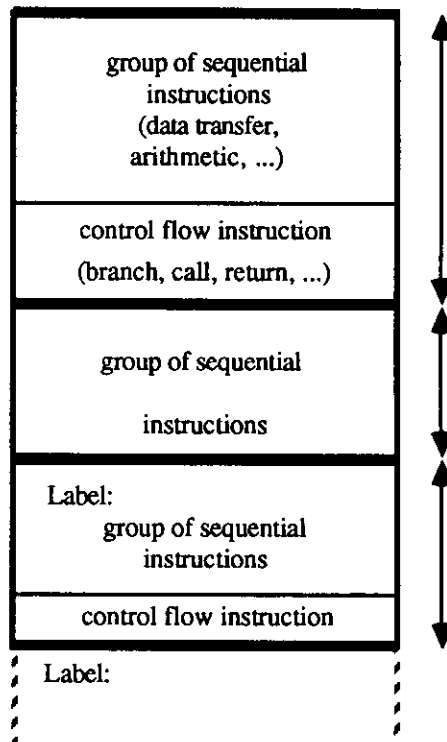


Figure 2.4: Basic Blocks

one pass through the program and, therefore, the storage class (memory or register) of a variable has to be decided at function entry without any knowledge on the variable usage in the function. That is, for one-pass compilers, both the register allocation and assignment phases are performed simultaneously.

Furthermore, one-pass compilers generate code once each statement has been parsed. Thus, only local optimizations per statement are performed (e.g., arithmetic computations of constants are performed during compilation time). To optimize the code further, a *peephole optimizer* [Aho86] is used once the assembly code has been generated. A peephole optimizer improves the code per basic block: eliminates redundant loads and stores, performs some algebraic simplifications, substitutes standard instructions for more efficient ones, etc. A *basic block* is a group of sequential instructions where control flows from the first instruction to the last one without branching outside the block, except from the last instruction (see Figure 2.4) [Back67]. Since no control-flow information is available across basic blocks, a peephole optimizer cannot allocate programmer-defined scalar variables to registers even though at this phase the alias information could be known.

To be able to allocate programmer-defined scalar variables (in one-pass compilers) some programming languages (like C [Kern78] and BLISS [Wulf71]) let the programmer specify which local scalar variables he/she wants to be allocated to registers per function. These variables are called *register variables*. The *address* operator cannot be applied to

register variables so that no alias can be defined for them. Notice that the programmer can only specify a register allocation for local scalar variables, not for globals. The reason for this is to prevent a programmer from allocating a few global scalar variables permanently to registers.

On the other hand, *multi-pass compilers* are able to select for allocation the variables without alias, which have been detected in the first pass. Moreover, in the first phase the compiler collects some information on the variable usage (e.g., the number of times that the variable is referred in the function) to help the register allocator select the variables to be assigned to registers so that the least data memory traffic is generated.

Multi-pass compilers only analyze the source code once. In the first pass, an intermediate code is generated to be used by the following passes. This is to prevent the development cost of an optimizing compiler for every language and every machine [John81]. Machine-independent and language-independent optimizations can be performed directly on the intermediate language. Thus, the register allocator is portable from different source languages to different machines [Perk79, Alle80, Ausl82, Tane83, Powe84]. This is the situation shown in Figure 2.5.

Hardware support can be provided to solve the alias problem in multiple-window architectures [Kate83]. In this case, one-pass compilers perform a more efficient register allocation.³ Each register is *mapped* to a memory location so that the compiler can allocate all local scalar variables defined by the programmer to registers (if enough registers are available in the window), not only the ones explicitly defined by the programmer. When the address of a variable in a register is generated, its memory-mapped address is used so that the variable name and its alias refer to the same location.

As we said in Subsection 2.1.1, the alias problem has to be taken into account to compare the data memory traffic generated by a program in a multiple-window architecture (with hardware support to solve the alias problem) with the one generated in a single-window architecture when a one-pass compiler is used. The data traffic reduction for the former with respect to the latter is a consequence not only of the RSR traffic reduction due to the multiple windows, but also of the number of local scalar variables allocated to registers. In Section 4.4, the RSR traffic generated by an intra-procedural optimizer compiler is compared to the one produced by multiple-windows architectures.

2.1.3.2 Local and Global Register Allocation

This subsection presents the two types of register allocation approaches which can be performed by the compiler: local and global. One-pass compilers can only perform local allocation while multi-pass compilers can perform both.

³Notice that hardware support could also be provided for single-window architectures. In this case, the registers assigned per function should be saved/restored for every call in the corresponding *mapped* memory locations. Thus, this alternative is not attractive for single-window architectures due to the high RSR overhead generated. In contrast, for multiple-window architectures a register window has to be saved/restored only on overflow/underflow.

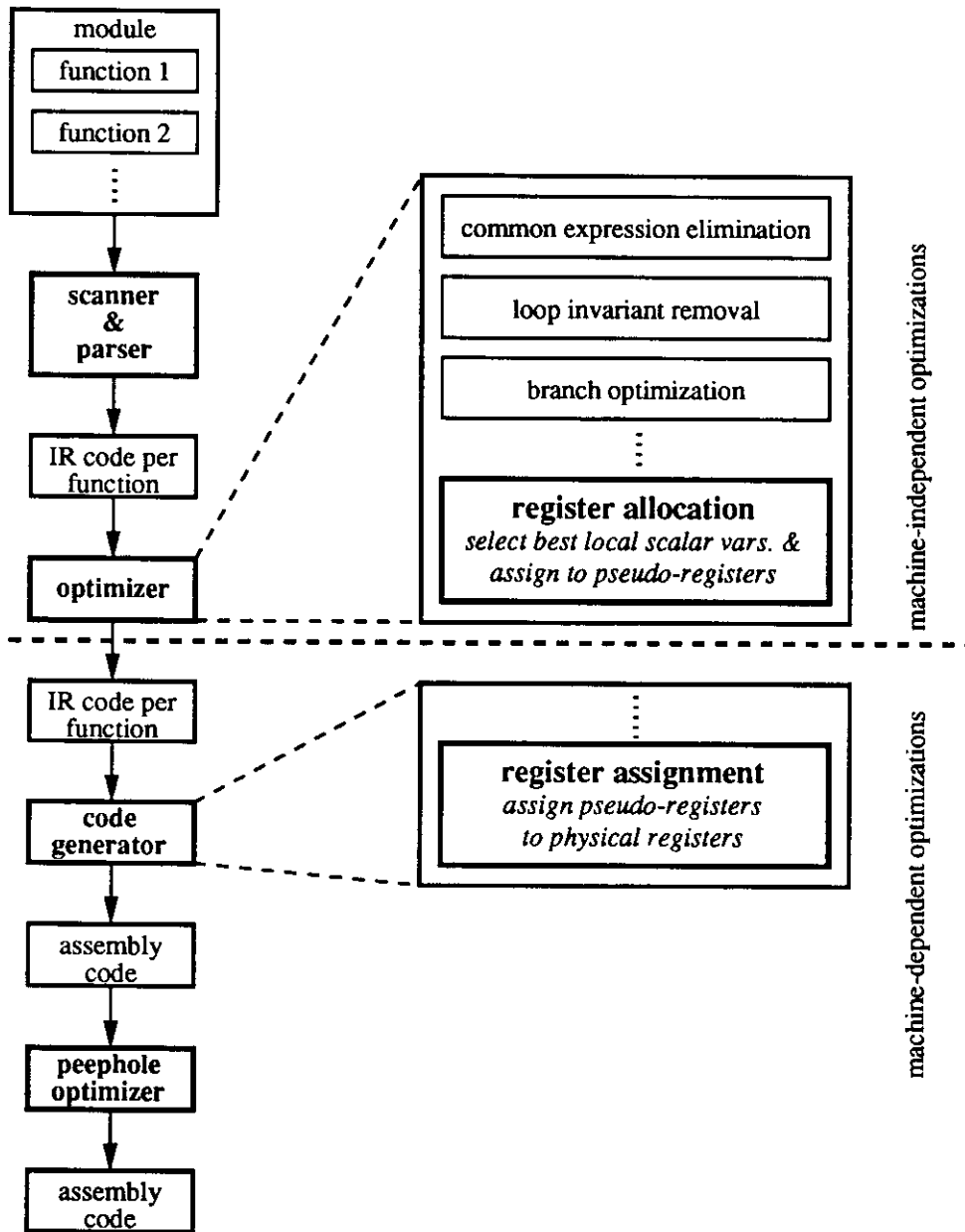


Figure 2.5: Intra-Procedural Register Allocation and Assignment

(1) *Local allocation.* The scalar variables selected for allocation are defined either in an arithmetic expression [Naka67, Redz69, Seth70, Beat74, Seth75] or in a basic block [Horw66, Lucc67, Frei74, Davi84, Hsu87, Good88].

For allocation of registers during expression evaluation, the allocator goal is to minimize the number of instructions required to perform the evaluation of the expression and the number of registers needed to carry it out. The compiler expects that if the smaller number of registers available is used, the number of variables that have to be loaded/stored to memory is minimized.

For block allocation, the allocator goal is to minimize the number of loads and stores for the variables which must be brought to registers. These variables must be loaded to registers because code is being generated for a load/store architecture [Hsu87] or because they must be loaded into an index register [Horw66, Lucc67]. Since the pattern of future references to the variables is known at compile time (inside the block), these algorithms are similar to the optimal page replacement strategy proposed by Belady [Bela66].

Local allocation is useful for temporary variables because these are generated inside the basic block and can be destroyed once the block exits. Variables that have to be preserved across blocks (i.e., *live variables*) have a “main” copy in memory. They must be loaded in each block that uses them and stored back if they are modified. Thus, local allocation performs poorly for these types of variables. One-pass compilers only perform local allocations for arithmetic expressions. Once the assembly code has been generated, a peephole optimizer can optimize the code per basic block (see Subsection 2.1.3.1).

(2) *Global allocation.* While local allocation is performed intra-block, global allocation is performed inter-block. Data-flow analysis [Aho86, Ryde86] is performed to detect the live variables in each block. Live variables do not need to be stored on block exit and do not need to be loaded at block entry (if they have been loaded previously). The use of the name “global” is ambiguous because it might refer to three different scopes where register allocation can be performed:

1. *In a region.* The region usually corresponds to a set of blocks in a loop [Lowr69, Day70, Beat74, Kim78]. The allocator tries to minimize the number of load and stores in the loop, moving them from the most frequent executed blocks to the least frequent ones (if possible).
2. *In a function or procedure (intra-procedural).* In this case, the local scalar variables defined by the programmer and by the compiler in the whole function or procedure are candidates for allocation [John75, Chai81, Ankl82, Miro82, Chow83, Cout86b, Kess86, Laru86]. Since the compiler generates code for each function independently of the others, registers have to be preserved across function calls. The conventional register saving and restoring policies used for this purpose are discussed in Subsection 2.1.3.4.
3. *In the whole program (inter-procedural).* To perform inter-procedural register allocation the program *call graph* (i.e., the relation of which function calls which) is

its partition, it allocates the temporary values to pseudo-registers. These pseudo-registers will compete later for physical registers with the pseudo-registers generated by the intra-procedural allocator.

For multiple-window architectures with fixed-size or multi-size windows the partition of the register set is (almost) determined by the architecture. The maximum number of parameters to be passed through registers is fixed by the number of overlapped registers. For the to be destroyed variables, the compiler can use the OA registers or the common registers. Common registers can be used to allocate global scalar variables only if inter-procedural allocation is performed because an intra-procedural register allocator does not know which global scalar variables have an alias. Local registers and the free IA registers (i.e., registers which have not been used to pass any parameter) are used for the to be preserved variables. A compiler for multi-size windows has to allocate the most suitable window depending on the number of variables to be preserved. Coloring can be performed to reduce the number of registers in a window and to increase the number of variables that will be assigned to registers.

Variable-size windows give the compiler more flexibility because no restriction on the number of parameters to be passed through registers is imposed by the architecture and the exact number of registers required by the function can be allocated (with the only limitation being the maximum window size). Coloring can also be performed to reduce the number of registers in the window. Moreover, the use of the `catch` instruction (the instruction that is executed at the caller after return to indicate the number of registers to be restored in case of underflow [Ditz82]) can be optimized. For instance, the `catch` between a function call and a return can be eliminated because there are no references to the registers after return [Band87]. In this case, the restoring traffic might be reduced if registers are not present in the register file.

2.1.3.4 Conventional Register Saving and Restoring Policies

The *conventional* register saving and restoring policies only apply to intra-procedural register allocators for single-window architectures. They do not apply to multiple-window architectures windows because these are saved/restored during overflow/underflow or during program execution for dribble-back register files (see Subsection 2.1.1) and to inter-procedural allocation because there are as yet no *conventional* RSR policies since it is a current research topic. Subsection 2.1.3.5 mentions the RSR alternatives available for inter-procedural allocators.

There are two conventional register saving/restoring policies for intra-procedural allocation:

1. To save the registers in the caller (before the call) and restore them upon return (after the call). We call this approach **Policy A**. If *live-variables analysis* has been performed [Hech77, Aho86], only registers which contain live variables are

saved/restored. Otherwise, all the registers defined in the function have to be saved/restored.

2. To save the registers in the callee (at function entry) and restore them before (or during) the return. We call this approach **Policy B**. This policy is usually used with two partitions so that all the TBP registers defined in the function are saved and restored, but the TBD registers are used without having to save/restore them. Moreover, if the function is a *leaf function* (i.e., it does not call any other function), then the TBD registers can be used to assign variables [Chow86, Cout86b]. In this case, no (or a few number of) TBP registers have to be saved/restored.

For a particular register saving/restoring policy, a *threshold* is defined at which the register allocator bases its decision to assign a variable (or pseudo-register) to a physical register. The goal is to prevent the case where the traffic generated by a variable in memory is less than the RSR traffic generated when this variable is assigned to a register. This threshold is easier to determine for Policy B than for Policy A, because for the former a variable assigned to a register generates two memory references every time that the function is called. Thus, if the variable is referred to at least twice during function execution, less data memory traffic is generated when the variable is assigned to a register. On the other hand, for Policy A a variable assigned to a register is saved/restored at each call, if it is alive when live-variable analysis is performed. Thus, the traffic reduction depends on the number of calls generated and the number of times that a variable is referred to. These numbers are more difficult to estimate during compilation time than the simple two references for Policy B.

Although compilers currently use these conventional policies, the author has only been able to find one performance evaluation study which compares both of them. McKusick [McKu84] has evaluated the size and execution time of the whole set of UNIX utilities for both policies. Since instructions to save/restore the registers must be specified in every call instruction for Policy A and only once (at the entry point) for Policy B, the programs compiled with Policy A were 8% larger than the ones compiled with Policy B. Moreover, since the running time of the programs was not much different, McKusick concluded that Policy B is more efficient.

However, the compiler that he used to perform these measurements, the Graham-Granville compiler [Henr84], is a one-pass compiler and, therefore, only explicitly-defined variables are allocated to registers. Moreover, no live-variable analysis is performed by the compiler, and thus all the assigned registers in the function have to be saved/restored. As a consequence, McKusick's conclusion might be true for a non-optimizing compiler, but not for an optimizing one with live-variable analysis. In fact, this is what we found from our work: when neither live-analysis optimization is performed for Policy A nor leaf-function optimization for Policy B, Policy B always generates less RSR traffic than Policy A, as we will show in Chapter 3. However, when these optimizations are performed, this is no longer true, as we will show in Chapter 4.

2.1.3.5 Register Assignment

In this phase, the pseudo-registers with the variables selected for allocation have to be assigned or *mapped* to a physical register. This mapping depends on how the general-purpose register set has been partitioned by the compiler writer as discussed in Subsection 2.1.3.3. In this subsection we present two approaches for intra-procedural register assignment, and then summarize how three specific intra-procedural register allocators assign variables to registers. Last, we discuss two approaches for inter-procedural register assignment.

When more pseudo-registers (with a priority above the RSR threshold and with disjoint live times if coloring is used) are available than physical registers, the compiler can:

1. Assign only a group of pseudo-registers to physical registers and assign the remainder of these pseudo-registers to memory (based on some static usage frequency).
2. Insert an instruction to release a “busy” physical register in a specific block (i.e., to store a register to memory) so that it can be loaded with the variable required in this block. In this case, another instruction to load this variable to a register will have to be inserted later when the variable is needed. These instructions are referred as *spill* instructions.

In the former case, data memory traffic is generated for the variables that have not been allocated to registers, while in the latter, it is generated for the spill instructions. Although the second approach seems more attractive because it allows the sharing of registers by all the variables defined in the function, it might generate more data memory traffic. The author has not been able to find any study to confirm or deny this hypothesis. The three compilers used in this dissertation use the first approach so that we have not been able to compare these two approaches; thus, we have left this evaluation as an open topic for discussion (see Section 6.2).

The register assignment depends on the optimization degree of the compiler. From all the possible register allocation combinations which have been presented in Subsection 2.1.3.2, in this subsection we will only comment on three intra-procedural allocators. These allocators cover the three possible partitions of the register set and correspond to current compiler technology. Thus, our discussion is simplified without any loss of generality.

1. One-pass compiler with local register allocation and with programmer hints for local scalar variables. This is used by the Portable C Compiler⁵ [John78, John79]. The compiler uses two independent partitions. If the programmer has specified more register variables than the number of TBP registers, only the ones defined

⁵Intra-procedural register allocation is performed in this case by the programmers themselves, since the compiler only assigns explicitly-defined register variables to registers (see Subsection 2.1.3.1).

first are assigned to physical registers. The reason for assigning the first-defined register variables is that only one pass is performed so that the storage class of a variable is determined when its definition is found (see Subsection 2.1.3.1). Notice that the compiler does not use spill instructions.

2. Intra-procedural allocation without local allocation. This is used by the PL.8 Compiler [Chai81]. The compiler uses a unique partition. Optimizing variables, local scalar variables, and temporary values that have been allocated to pseudo-registers compete for the same pool of physical registers. Variables with disjoint lifetimes are assigned to the same register. When no more registers are available, the register selected to be spilled is the one with more interferences in the color graph (i.e., the node with more arcs).
3. Intra-procedural and local allocation. This is used by the machine-independent UOPT optimizer [Chow83]. The UOPT optimizer uses two dependent partitions. When the local allocator has used all its registers, it allocates the temporary values to pseudo-registers. Individual registers for each allocator can be shared if they have disjoint lifetimes. The priority of a variable is considered to decide which register to spill (in contrast to Chaitin's allocator [Chai81]).

We now present two approaches for assigning pseudo-registers allocated by an inter-procedural optimizer. These are:

1. To assign only the variables that fit in the register partition. This is the approach followed by Wall [Wall86]. In this case, a variable is selected based on:
 - (a) the priority of the function where the variable is defined and
 - (b) the priority of the variable with respect to the other variables defined in the same function.

Notice that in this case, no RSR traffic is generated.

2. To combine intra-procedural register allocation with inter-procedural register assignment. This is the approach taken by Steenkiste [Stee87] and Chow [Chow88]. Registers are assigned first for the leaf functions in the call graph. Second, registers for the callers to the leaf functions are assigned. The registers assigned are the ones that are not being used by the function descendants. If both the caller and all its descendants have enough free registers in the partition, no RSR instructions have to be generated. The process continues until the descendants have exhausted the registers in the partition. In this case, RSR instructions (Policy A with live-variable analysis) have to be inserted to preserve registers across function calls.

Notice that the RSR instructions are eliminated only from the functions at the bottom of the call graph. Steenkiste claims that this is a valid approach for LISP programs since 60–70% of the calls are done to leaf functions or their immediate ancestors (for 10 small programs and 3 large ones). However, this might not be true for any programming language or application. For this reason, Chow's register

allocator has a priority criterion which defines when a register has to be taken by the function at the bottom (so that the register saving/restoring instruction can be eliminated) and when the register has to be left for a function up in the call graph.

In Section 5.4 we present our approach, which also combines intra-procedural register allocation with inter-procedural register assignment. Our goal is to perform a disjoint register assignment for the functions that generate the most RSR traffic, not necessarily the ones at the bottom of the call graph. A comparison of our approach with Steenkiste's and Chow's register allocators is given in Subsection 5.4.1 and with Wall's in Subsection 5.5.1.

Table 2.3 shows the characteristics of several register allocators for some of the processors given in Tables 2.1 and 2.2. The contents of the table should be self-explanatory, except for the notation $sp \equiv fp \equiv ap$, which means that the stack pointer is used as frame pointer and as argument pointer. When the function's activation record is fixed during its execution and, therefore, locals and parameters are referred with respect to the stack pointer, no environment register has to be saved/restored across function calls [Wulf75].

2.1.4 Performance Studies on Register Allocation

No major performance study has been found in the literature that compares the alternative register allocation approaches discussed in the previous section. In this section we comment on a few individual evaluation studies of register allocators. Our goals with these studies are:

- To determine a suitable size of the register file based on the number of registers required by the local and intra-procedural allocators.
- To evaluate the percentage of data memory references eliminated when variables are assigned to registers.

The measurements obtained by different studies cannot be easily compared, because the performance methodology used by each study is different: some are static while others dynamic; some include RSR traffic while others ignore it; some include the traffic generated by both local and intra-procedural allocators while others only the traffic generated by the latter; etc. Also, the size of the programs measured and the programming languages used are different.

Local Allocators

Davidson and Fraser [Davi84] have shown that a local optimizer with only 3 registers available generates only 22 spill instructions compiling the Y compiler (a 3500-line pro-

Compiler for	Description	References
RISC I & II	<ul style="list-style-type: none"> • Portable C Compiler (one-pass compiler; intra-procedural) • register allocation: Sequential by definition alias problem solved by hardware • register set usage: $r0 \equiv 0$ $r1-r3$: environment registers ($r1 \equiv sp$) $r4-r9$: temporary values (to be destroyed) $r10-r15$: incoming arguments & return address $r16-r25$: local regs.; $r16 \equiv fp \equiv ap$ $r26-r31$: only used for outgoing args. • 1 window transferred on overflow/underflow 	[Miro82] [Tami83]
IBM 801	<ul style="list-style-type: none"> • register allocation by graph coloring (intra-procedural; Policy A with live information) • parameter passing through registers 	[Chai81] [Ausl82] [Chai82]
RIDGE 32	<ul style="list-style-type: none"> • only integer register variables allocated • register set usage: $r0-r5$: to-be-destroyed registers $r6-r13$: to-be-preserved registers ($r11 \equiv ret. \text{ addr.}$) $r14 \equiv sp$; $r15 \equiv fp \equiv ap$ 	[RID83b]
MIPS	<ul style="list-style-type: none"> • reg. allocation by priority-based coloring (intra-procedural) • parameters passed through registers (up to 4) • leaf-function optimization • $sp \equiv fp \equiv ap$ 	[Chow83] [Chow84] [Chow86] [Chow88]
SOAR	<ul style="list-style-type: none"> • one pass compiler (intra-procedural allocation) • IA regs.: 2 for return address & value 6 for arguments & to-be-preserved variables • OA regs.: 2 for return address & value 6 for callee args. & to-be-destroyed temporaries • no usage indicated for common registers 	[Bush87]
SPUR	<ul style="list-style-type: none"> • reg. allocation based on [Chow83] (intra-procedural) • register set usage: common: 2 dedicated for loads/stores in spill 6 (unspecified) special usage IA: 1 for return address & 1 for number of args. 4 for parameter passing local: 1 dedicated to point vector for constants 9 available (plus free IA regs.) OA: only used for parameter passing 	[Laru86]
HP-Spectrum	<ul style="list-style-type: none"> • intra-procedural register allocation (Policy B) 16 regs. to-be-preserved & 13 regs. to-be-destroyed 3 environment registers ($sp \neq fp$) • parameters passed through registers (up to 4) • leaf-function optimization 	[Cout86a] [Cout86b] [Mage87] [Mage88]
CRISP	<ul style="list-style-type: none"> • stack compression by live/dead analysis • catch instruction optimization • $sp \equiv fp \equiv ap$ 	[Band87]
Titan	<ul style="list-style-type: none"> • inter-procedural register assignment 52 registers available no register saving/restoring overhead 	[Wall86]

Table 2.3: Features for Several Register Allocators

gram written in Y). Thus, they concluded that the number of registers required for local register allocation is small.

Our measurements on three C programs (NROFF, SORT, and the Portable C Compiler) showed that 2 TBD registers are enough to evaluate 95% of the executed arithmetic expressions and that 3 TBD registers are enough to pass parameters through registers for 98% of the executed functions [Hugu85a].

Flynn *et al.* [Flyn87] have measured the number of registers required by Chow's local allocator [Chow83] for five large PASCAL programs (a desk calculator emulator, a comparator for two text files, a PASCAL compiler, a P-code assembler, and a macro processor). They concluded that the data memory traffic generated by the local allocator does not become smaller for more than 2 TBD registers. In this case, 18% of the (stack) data memory traffic is eliminated.

Therefore, the number of registers required by a local allocator is small. For this reason, we ignored local allocation in this dissertation.

Intra-Procedural Allocators

The design of a register set was studied by Lunde [Lund77]. He measured six algorithms coded in four different programming languages (ALGOL, BASIC, FORTRAN, and BLISS) and concluded that the register set should include two floating-point accumulators, two fixed-point accumulators, and eight registers for simple fixed-point operations. Since the average number of registers used by the compilers for these algorithms was 3.9, these numbers seem appropriate for the compiler technology available at that time.

We also studied the size of the general-register set for C programs [Hugu85a]. We measured the number of local scalar variables defined per (executed) function in NROFF, SORT, and VPCC to estimate the number of TBP registers for intra-procedural allocators (when neither live-variable analysis or leaf-function optimization is performed). Our old measurements showed that 75% of the functions have up to three local scalar variables (including the arguments passed to the function), 12% have between 4 and 6, 9% have 7 or 8, and 3% have between 9 and 14. Therefore, our measurements indicated that there is no reason for having more than 12 TBP registers available to the allocator.

Chow [Chow83] has measured the percentage of variables assigned to registers and the percentage of variable references found in registers by his priority-based coloring algorithm for two different machines, DEC-10 and MC68000. The compiler has 9 registers available for the first machine and 6 data and 4 address registers for the second. The programs measured were 12 small programs (Towers of Hanoi, Queen, Puzzle, Quicksort, Erasthenes Sieve, etc.). On the average, the compiler assigns 76% of the variables to registers for DEC-10 and 86% for MC68000. These variables account for 77% and 87% of the (static) variable references to memory, respectively. The difference between the number of variables assigned to registers is not due to the extra register available in MC68000, but to the register saving/restoring policy. The compiler for DEC-10 uses

Policy A and Policy B for MC68000. Chow claims that the allocation cost (i.e., the threshold) is lower for Policy B and, therefore, more variables are allocated.

Flynn *et al.* [Fly87] have measured the number of registers required by Chow's priority-based coloring allocator (for the five PASCAL programs given above). They concluded that the data memory traffic (including RSR traffic) generated by the intra-procedural allocator does not become smaller for more than 8 TBP registers. For 2 TBP registers, 30% of the (stack) data memory traffic is eliminated; for 4, 35%; and for 8 and up, 37%.

Larus and Hilfinger [Laru86] have also measured the performance of Chow's register allocation algorithm for LISP programs in SPUR. They have eliminated Chow's local allocator so that temporary variables are also allocated by the intra-procedural allocator. The programs measured were the Spice LISP interpreter, the LISP compiler for SPUR, and a circuit simulator. As we indicated in Table 2.3, the optimizer has 9 local registers available, plus the free IA registers. On the average, the compiler has to color about 42% of the functions and only has to insert spill code for 5% of them. Thus, about 95% of the defined functions require up to 11.5 registers (9 locals plus an average of 2.5 free IA registers). They also investigated the number of registers required by the allocator for only one of the programs (the SPUR LISP Compiler). They concluded that 70% of the defined functions need up to 5.4 registers, 92% need up to 8.4, 97% need up to 11.4, and 99.1% need up to 14.4 registers.⁶

Inter-Procedural Allocators

Wall [Wall86] used 52 registers for his inter-procedural allocator. He has measured six programs: four small ones (Livermore Loops, Whetstone, Linpack, and the Stanford Benchmark Suite) and two medium-sized ones (a logic simulator and a timing verifier). Since we do not consider the behavior of small programs typical of a real system workload (see Section 1.3), we concentrate our attention on the last two. Once register allocation has been performed, 73% of the dynamic memory references generated without register allocation have been eliminated for the simulator and 52% for the verifier.⁷ To increase the number of memory references eliminated, Wall performs intra-procedural coloring,⁸ execution profiling (to know the number of memory references generated by each variable so that a better variable selection can be performed), and a combination of both. In this case, the percentage of dynamic memory references eliminated is 83% (coloring), 92% (profiling), and 95% (both) for the simulator and 61%, 78%, and 83% for the verifier, respectively. In a later paper [Wall88], Wall compared his inter-procedural register allocator with multiple-window register files and concluded that the overall memory traffic is equivalent for both schemes.

⁶These numbers have been computed adding the 3, 6, 9, and 12 local registers with the average of IA registers which do not contain any argument (2.4).

⁷These percentages are given over "[the loads and stores] that we can remove by keeping some scalar variable in a register instead of memory," not over the total data memory traffic.

⁸He calls it "local" coloring (*sic*).

Steenkiste [Stee87] has implemented his inter-procedural register allocator on MIPS-X for LISP. He has measured 10 small LISP programs. When only intra-procedural register allocation is performed, 55% of the stack references are eliminated (including RSR traffic). When inter-procedural register assignment is made, only 22% of the stack references remain. Since his register assignment is simpler than Wall's and the number of registers used is also smaller, we expect that the traffic reduction would be smaller for larger programs.

Flynn *et al.* [Flyn87] have also measured the number of registers required by Steenkiste's inter-procedural register allocator (for the five PASCAL programs given above). The compiler eliminates 47% of the data memory traffic for local scalar variables (including the RSR traffic) when 8 registers were available to the inter-procedural allocator, 55% when 16, 60% when 32, and 62% when 64. There is no data memory traffic reduction for more than 64 TBP registers.

Chow incorporated an extension of Steenkiste's inter-procedural allocator to his intra-procedural optimizer [Chow88]. The optimizer eliminates between 4.5% (for `awk`, a 2,500-source-line program written in C) and 57.6% (for `calc`, a 500-source-line program written in PASCAL) of the scalar load and store instructions. However, the inter-procedural optimizer generates more traffic (16%) for the Portable C Compiler because some loads and stores are moved up to the call graph to functions which are more frequently executed. Chow claims that this problem could be solved if dynamic information were available.

Therefore, as we said at the beginning, we cannot compare the performance of these register allocators based on the measurements available. Our conclusion from this discussion is that recent advances in compiler technology make it possible to use efficiently a larger register set by the register allocator. For this reason, modern processors are increasing their number of general-purpose registers: 16 registers for RIDGE 32, MIPS, and CLIPPER; 32 for RISC, MIPS-X, SOAR, SPUR, IBM 801, and HP-Spectrum; and 64 for PYRAMID 90x, CELERITY C1200, and Titan. As we will discuss in Chapters 3, 4 and 5, there is no need for having more than 12 TBP registers for the non-numeric programs measured (ASM, NROFF, SORT, and VPCC) unless the dynamic Policy G and/or the inter-procedural optimizations are available.

2.1.5 Performance Evaluation of New Architectures

To design a new processor or to modify an existing one, designers need to estimate the influence of specific architecture features on the performance of the processor. To perform these estimates, it is necessary to measure the *the dynamic behavior of typical programs*. The measurements let the designer (1) discover which architecture features are critical in the design so that these features can be tuned to improve performance; (2) compare several implementation alternatives of the same architecture; and (3) compare the performance of different architectures.

For the processor that is being designed, called the *Proposed Machine* (PM), a model that defines the processor characteristics must be used. Traditionally, this model is de-

ned either by a *simulator* or an *emulator*. (For existing processors one can use software, firmware, or hardware monitors [Svob76, Ferr78]; however, these are not discussed in this dissertation because we are interested in the development of new processors.) The machine where the measurements are performed is called the *Existing Machine* (EM). In this section we present the tools that are currently used for collecting measurements: simulators, emulators, and step-by-step instruction tracers. We describe their limitations and introduce a new generation of measurement tools for architectural evaluation.

In a simulator the processor characteristics are described in a high-level language or in a specialized language [Chu74] such as ISPS [Barb81]. On the other hand, an emulator uses the microprogrammable features of an existing general-purpose microprogrammable machine to describe the new processor. The advantage of a simulator versus an emulator is that the former is easier to design, to modify, and to maintain. These characteristics are important if we need a flexible tool for studying several architectural modifications. Moreover, since a simulator is written in a high-level language or a specialized language, it is portable to different general-purpose processors (if a compiler for the language is available) and does not require a general-purpose microprogrammable processor.

The main disadvantage of a simulator is its execution time. Since several hundreds of machine language instructions are required to decode, interpret, and measure each simulated instruction, the total execution time of the simulated program is several orders of magnitude larger than if the program were executed directly [Tami81, Rose84]. For instance, the execution time of the Berkeley RISC simulator simulating some of the CFA benchmarks [Full77] varies from approximately 100 to 400 times their normal execution time [Hugu87]. Therefore, the simulation time of typical programs is prohibitively large, and, as a consequence, designers tend to simulate only small programs. These programs might not be representative of a typical system workload (see Section 1.3).

Some machines provide a trace option that transfers execution control to an exception handler after each instruction is executed. This is provided mainly for debugging purposes, but can also be used for collecting dynamic measurements [Eick87, Hsu87]. The modifications that can be simulated with this method are limited because the instructions cannot be modified since they must be executed by the machine itself. For instance, a tracer can measure the performance of a multiple-window register file in a single-window architecture because no information has to be included in the instruction itself for this architectural modification. However, since the number of bits in the instruction for the register number cannot be changed, the measurements obtained by this method are limited to a maximum window size equal to the existing register-set size.

The advantage of using a tracer instead of a simulator for collecting measurements is that the instruction is directly executed by the machine so that the tracer only needs partially to decode and interpret each instruction to extract the measurements which are being collected. On the other hand, a major disadvantage of using the tracer is that, due to operating system overhead, it is too slow for measuring large typical programs. For example, consider two possible implementations of the tracer on a VAX-11 [DEC79] under 4.3BSD UNIX. In the first scheme, the tracer controls the execution of the program being measured and examines its state using a set of system calls (`ptrace` [UNI81]). In the

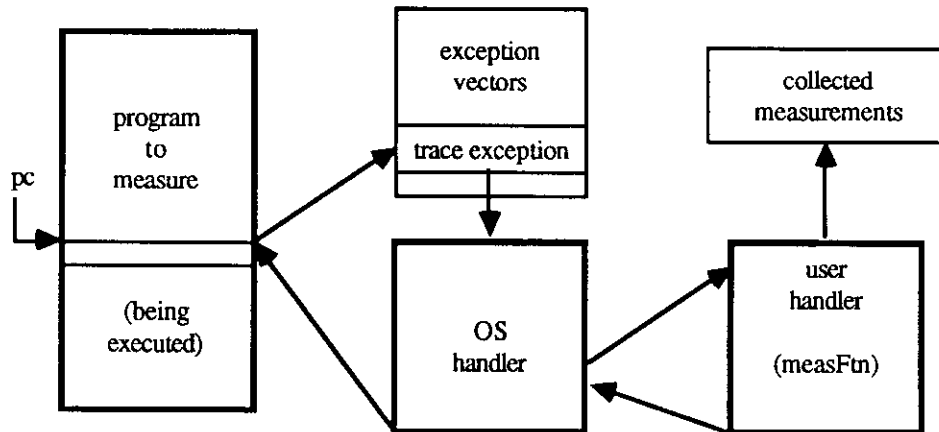


Figure 2.7: Gathering Measurements with the Step-by-Step Instruction Tracer

second scheme, the trace bit of the processor status word is set and the tracer is defined as the exception handler for trace exceptions. Figure 2.7 shows this second implementation. The execution time for counting the number of machine instructions executed⁹ for the UNIX SORT program is 1,540 times its normal execution time when `ptrace` is used¹⁰ and 540 times when `tracer` is implemented as the trace exception handler.

Although the instructions are executed directly by the EM instead of being interpreted by a simulator, the overhead introduced by the operating system and the limitations on the measurements which one can perform make this alternative less attractive than simulation.

Therefore, the simulation time of typical programs is prohibitively large and, as a consequence, designers tend to simulate only small programs. For this reason, we have designed a new tool for collecting architectural measurements: the Block-and-Actions Generator (BKGGEN). This has been presented in Section 1.2. Here we will discuss similar alternatives to a simulator known to the author.

Campbell [Camp85] has implemented a *fast* simulator for the MC68000. This simulator works as follows: (1) the programs measured are compiled to MC68000 assembly code; (2) the assembly code is decompiled to C; (3) code is inserted to perform the measurements; and (4) the “high-level” program is translated to the EM assembly code to be executed. To reduce the overhead even further, measurements are collected for the whole basic block. This generates less overhead than performing the measurements for each simulated instruction. Campbell concluded that the overhead introduced by the simulator is between 6 and 28 times the normal execution of the programs measured: a simple incrementing loop, the CFA H benchmark, and a recursive solution to the Knight’s Tour

⁹This can be considered the simplest type of measurement and, therefore, the observed overhead is the minimum overhead introduced by the different measurement techniques.

¹⁰Note that no operating system call to examine the state is necessary for counting the number of instructions executed. In measurements where we need more detailed information about each instruction, the overhead would be much larger.

problem.

To verify the code generated by the MIPS compiler (before the hardware was available) an *object-to-object translator* from MIPS (the PM) to VAX-11 (the EM) was designed. This is called Moxie [Chow86]. Moxie translates MIPS instructions to VAX-11 instructions as well as MIPS UNIX system calls to VAX UNIX system calls. Moxie also provides tracing facilities which perform instruction counting and cache simulation. MIPS code is “simulated” at the rate of 600K instructions per second. Since MIPS is an 8-MIPS machine [Gima87], the overhead introduced is about 13 times the normal execution of the programs.

Cortadella and Llabería [Cort87b] have implemented an *assembly-to-assembly translator* from RISC to VAX-11. Each RISC basic block is identified by a *unique block number* and translated to VAX-11 code. At the beginning of each block, code is inserted to obtain the program execution trace given as the sequence of block numbers executed. Once the execution trace is known, measurements for different PMs can be obtained because the program behavior is the same for any machine. Cortadella and Llabería have measured the execution time of a small program (Quicksort) running in MicroVAX-II (2 seconds) with a conventional simulator (15 minutes) and with their new assembly-to-assembly translator (6 seconds). Therefore, they have reduced the overhead from a factor of 450 to a factor of 3.

May [May87] has implemented MIMIC, a fast simulator for the IBM 370 on an IBM RT PC. The simulator receives executable IBM 370 code. Instead of interpreting each instruction as a conventional simulator would do, MIMIC performs *incremental translation*. Basic blocks are grouped in a *code block*. The code block corresponds to an execution sequence such that the code block is reachable from outside only at its first instruction and it can be exited through its last. Thus, a code block could correspond to a procedure. Once the code block has been translated, it is directly executed. When the code block exits, the destination block is located. If it has already been translated, direct execution proceeds; otherwise control-flow analysis is performed to determine the largest grouping of basic blocks, the new code block is translated, and then direct execution can proceed.

The advantage of this approach to the previous ones is that the translator has information with respect to the whole code block so that a better use of the resources (EM registers) is made, because the other approaches translate each PM instruction individually. Thus, MIMIC can perform “global” register allocation rather than the local allocation used by the previous two approaches. May concluded that the *expansion factor*, i.e., the number of EM instructions executed per PM instruction is between 2.7 and 4.4 for the two large programs measured (an interpreter and a file encipher/decipher program).

2.2 Related Work

There are two topics that are closely related to registers, but that are not discussed in detail in this dissertation. For this reason we did not want to include them in the previous section. These are:

- The advantages of having registers in the processor instead of having a unique memory address space with a cache memory to obtain fast access to the operands.
- The selection of the most appropriate instruction format to address the general-purpose registers.

In this section we offer a general discussion on these topics and some references where the interested reader is referred to obtain more information.

2.2.1 Cache versus Registers

Cache memories effectively reduce the processor data memory traffic [Smit82]. Data memory traffic can be divided into three different streams:

- Instructions.
- Global scalar variables and non-scalar data (arrays and structures either local or global).
- Local scalar variables and compiler generated variables which perform expression evaluation and store optimizing variables for loop-invariant expression, for common subexpressions, etc.

Although most prefer having a cache for the first two streams, there is some debate about the best alternative for the local scalar variables. Some architects would prefer to have a unique address space (memory) with a cache memory for fast access to operands [Ditz82, John82, Wirt86], while some others would have two address spaces (memory and registers) [Kate83, Hugu85a, Hsu87].

The drawbacks for having registers are:

1. Registers have to be saved/restored on function calls with the overhead that we have already mentioned in Section 1.1.
2. Registers can only store scalar data.
3. Registers are not equivalent to memory, thus, some operations performed on memory data might not be applied to data on registers. For instance, the address operator cannot be applied to a variable assigned to a register unless hardware support is provided (see Subsection 2.1.3.1).

4. Registers increase the complexity of the compiler because of the multiple passes required for register allocation, while stack architectures simplify the compilation process, mainly for code generation. This results in smaller and faster compilers. For instance, the Lilith PASCAL compiler is one third the size of the Motorola compiler [Wirt86].

While both cache memory and registers are valid methods to get fast access to operands, there are still advantages for having registers:

1. Cache memories create overhead because they check whether the data is available. Furthermore, if the data is not available, the processor has to wait for a new block from main memory.
2. Cache memories do not reduce the instruction length because a full address specification is still required to refer to a local (stack) operand. The address of a local variable stored in the activation record is usually specified as a displacement relative to the frame pointer. Fewer bits are required to specify a register address than to specify the displacement.
3. The addressing mode specifier has to be decoded and the operand address has to be computed every time the instruction is executed. This overhead can be reduced if the virtual address of the operands (once computed) is stored in the instruction cache [Nort83, Ditz87c].
4. It has been shown that the reference pattern for local operands is different than the one generated by global references [Hsu87]. In a similar way that a split instruction/data cache increases system performance [Smit85], the removal of local scalar operands from the data cache might make its design parameters more tailored to obtain a better performance.
5. When local operands are stored in registers by the compiler, fewer addressing modes are necessary to be provided by the architecture [Chow87a].

We believe that the advantages for having registers outbalance their drawbacks and that the compiler complexity for a register-oriented machine is not a significant argument, because:

- The complexity of the compiler is reduced by having machine-independent register allocators [John75, Leve83, Chow83].
- With the current trend toward providing cache control instructions to the compiler to reduce unnecessary data memory traffic for dead data [Radi82, Birn85] or toward performing live-variable analysis and static frequency usage to increase the cache hit ratio [Band87], the cache memory is no longer hidden at the compiler. Thus, the complexity of the compiler has to be increased to manipulate the cache memory.

Therefore, we conclude that registers will still be provided in future processors and that we have to study the mechanisms (in hardware and software) to reduce their main drawback: the overhead caused by register saving and restoring.

2.2.2 Instruction Format (RISC versus CISC)

There has also been major discussions among computer architects about what is the most appropriate instruction format: 0-address, 2-address, register-to-memory, or memory-to-memory [Myer77, Schu77, Keed78b, Myer78, Keed78a, Site78, Keed79, Myer82, Keed83, Wong89]. At the beginning of this decade, several processors (RISC I and II, IBM 801, MIPS, etc.) were introduced with a new architectural design style called *Reduced Instruction Set Computers* (RISC) to counter the tendency of increasing the complexity of the processors (which are called CISCs, *Complex Instruction Set Computers*).

There is some debate on the elements which define a Reduced-Instruction-Set-Computer approach [Hopk84, Patt85a, Wall85, Taba86, Gima87]. Patterson [Patt85a] has characterized the existing RISCs as having the following: operations are only register-to-register; memory operands must be loaded first to a register to be operated; instruction decoding is simpler; operations and addressing modes are reduced; and branches avoid pipeline penalties. The primary goal is to execute fast the small number of instructions which occur frequently [Alex75, Shus78, Swee82, McDa82, Wiec82, Clar82]. As a consequence, the previous controversy about instruction format is now centered around RISC and CISC architectures and the direction that computer architects should take in their designs [Patt80, Clar80, Nort83, Heat84, Colw85, Patt85b, Davi87, Flyn87, Borr87].

With the appearance of the RISC architectures, a general-purpose register file of 16 or 32 registers became the *de facto* standard for modern processors. It seems that choosing the number of registers is based on the instruction format provided by the processor: 2-register-address architectures have 16 registers (due to the limitation of the 16-bit instruction format to specify a register-to-register operation) and 3-register-address architectures have 32.

Since it has been claimed [Chow87a, p. 300] that:

The goal of any instruction format should be:

1. *Simple decode,*
2. *simple decode, and*
3. *simple decode.*

Any attempts at improved code density at the expense of CPU performance should be ridiculed at every opportunity.

the 3-register-address instruction formats have become the *de facto* standard for today's RISC processors. For instance, MIPS-X has eliminated the 2-address instructions its predecessor (MIPS) had.

However, some architects, mainly pro-CISC designers, claim that the addition of more instruction formats reduces the program size [Davi87, Flyn87]. The advantages of a smaller program size need to be recognized: the reduced space required to store the program (in disk); the reduction in the number of pages in memory which is necessary to keep the program working set; and, most importantly, the reduction in the size of the instruction buffer. For instance, Flynn *et al.* [Flyn87] have shown that the addition of a memory-to-register format to a simple load-store architecture can cut the instruction buffer size by half yet still offer the same performance (given as the hit ratio). This is important for VLSI systems because the area available in the chip is limited.

Therefore, it seems that this problem is still open to debate. To be able to increase the complexity of the instruction format with the addition of register-to-memory instructions the following should be true:

1. The decode phase of the processor (in a pipelined system) should be able to interpret more complex instruction formats without increasing the processor cycle time.
2. The number of operations required to execute the instruction (number of reads and writes from the register file) should not increase the number of stages in the pipeline (versus a simple load-store architecture).

However, if this is not the case, we can expect that the 3-register-address instruction format will provide the highest throughput rate (1 instruction per cycle except loads and stores).

Chapter 3

RSR Architectural Support for Single-Window Register Files

With an intra-procedural register allocator, the conventional register saving/restoring (RSR) policies used in single-window architectures are: to save/restore registers at the caller (**Policy A**), to save/restore them at the callee (**Policy B**), or a combination of both. These policies were introduced in Subsection 2.1.3.4. Policies A and B are called *static* because they both save and restore the registers that have been defined by the compiler without taking into account the usage of these registers during program execution. In this chapter we propose six architectural policies that make use of *dynamic* information to reduce the RSR traffic, we compare the architectural support required by each dynamic policy, and we select one (**Policy G**) as the best candidate for implementation and comparison.

To evaluate the RSR traffic reduction obtained for each policy it is necessary to have a specific register allocation and assignment performed by the compiler. The alternative of presenting each policy with each possible register allocation scheme is discarded because it would be too repetitive and confusing. Thus, we have selected a register allocation scheme based on the register allocation performed by the Portable C Compiler (PCC) [John79, Lion79, Ritc79, Kess83]. This scheme has already been used to evaluate several register saving/restoring schemes by some other authors [Patt82b, Kate83, Eick87, Hsu87], although it is not optimal for an optimizing compiler, because the overall traffic can be reduced with a better selection of which local scalar variables should be allocated to registers (see Subsection 2.1.3 and Section A.1). However, since we are now just considering RSR traffic and not overall traffic, this scheme is still useful for evaluating the RSR traffic reduction caused by each policy because it implies a heavy register usage. Moreover, in Section 4.2 we will verify that the conclusions obtained hold for two others register allocation approaches (one used by the Amsterdam Compiler Kit [Tane83] and the second used by the GNU C Compiler [Stal88]).

The PCC standard intra-procedural register allocation only allocates to registers integer and pointer *register variables* [Kern78] explicitly defined by the programmer

Save registers	Restore saved registers		
	on return	on return if already used by caller	when first read by caller
<i>on call</i> if defined in caller	A {M}		
<i>on call</i> if defined in callee	B {M}		
<i>on call</i> if defined in callee and used in exterior levels	C {M, TBS}		
when used by callee if used in exterior levels	D {TBS, CU}		
<i>on call</i> if defined for callee and used in exterior levels and not yet saved		E {M, TBS, CU}	G {M, TBS}
when used by callee if used in exterior levels and not yet saved		F {TBS, CU}	H {TBS, CU}

- M register Mask given at the function entry point
(to indicate registers defined by the function)
- CU Current Usage mask
(to indicate registers used by the current function)
- TBS To-Be-Saved mask
(to indicate registers used by the exterior levels)

Table 3.1: Register Saving/Restoring Policies

(see Subsection 2.1.3.1). Consequently, our measurements show that, on the average, only 31% of the defined local scalar variables are allocated to registers and there are 1.94 registers assigned per executed function when there are 6 *to-be-preserved* (TBP) registers. This number increases only to 2.0 when the number of TBP registers is 32. In this case, the traffic for the local scalars which have not been allocated accounts for 36% of the total data memory traffic generated by the local scalar variables. To increase the register usage the register allocation policy has been modified so that local scalar variables are assigned to registers¹ while there are TBP registers available. The scalar variables considered for allocation not only include integer and pointer variables (as PCC does), but also characters and short integers. As we mentioned in Section 1.3, the four programs measured (ASM, NROFF, SORT, and VPCC) do not use scalar floating-point variables. In this case, the local data traffic has been reduced to zero when there are 32 TBP registers available.

Moreover, the PCC intra-procedural register assignment policy has also been modified. PCC always assigns registers in the same order for each function. Thus, some registers are assigned more frequently than others. Since the dynamic policies require that the intersection of registers defined by the caller and by the callee be as small as

¹Our previous measurements have showed that the alias problem can be ignored in performing this type of measurement since the number of variables with an alias is insignificant [Hugu85a].

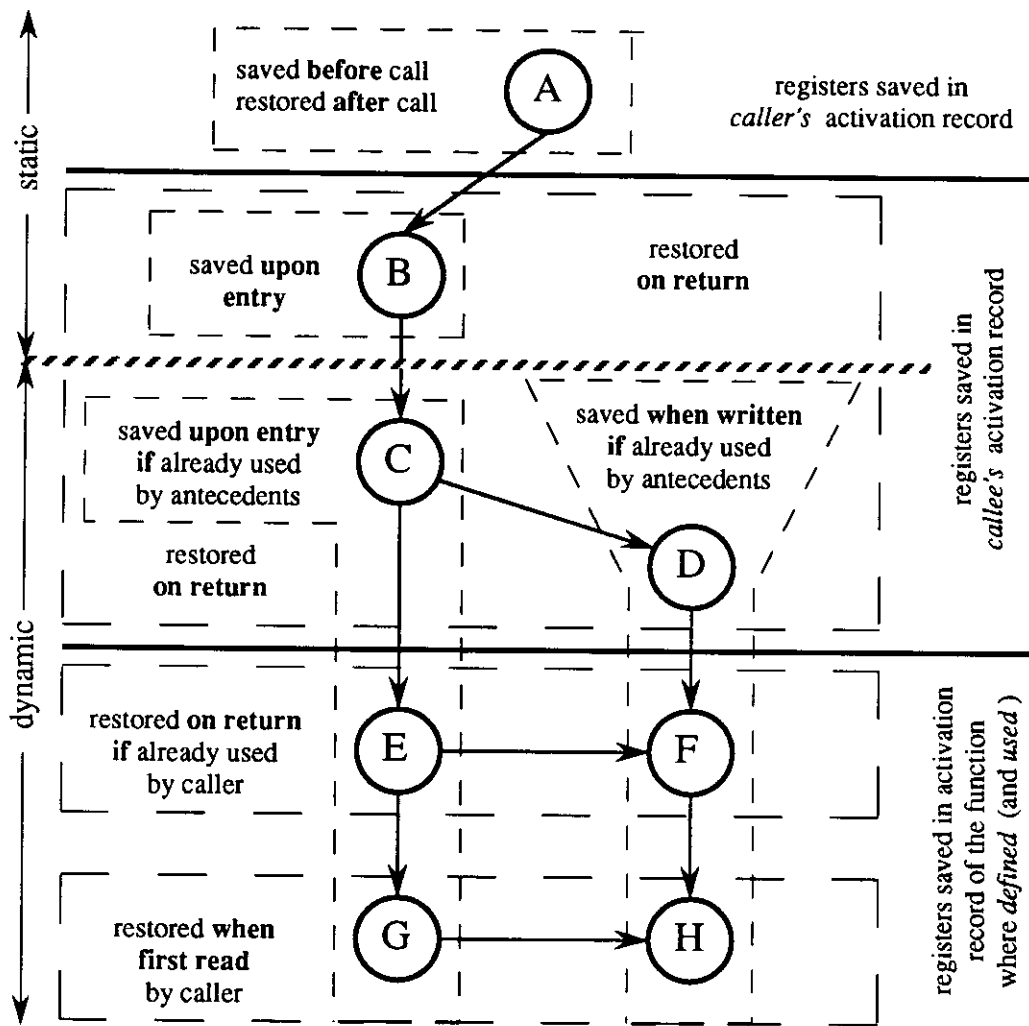


Figure 3.1: Relation among the Register Saving/Restoring Policies

possible, registers are assigned in a *round-robin fashion*. As a consequence, consecutively-defined functions will use different registers while these functions have less local scalar variables than the number of TBP registers available. This does not guarantee having disjoint registers for a caller and a callee, but it increases the probability for doing so. In Subsection 5.4.4, we show an inter-procedural optimization for a more efficient register assignment policy.

Table 3.1 shows the two static and the six dynamic policies to save/restore registers for single-window architectures. They are named from A through H. Since the objective is to reduce the RSR memory traffic, the approaches are presented in order from the more conventional ones to those that we expect will produce the least traffic. Register usage is defined by a *mask* so that each bit in the mask has associated with it one of the general-purpose registers to be preserved across function calls. Table 3.1 also shows the masks required for each RSR policy. The *static mask M* indicates the registers that have been

defined in the function. This mask is created by the compiler and included in the code section of the program. The *dynamic masks* **CU** and **TBS** indicate the dynamic usage of the registers during program execution. A detailed description of the way the masks are used in each policy is given in Sections 3.1 through 3.3.

Figure 3.1 shows a graph with the relation among policies. Each policy reduces the RSR traffic generated by its predecessor(s) with the addition of more architectural support (i.e., hardware). The figure is divided into three parts according to the storage location where the registers are saved by each policy:

1. Policy A saves the registers in the *caller's activation record* before performing the call (see Figure 3.2.a), thus the callee always receives a *clean* register set.
2. Policies B, C, and D save the registers in the *callee's activation record* (see Figure 3.2.b). In these cases, only the registers that are required by the callee need to be saved (how the registers are saved/restored for Policies C and D is discussed in Section 3.2). Registers are restored before the activation record is destroyed, i.e., before the return instruction, even though the caller might not need them.
3. Policies E, F, G, and H save the registers in the activation record of the function that has *defined* them, i.e., *used* them, so that they do not have to be restored until this function becomes active. Notice that this function is not necessarily the immediate caller (see Figure 3.2.c). The architectural support required to perform this saving/restoring and a description for each of these policies are given in Section 3.3.

Figure 3.1 also specifies where the registers are saved (before the call, at function entry, or when the register is first written) and restored (after the call, on return, or when the register is first read). This is also discussed in detail in Sections 3.1 through 3.3.

Table 3.2 shows the RSR traffic caused by each policy for six register-set configurations of TBP registers (6, 8, 12, 16, 24, and 32). The RSR traffic is normalized with respect to Policy B since this is the conventional policy used when no optimizations are performed (these will be discussed in next chapter). The column for Policy B shows the RSR traffic generated per function enclosed in parentheses. In addition to the RSR traffic generated by each program, an *average traffic* for the four programs is also given (labeled **4 P.**). This average has been computed as the sum of the total traffic for each program divided by the sum of the number of function calls in each program. In general, the measurements discussed in the text correspond to this average. Measurements for SPICE are shown in Section 4.3, once the intra-procedural optimizations have been presented.

To compare the different architectural policies we just consider the RSR traffic because the data traffic caused by global variables, local scalar variables not assigned to registers, local arrays and structures, etc., is identical for all the policies (for a given register-set configuration). Thus, the RSR traffic allows us to measure the data memory

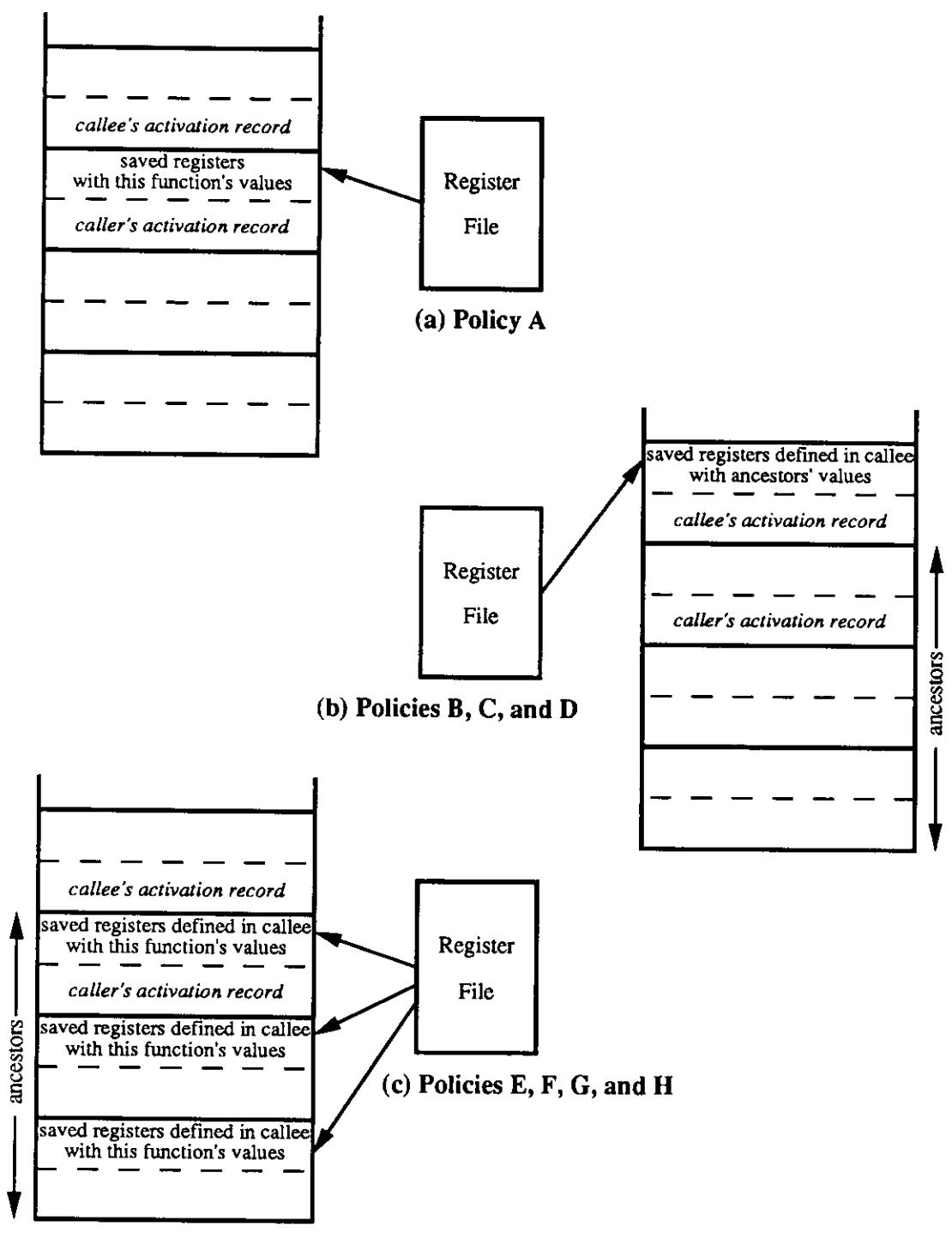


Figure 3.2: Activation Records where Registers Are Saved

no. regs.	program	Policies							
		A	B	C	D	E	F	G	H
6	ASM	1.20	1.0 (9.67)	0.75	0.68	0.75	0.56	0.23	0.21
	NROFF	1.36	1.0 (4.71)	0.64	0.71	0.50	0.41	0.19	0.17
	SORT	2.87	1.0 (4.18)	0.89	0.99	0.89	0.88	0.56	0.55
	VPCC	1.18	1.0 (7.60)	0.81	0.72	0.68	0.53	0.39	0.30
	4 P.	1.52	1.0 (5.44)	0.74	0.75	0.64	0.53	0.31	0.27
8	ASM	1.49	1.0 (10.11)	0.74	0.67	0.74	0.56	0.23	0.21
	NROFF	1.43	1.0 (4.73)	0.67	0.63	0.40	0.33	0.17	0.15
	SORT	3.20	1.0 (5.00)	0.91	0.98	0.91	0.90	0.47	0.47
	VPCC	1.19	1.0 (8.73)	0.78	0.67	0.62	0.46	0.36	0.25
	4 P.	1.66	1.0 (5.88)	0.75	0.71	0.58	0.48	0.29	0.24
12	ASM	2.05	1.0 (10.22)	0.72	0.64	0.68	0.50	0.20	0.18
	NROFF	1.44	1.0 (4.74)	0.61	0.51	0.37	0.28	0.12	0.10
	SORT	3.52	1.0 (6.60)	0.82	0.92	0.76	0.75	0.40	0.39
	VPCC	1.25	1.0 (8.91)	0.74	0.66	0.47	0.40	0.25	0.22
	4 P.	1.86	1.0 (6.25)	0.70	0.65	0.50	0.43	0.23	0.21
16	ASM	2.52	1.0 (10.22)	0.63	0.56	0.63	0.43	0.16	0.13
	NROFF	1.44	1.0 (4.74)	0.41	0.38	0.23	0.19	0.10	0.08
	SORT	3.62	1.0 (8.15)	0.64	0.78	0.47	0.41	0.18	0.14
	VPCC	1.29	1.0 (9.03)	0.76	0.58	0.38	0.28	0.22	0.16
	4 P.	2.01	1.0 (6.59)	0.58	0.55	0.36	0.29	0.16	0.12
24	ASM	3.31	1.0 (10.22)	0.62	0.55	0.58	0.39	0.14	0.12
	NROFF	1.44	1.0 (4.74)	0.39	0.30	0.13	0.12	0.06	0.06
	SORT	3.65	1.0 (8.93)	0.55	0.50	0.33	0.28	0.13	0.12
	VPCC	1.31	1.0 (9.10)	0.60	0.49	0.29	0.24	0.16	0.14
	4 P.	2.11	1.0 (6.76)	0.51	0.42	0.26	0.22	0.12	0.10
32	ASM	4.08	1.0 (10.22)	0.59	0.53	0.58	0.37	0.15	0.12
	NROFF	1.44	1.0 (4.74)	0.32	0.30	0.13	0.12	0.05	0.05
	SORT	3.65	1.0 (8.93)	0.66	0.57	0.50	0.41	0.21	0.18
	VPCC	1.31	1.0 (9.10)	0.64	0.46	0.29	0.21	0.17	0.12
	4 P.	2.16	1.0 (6.76)	0.52	0.43	0.31	0.24	0.14	0.11

Table 3.2: RSR Traffic Relative to Policy B

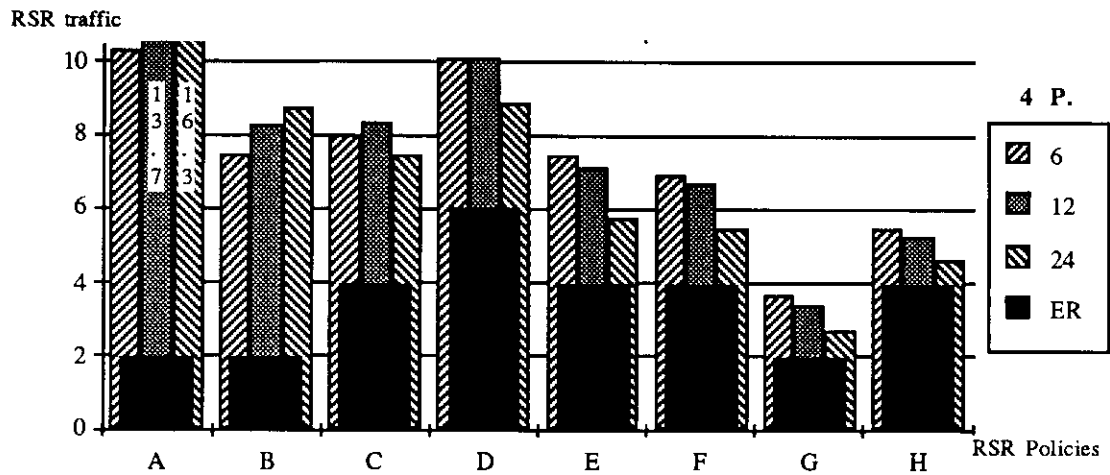


Figure 3.3: RSR Traffic per Function Call with Fixed Cost

reduction given by each policy. However, to select one of the dynamic policies for implementation we have also to consider the traffic caused by the *environment registers* (ER). In addition to the *return address* (i.e., the program counter) which must be saved and restored each time a function is called, some dynamic masks have to be save/restored depending on the policy. The masks that need to be saved/restored are boldfaced in Table 3.1. This is considered a *fixed cost* per function. No traffic is accounted for other environment registers (*frame pointer* and/or *argument pointer*) because we assume that the activation record size is fixed during function execution so that locals and parameters are referred to with respect to the *stack pointer* [Wulf75]. Figure 3.3 shows the RSR traffic for the average of the four programs including the traffic caused by the return address and the masks. The measurements that are usually mentioned in the text correspond only to the RSR traffic without the ER traffic because special compiler and architectural support is proposed to reduce the return-address traffic (see Subsection 4.1.3 and Section 5.3).

From Figure 3.3 we conclude that the dynamic Policy G is our best candidate for implementation since it is the one that generates the least RSR traffic. Policy G has between 12% (when 24 registers are available to the allocator) and 31% (for 6) of the RSR traffic generated by Policy B and between 5% (for 24) and 20% (for 6) of Policy A. In addition to reducing the RSR traffic, Policy G also reduces the number of registers to be saved/restored during context switching. We also show that when the register set size is increased, the static Policies A and B generate more RSR traffic while this is not the case for the dynamic Policy G.

Sections 3.1 through 3.4 present the architectural support required for each policy and the RSR traffic reduction given by each one. They also discuss the measurements given by Table 3.2 and Figure 3.3. Finally, Section 3.5 comments on the implementation of Policy G.

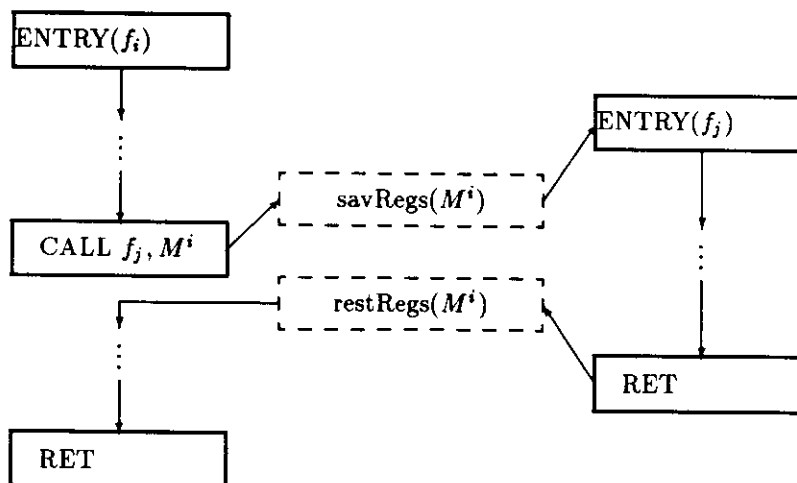


Figure 3.4: Policy A

3.1 Static Policies A and B

Policies A and B are the conventional RSR policies for intra-procedural allocators (see Subsection 2.1.3.4). **Policy A** saves all the registers defined in the caller when a function is called, and restores the saved registers when the function returns (see Figure 3.4). The registers to be saved/restored can be specified in three ways:

1. By explicit instructions. One instruction per register to be saved/restored. In this case, no mask is needed.
2. By an explicit instruction to perform the saving and by a second to perform the restoring. In this case, a mask in the instruction is used to specify the registers to be saved/restored.
3. By a mask given in the call instruction. In this case, the same mask (in the call instruction) can be used when the function returns. This is the case shown in Figure 3.4.

The most appropriate specification depends on instruction format provided by the machine (see Subsection 2.2.2). For our discussion we assume that registers are saved/restored by a mask M given in the call instruction for Policy A or at function entry and in the return instruction for Policy B (see below).

In **Policy B**, the registers defined in the callee are saved at function entry, and these saved registers are restored upon return (see Figure 3.5). The registers to be saved can be specified in a mask located at the function entry point. Since the return instruction must know which registers to restore, the same mask can be duplicated on each return instruction. The alternative approach of saving the register mask at function entry is discarded because it generates two extra memory references for each function call.

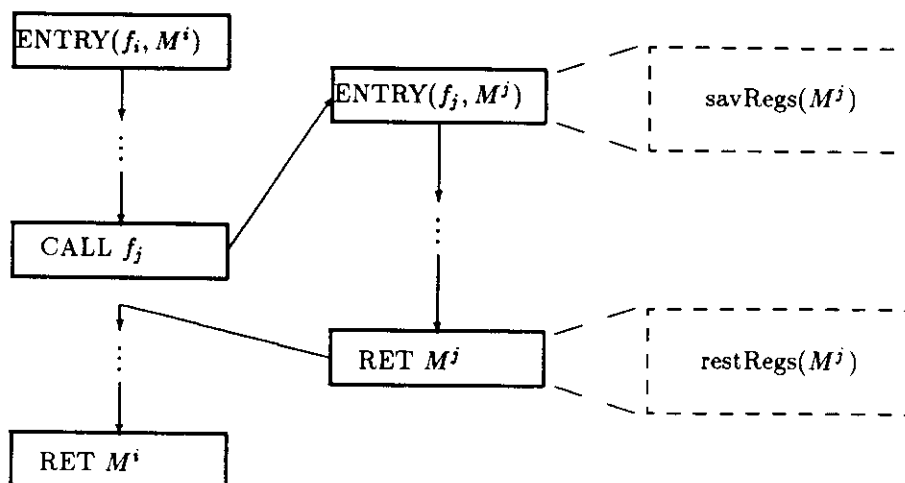


Figure 3.5: Policy B

As we can see in Table 3.2, Policy B is systematically better than Policy A because the register set is *cleaned* before each call for Policy A, while for Policy B some of the registers might be *dirty* upon entry and only the ones needed by the callee are forced to be cleaned. However, when live-variable analysis is performed, Policy A performs better than B for some programs (see Subsection 4.1.1).

Notice that for ASM and SORT, the register saving/restoring traffic for Policy A is about four times the one for Policy B when 32 registers are available, because both programs have a main function with a large number of variables. For instance, the parser for the assembler requires 32 registers. This function is only called once so that 32 registers have to be saved/restored for Policy B (assuming the largest register set). However, this function generates 11% of the calls and, therefore, Policy A has to save/restore 32 registers for each call.

It is also worth noticing that due to the nature of these saving/restoring policies there is an increase in RSR traffic when the number of registers is increased from 6 to 32. In contrast, we will see that there is some reduction in the dynamic policies.

3.2 Policies C and D

The large register saving/restoring traffic for Policies A and B is due to the fact that all registers defined for a function are saved, irrespective of their use, that is, a register is saved even if it has not been used previously. To reduce the RSR traffic in Policies C and D a register is saved only if it has been used in *exterior levels* (i.e., by one of the functions which is currently active). The management of these policies requires a dynamic mask ($TBS \equiv$ To-Be-Saved) which specifies the registers that have been used in those levels. This mask has a bit associated to each register; this bit is set when the

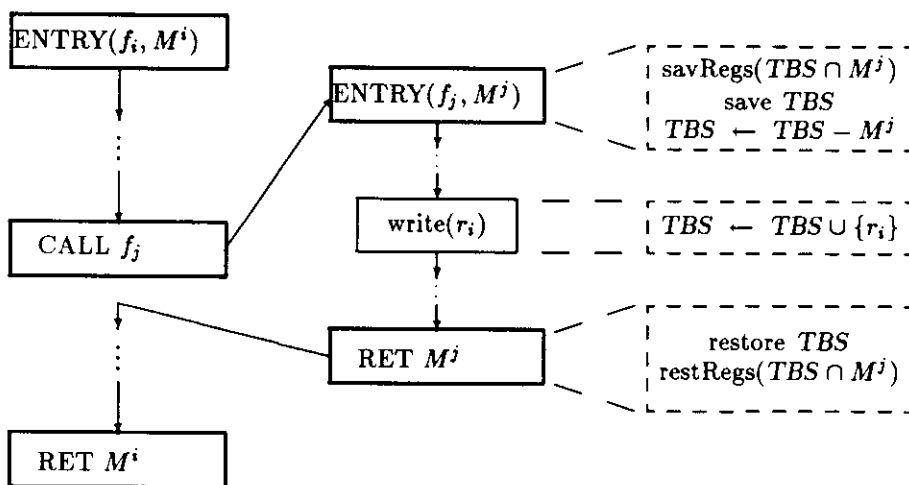


Figure 3.6: Policy C

Set Operations	Hardware Implementation
$TBS \cap M$	TBS AND M
$TBS \cup M$	TBS OR M
$TBS - M$	TBS AND NOT M
$TBS \leftarrow TBS \cup \{r_i\}$	SET TBS < i >
$TBS \leftarrow TBS - \{r_i\}$	RESET TBS < i >

Table 3.3: Hardware Implementation for Set Operations

register is written. The algorithms to describe the dynamic policies (like the one given in Figure 3.6 for Policy C) use set operations (union, intersection, difference) between the masks. Table 3.3 describes the correspondence between the set operations and their hardware implementation with AND, OR, and NOT primitives and SET and RESET operations on *SR* flip-flops for the masks.

In **Policy C** a register is saved during a call if it is defined for the callee and has been used by exterior levels. The registers saved are restored on return (see Figure 3.6). The architectural support for this policy consists of the dynamic mask *TBS* and one static mask *M* per function. During a call, the registers that correspond to the intersection of both masks are saved, *TBS* is also saved, and finally the bits corresponding to the saved registers are cleared. When a register is written the bit of the *TBS* mask is set. During return *TBS* is restored and the intersection of both masks determines which registers to restore. Note that this mask saving/restoring increases the data traffic (see Figure 3.3).

As we mentioned at the beginning of this chapter, this policy (and the other dynamic policies) would perform better if disjoint registers are assigned to the caller/callee functions. The smaller the intersection, the fewer registers that have to be saved/restored. For this reason, when the number of TBP registers increases, the RSR traffic may de-

crease although the number of locals assigned to registers also increases. Our measurements show a RSR traffic reduction of 14% when the number of TBP registers is increased from 6 to 24 (i.e., when almost all the local scalar variables have been allocated). This is a characteristic for all the dynamic policies, but not for the static ones (see Figure 3.3). Notice that there is a 2% RSR traffic increase from 24 registers to 32. This is a consequence of the round-robin assignment. We show that the inter-procedural optimization proposed in Subsection 5.4.4 prevents this to happen.

Table 3.2 shows the register saving/restoring traffic for Policy C with respect to Policy B. Policy C has between 70% and 74% of the RSR traffic generated by Policy B when there are between 6 and 12 TBP registers and between 52% and 58%, when there are between 16 and 32. Thus, a larger register set gives us a larger traffic reduction with respect Policy B. However, if the mask traffic is considered, then the implementation of Policy C cannot be justified for smaller register sets (12 TBP registers or fewer) because it generates more traffic than Policy B (between 102% and 108%). This is not the case for larger register sets. Policy C has between 91% (for 16 TBP registers) and 86% (for 32) of the RSR traffic and the mask traffic generated by Policy B.

Some authors have proposed to implement this RSR policy directly by the compiler without any architectural support [Stee80, Cohe88]. In this case, the compiler has to generate code to perform the operations indicated above (in Figure 3.6). One register is selected to be used as the *TBS* mask. Before each call, one instruction is generated to indicate the registers which are alive at the call (i.e., the registers which might have been written during program execution). The callee performs the intersection of both masks to determine the registers to be saved and restored. This implementation has an advantage with respect to when it is implemented by the architecture: dead registers are not saved. However, it has the following two drawbacks:

1. Live registers might have not been written for this specific execution path, but they will be saved. Remember that live registers are computed following all possible paths to the call.
2. It increases the instruction memory traffic because more operations have to be explicitly executed by the program.

In Subsection 4.1.4 we propose an optimization for Policy C so that dead registers are not saved. In this case, the data memory traffic generated by the dynamic Policy C with this optimization is less than the one produced by a direct implementation of Policy C by the compiler because of the first mentioned drawback. Since in Subsection 4.1.4 we conclude that the dynamic Policy C should not be implemented because the increase in instruction memory traffic does not compensate for the reduction of data memory traffic obtained, a direct implementation of Policy C by the compiler cannot be justified either.

In Policy C registers are saved at function entry, whether the register is going to be used or not. To reduce the traffic further, **Policy D** saves the registers when they are going to be written instead of saving them at function entry.

Two dynamic masks are required in this case: *TBS*, as before, and *CU* (Current Usage) to indicate the register usage in the current function. This last mask is required for the restoration. The architectural support required for this policy is used for manipulating both masks and detecting whether a register has to be saved when it is going to be written. During the call, both masks are saved, the *TBS* is ored with the *CU*, and the *CU* is set to zeroes. When a register is written both masks are checked: if the register has been used in an exterior level but not in the current function, then it is saved before being written. On return the intersection of both masks determines the registers to restore and both masks are restored (see Figure 3.7).

Although two dynamic masks have to be saved/restored, this does not imply that four memory references are always generated per function (as Figure 3.3 shows). Depending on the register-set configuration and the machine word size it might be possible to pack both masks in a single word (e.g., a 32-bit word and a 16-TBP-register set configuration). In this case, only two memory references would be generated. (We decided to show the most general case in Figure 3.3 to emphasize the fact that both masks have to be saved/restored.)

Since Policy D does not require a static mask at function entry, it is necessary to have some other way of differentiating between TBP registers and TBD registers. Two alternatives exist for this differentiation: the separation can be fixed, or it can be specified by a global mask. The second solution allows more flexibility because each compiler is allowed to have its own partition of the register set.

Table 3.2 shows the RSR traffic for Policy D. Policy D generates between 93% and 95% of the RSR traffic produced by Policy C when 8, 12, and 16 TBP registers are available, 82% when 24, and 83% when 32. The performance for 6 registers has been a surprise since Policy D gives an unexpected, slightly higher average RSR traffic than Policy C (0.7%). This is caused by the ASM and SORT programs. Policy D generates 110% of the RSR traffic produced by Policy C for each of these programs. Moreover, Policy D for SORT also generates more RSR traffic for 8 (8%), 12 (13%), and 16 registers (22%). An example which reflects how this situation might happen is given in Figure 3.8. Policy C would save register r_3 at the entry point of the second function (since it would have been used previously) and, therefore, no further saving/restoring would be required at the entry point of the third function (since the register would not have yet been used). However, Policy D would save/restore the register r_3 every time that the third function is called. This might happen for SORT and ASM.

SORT has only one function with 18 local scalar variables. However, this function accounts for 20% of the calls and is heavily used (i.e., it generates 70% of the local references and 78% of the calls). Thus, when there are from 6 to 16 registers available, this function always has the whole set of registers assigned. If the registers are saved at the entry point of this function, it is more likely that this 80% of the functions being called will find a smaller intersection with the registers previously used.

ASM also has two functions with a larger number of local scalar variables. One is the parser with 32 local scalar variables. The parser generates 11% of the calls and 9%

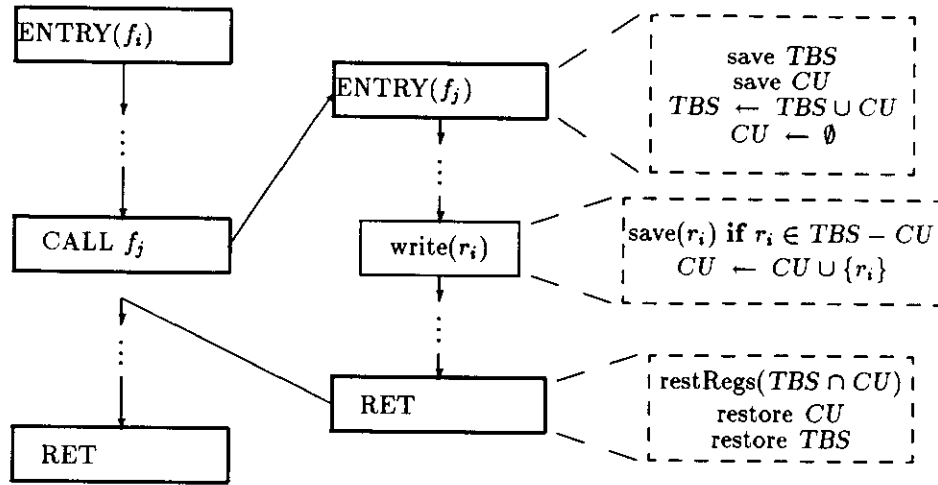


Figure 3.7: Policy D

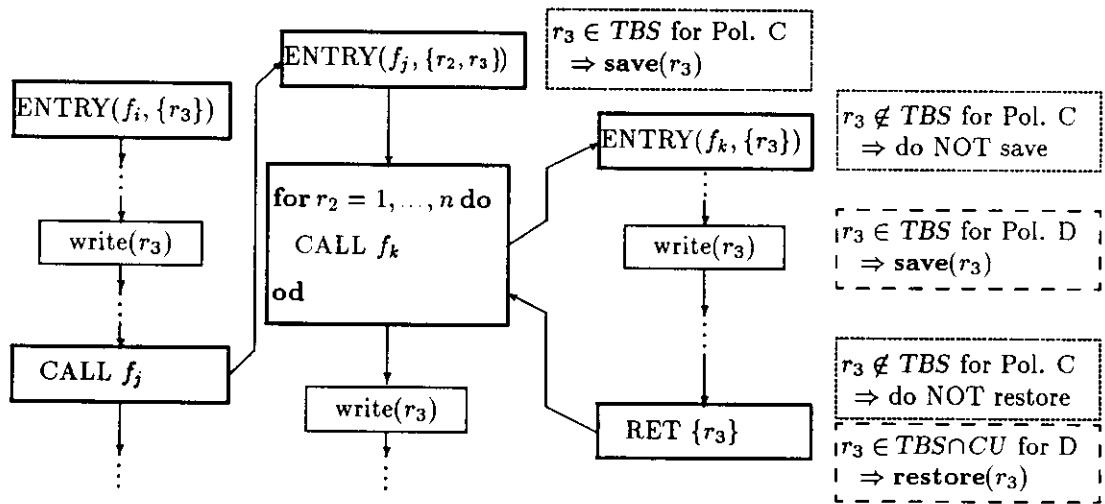


Figure 3.8: Policy D versus Policy C

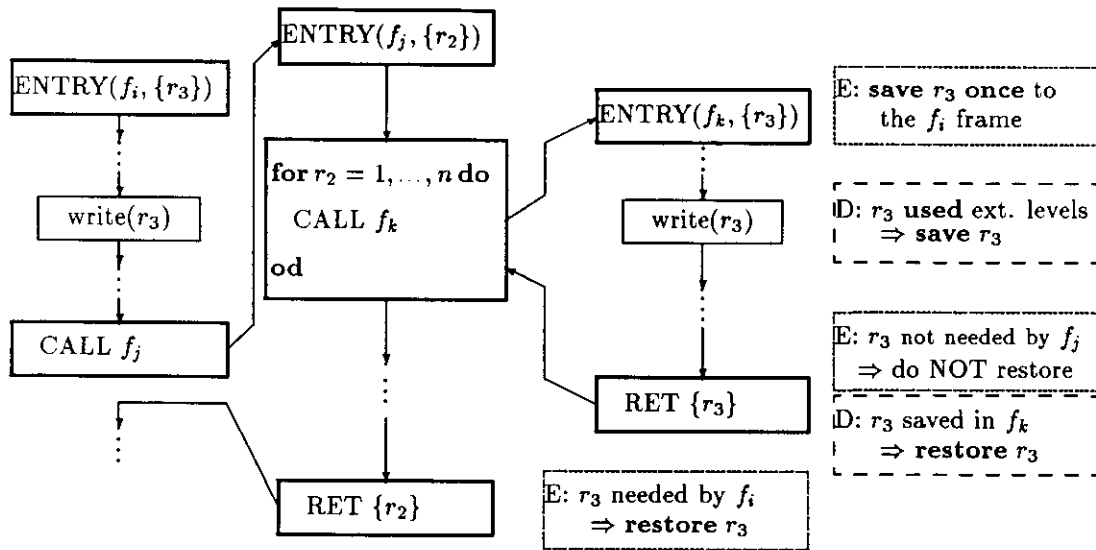


Figure 3.9: Policy E versus Policy D

of the local references. The other is one of the scanner functions with 18 local scalar variables. This one generates 49% of the calls and 24% of the local references. Thus, this could explain the abnormal behavior of Policy D with respect to Policy C when a small set of registers is available.

3.3 Policies E, F, G, and H

Better use of the registers can be made and the RSR traffic reduced by implementing other policies which reduce the unnecessary saving and restoring of Policies C and D. Consider for example the situation in Figure 3.9: in Policy D register r_3 is saved and restored every time that f_k is called from f_j , even if it is not used by f_j . It is clear that it is sufficient to save it once in f_k and restore it when returning to f_i . This is exactly what **Policy E** does. In this section we discuss the architectural support necessary to determine the address where the register has to be saved, we present the operations performed, and the masks required, by each policy, and we compare the RSR traffic generated.

The register is saved in the activation record of the outer function which last used it (f_i in the example). To do this, the architecture has associated with each register a *pointer* (RP) which is loaded with the current frame pointer (plus an offset associated to each register number) each time a register is written. This pointer is used to determine the memory location in which the register has to be saved. For instance, Figure 3.10 shows the values of the register pointers associated to registers r_1 through r_4 when the portion of the “program” given on the top of the figure has been executed. As you can see, rp_4 is pointing to the f_i frame since r_4 was last written in f_i , rp_3 and rp_2 to f_j ,

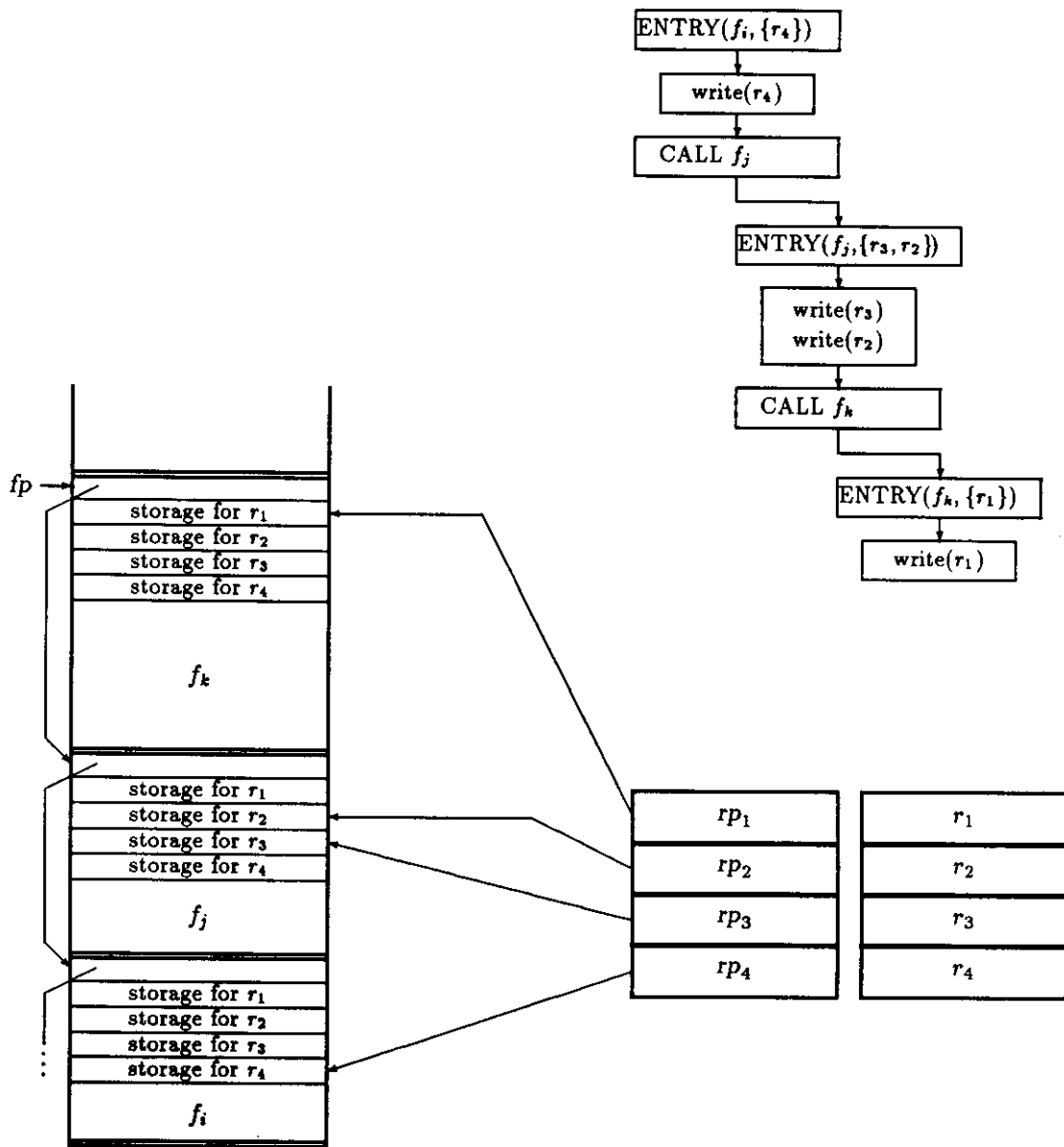


Figure 3.10: Register Pointers

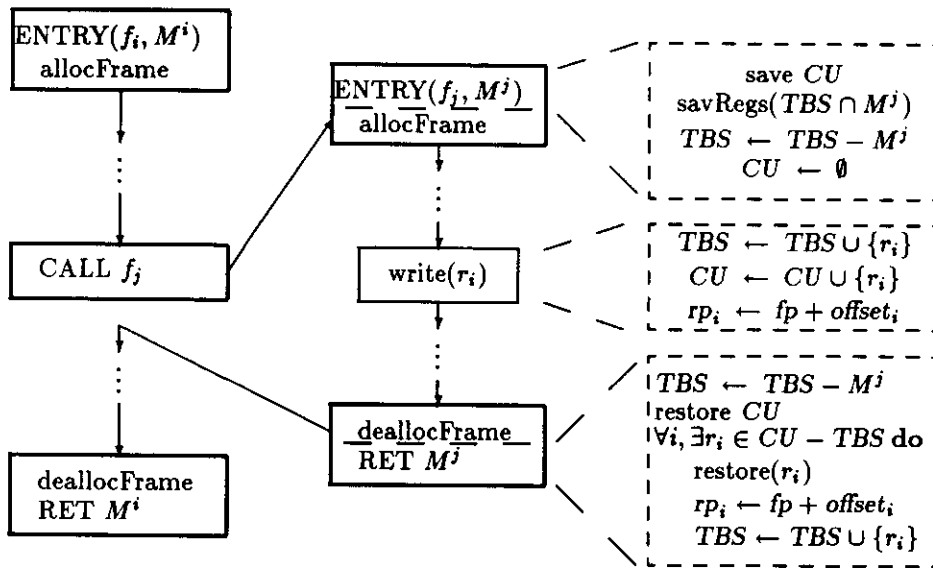


Figure 3.11: Policy E

and rp_1 to f_k . There is a drawback to this policy: storage space for each register has to be provided in each frame because the fp offsets where the register has to be saved are directly computed by the architecture.

Policy E requires the use of all three masks: the static M mask, the TBS mask, and the CU mask. When a register is written the corresponding bit for both the TBS and the CU masks are set to 1 (see Figure 3.11). A register is saved at function entry if its bit in both the TBS mask and the M mask for that function are 1 (as in Policy C). The TBS bits for the registers saved are cleared to indicate that the registers do not need to be saved (until they are written again). Also, the caller's CU mask is also saved to keep track of the registers that have been used.

To determine when a register is to be restored, the caller's CU mask is needed. When returning, the TBS bits associated to the saved registers are cleared and the caller's CU mask is restored. Then, if the CU mask has the bit set and the TBS mask indicates that it has been saved (i.e., it is zero), the register is restored (see Figure 3.11).

The RSR traffic produced by Policy E can be reduced still further by the following mechanisms:

1. Save a register when it is used (written) instead of at function entry (as in Policy D). This is **Policy F**. This policy uses two dynamic masks in a fashion similar to Policy D. However, only one mask, CU , needs to be saved/restored (see Figure 3.12).
2. Restore a register when it is needed (read) instead of at function return. This is done in **Policy G**. Note that in this case the restoring has to be done during

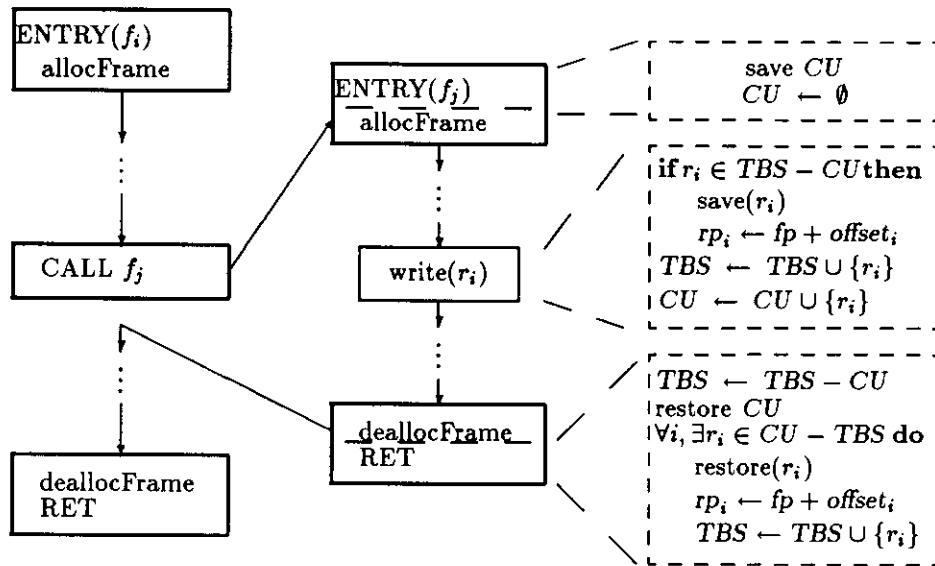


Figure 3.12: Policy F

execution of the instruction. This policy, like Policy C, uses the static mask and the dynamic TBS mask, but the TBS mask does not need to be saved/restored across function calls (see Figure 3.13).

- Combination of (1) and (2). This is **Policy H**. This policy uses both dynamic masks and CU must be saved/restored across function calls to know which registers have been used by the current function (see Figure 3.14). Observe that this information is given by the static mask for Policy G, which is available in both the call and the return instructions.

The minimum RSR traffic occurs with Policy H, because

- a register is not saved until the moment that it needs to be written and it was already used, and
- a register is not restored until the moment that it needs to be read.

This is the best that the architecture can provide. The restoring policy is optimal since it only restores registers whose value is needed during program execution. However, the saving policy is not optimal since *dead registers* (i.e., registers for which the first operation to be performed on them after the return is a write) are saved by this policy. A dead register is saved because the architecture does not know that its value can be destroyed since it will never be read by the function which used it. An optimizing compiler that performs live-variable analysis [Aho86, Hech77] knows this and, therefore, it could issue an instruction to clear the bits in the TBS mask associated to dead registers. In this

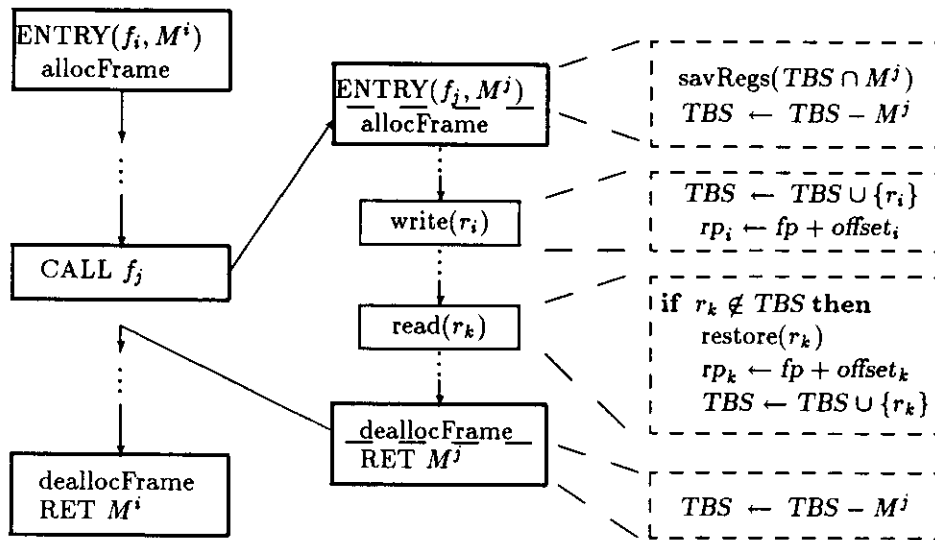


Figure 3.13: Policy G

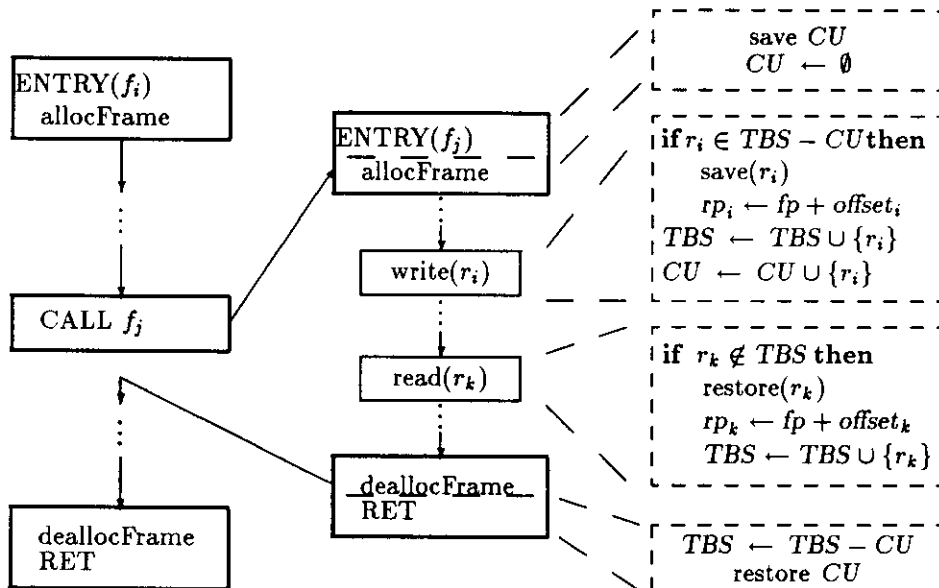


Figure 3.14: Policy H

case, the registers would not be saved. Notice that this can be done for each dynamic policy. This optimization is discussed in Subsection 4.1.4 for Policies C and G.

The saving/restoring traffic produced when using these policies is shown in Table 3.2. Policy H has between 21% and 27% of the RSR traffic generated by Policy B for 6, 8, and 12 registers, 10% for 16, and 12% for 24. As we noted above, the RSR traffic reduction becomes more significant for larger register sets. However, Policy H has the drawback that one mask has to be saved and restored, which increases the overall traffic (see Figure 3.3). Because of this, we select Policy G as the best candidate for implementation (discussed in Section 3.5).

Since Policy G is the dynamic policy to be considered for implementation, it is interesting to compare the RSR operations to be performed with the ones performed by one of the conventional static policies. We have selected Policy B because it generates less RSR traffic than Policy A when no live-variable analysis is performed (see Section 3.1).

Policy G as well as Policy B perform the register saving/restoring at the callee side (see Figure 3.1). However, while Policy B saves the whole set of registers defined in the function, Policy G only saves the registers that have been used (i.e., written) by the functions in the exterior levels. Once a register is saved, it is marked *unused* so that it is not saved again until a write operation is performed on it. Moreover, the register is not saved in the current function's activation record (as happens in Policy B), but in the activation record of the function that was using it (see Figure 3.10). In this manner, the register does not have to be restored when the function returns (as in Policy B), but when the function that was using it needs it (i.e., reads it). No explicit instruction is required for the restoring since it is performed implicitly when the register is read and is not present (see Figure 3.13). Notice that the register is not restored if the first operation to be performed after return is a write. Policy G generates from 12% of the RSR traffic produced by Policy B (for 24 TBP registers) to 31% (for 6).

In addition to reducing the RSR traffic, the dynamic Policy G also reduces the number of registers to be saved/restored during context switching. While multiple-window architectures and single-window architectures with the conventional static policies have to save/restore the whole register file, an architecture with the dynamic Policy G has only to save the TBP registers that are currently used (i.e., they have been written) and the whole set of TBD registers. The whole set of TBD registers has to be saved (and restored when the process is rescheduled to run) because they are not part of the *TBS* mask² (since we do not want to save them across function calls). The TBP registers needed by the process (i.e., registers that were alive when the process was switched) are restored dynamically when a machine instruction reads them.

The drawback of Policy G with respect to H is that it saves registers that are defined

²To avoid saving/restoring the TBD registers, a mask can also be provided. The corresponding bit of a TBD register is set to one when the register is written. This bit should be reset when the content of the register becomes dead. This could be performed by an instruction before a system call (which might cause a context switch) or by providing a bit per source register in the instruction format to indicate when the mask has to be reset. Since context switching is an unfrequent operation and the number of TBD registers is usually small, these alternatives cannot be justified.

in the function although they might not be used during the current execution of the function. These registers will be restored afterwards even though the content of the register is the same as the value being restored. However, this situation can be detected because the address in the register pointer is equal to the address being generated for the restoring³ (since no write has been performed in the register). In this case, the restoring could be aborted. Since the average number of restoring operations per function is small (e.g., 0.8 when 12 TBP registers are available) we decided to ignore this situation in the implementation of Policy G and to perform the restoring anyhow.

Policy G generates between 108% (for 12 registers) and 130% (for 16) of the RSR traffic produced by Policy H. However, when the ER traffic is also considered, Policy G generates between 32% and 50% of the traffic produced by Policy B and between 59% and 83% of Policy H. Therefore, Policy G is the best dynamic policy because no mask has to be saved/restored.

3.4 An Example for the Eight RSR Policies

Figure 3.15 illustrates the eight policies described. On the left, the “program” being executed is given. It indicates the function calls and returns and the operations performed on registers. For each policy we indicate the registers that are saved (*S*) and restored (*R*).

3.5 On the Implementation of Policy G

It could be the case that although the total number of data memory references becomes smaller, the total time to execute the program is larger because either the processor cycle time has increased as a consequence of the extra hardware required for Policy G or more processor cycles are required to execute the operations associated to Policy G. In this section we consider the implementation for our dynamic policy in a RISC-like processor. RISC II has been chosen because it is well documented [Kate83, Chapter 4]. A few modifications have been made in order to implement Policy G in RISC II; for this reason, we call it a RISC-like processor. These modifications are mentioned below. We show that the operations required by Policy G are performed in parallel with the main CPU activities in our RISC-like processor and that the number of cycles required to perform the register saving/restoring operations are the same as the ones required by the conventional static policies.

While RISC II saves and restores each register individually, i.e., with store and load instructions, this cannot be done in our RISC-like processor because the registers to be saved are given by the intersection of the mask provided by the instruction with the

³A mask cannot be used to detect this situation and to prevent the restoring because it would need to be saved/restored across function calls.

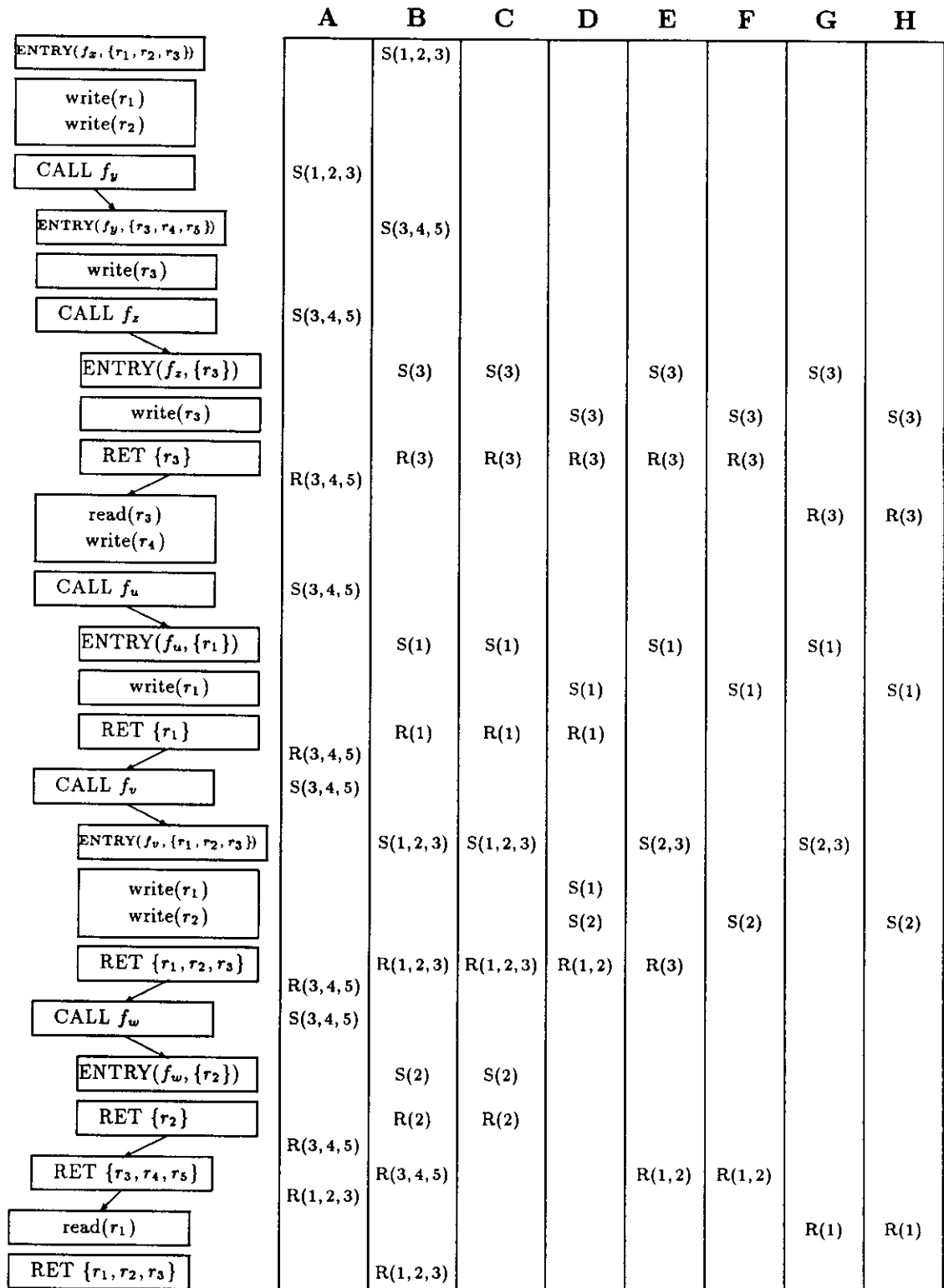


Figure 3.15: Example of RSR Traffic Caused by the Different Policies

TBS mask (see Figure 3.13). Thus, for our RISC-like processor, the registers to be saved are indicated by a mask M given by the instruction `savRegs` generated by the compiler at function entry. This mask is duplicated in the return instruction to avoid its saving/restoring (as proposed in Section 3.1).

When a single instruction is used to save a set of registers, the data memory transfers can be pipelined. For instance, SOAR saves/restores an 8-register window in 9 cycles rather than the typical 16 cycles it would take using individual `load` and `store` instructions [Unga84, p. 191]. However, Ungar *et al.* claim that:

In retrospect, these multi-cycle instructions added considerable complexity to the design, and the benefits may not prove to be worth the costs.

This could be the reason why none of the single-window processors listed in Table 2.2 provides such instructions. Thus, we assume that no pipelining occurs in our RISC-like processor for the data memory transfers (i.e., each data memory transfer takes 2 processor cycles).

Similar to RISC II, instructions have 3-register addresses and the processor is implemented with a 3-stage pipeline and with a 4-phase clock: in the first stage the instruction is fetched from memory; in the second, the operands are read from the register file and operated; and in the third, the result is written back to the register file. If necessary, internal forwarding is performed for the last computed result. The register file read is performed in the first clock phase and the register write in the third.

Our RISC-like processor has the *frame pointer*⁴ (*fp*) implemented as a specialized register rather than being part of the general-purpose register file as in RISC II. In this case, instructions to add/subtract a constant with the activation record size are needed as well as to read/write the frame pointer from/to memory. This is necessary for the implementation of Policy G.

Figure 3.16 shows the data section of the processor. A few RISC II data-path components are not shown in the figure although they should also be available in our RISC-like processor:

- Input and output latches for the ALU.
- Multiple PCs to implement the 3-stage pipeline (LASTPC and NEXTPC).
- The 32-bit cross-bar shifter.
- Several other RISC II registers (DIMM, IMM, OP, etc.).

Although they have been eliminated to simplify the figure, they do not represent any loss of generality for our discussion.

The added components required for the register saving/restoring policy have been put in a dashed box. These are:

⁴Also used as the argument pointer and the stack pointer. See the discussion on this at the beginning of this chapter.

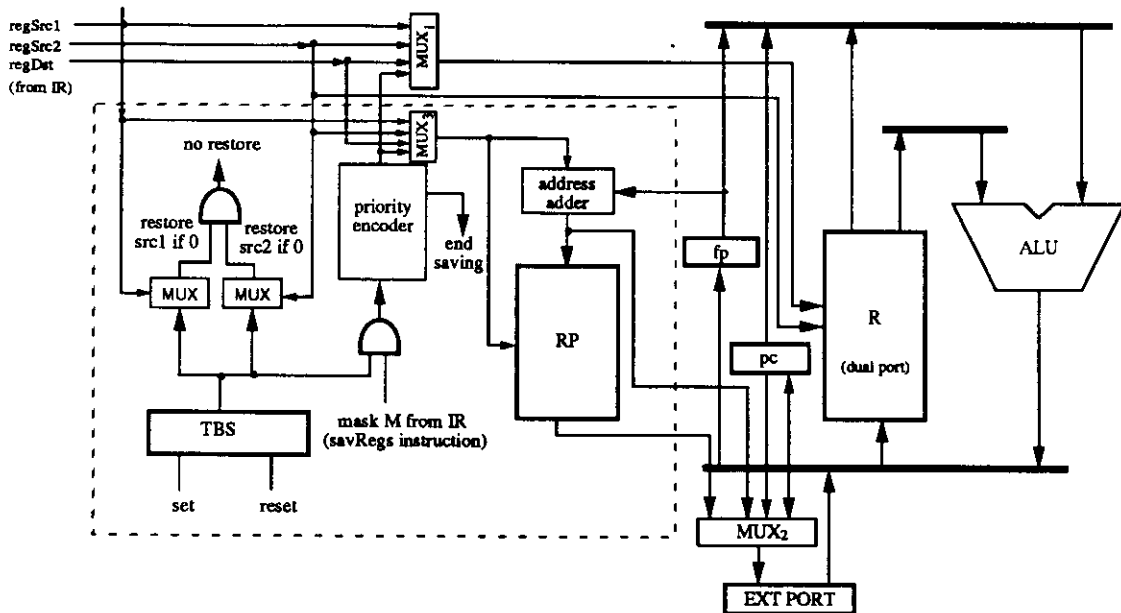


Figure 3.16: Implementation of Policy G

1. A set of registers RP to contain the pointers. One pointer is required for each TBP register. For this reason, the size of the RP register file is smaller than the general-purpose register file since the latter also includes the TBD registers.
2. A register to store the TBS mask.
3. An adder to compute the address of the memory location where the particular register is saved in (or restored from) the activation record. This address is obtained as the sum of the frame pointer (stored in the specialized register fp) and the register number (shifted according to the memory-word size). We call this adder the *address adder*.
4. One multiplexer (MUX_3) to select the register pointer to be accessed. The number of the register pointer (shifted according to the memory-word size) is the one added to the frame pointer.
5. A priority encoder of the AND of both masks to determine the next register to save. This should already exist to implement the instruction `savRegs` in the RISC-like processor.
6. Logic to determine whether to restore 0, 1, or 2 registers. This consists of two multiplexers and one AND gate.

Moreover, notice that the sizes of two of the existing multiplexers (MUX_1 and MUX_2) have been increased: one to select the register number to be used to address one of the ports in the general-purpose register file (R) and the other to select the output to the external port.

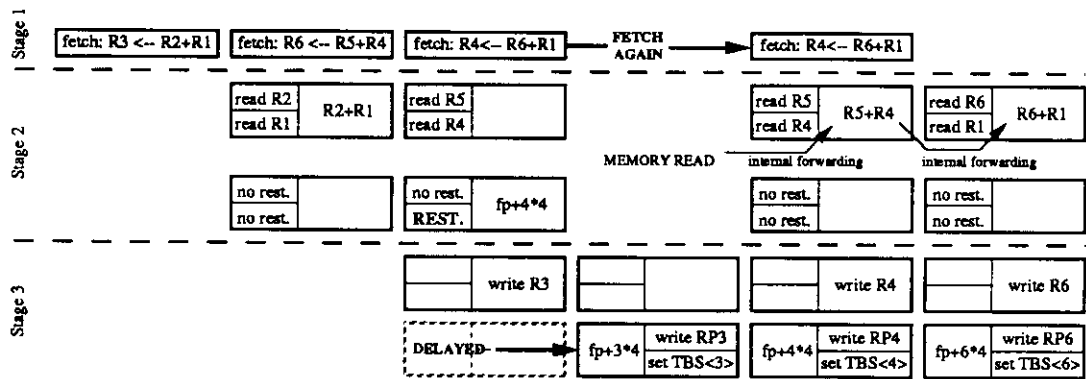


Figure 3.18: RISC-like Pipeline with Restoring

not written until the third stage, a hazard condition arises when the destination register of the current instruction is used as one of the source registers for the next one. However, the control unit already has to detect this situation to perform forwarding. Thus, the “false” restoring signal (because the corresponding bit has not been written yet) can also be detected and ignored.

If $no_restore = 0$, it is necessary to restore one or two registers. This restoration is done as normal register loads from memory, using as addresses the frame pointer plus the corresponding register numbers. This address is computed during the processor cycle of the aborted instruction as illustrated in Figure 3.18. Notice that since the address adder is required to compute this address, the RP write from the previous instruction has to be delayed by one processor cycle. After the register or registers are restored, the instruction is executed in the normal way. Since the restoring is performed in two processor cycles (as a conventional load instruction would do), no extra cycles are added.

Figure 3.19 highlights the critical path for a restoring operation: once the restoring operation is detected, the instruction should be aborted (not shown in the figure), the address of the TBP register to be restored computed, and this address sent to the external port. Notice that the address is not written to the RP register file until the restoring is performed (i.e., the general-purpose register is written; see Figure 3.18).

For the instruction `savRegs`, the priority encoder generates, in sequence, the register numbers of those that have to be saved. For each of them, the corresponding pointer is transferred to the external port followed by the register value. A memory write is performed and the corresponding bit of the TBS cleared. The clearing of this bit makes the priority encoder produce the next register number. The prefetched instruction is not loaded in the IR until the *end saving* condition is reached. Only one gate has been introduced in the data-path and we expect that this will not be critical because the restoring address has to be available at the beginning of the second cycle (see Figure 3.20).

For a context switch the registers are saved in the corresponding activation records so that the registers RP never have to be saved/restored. When the context has to be restored, the TBS is set to zero and no registers are restored. When the register is needed

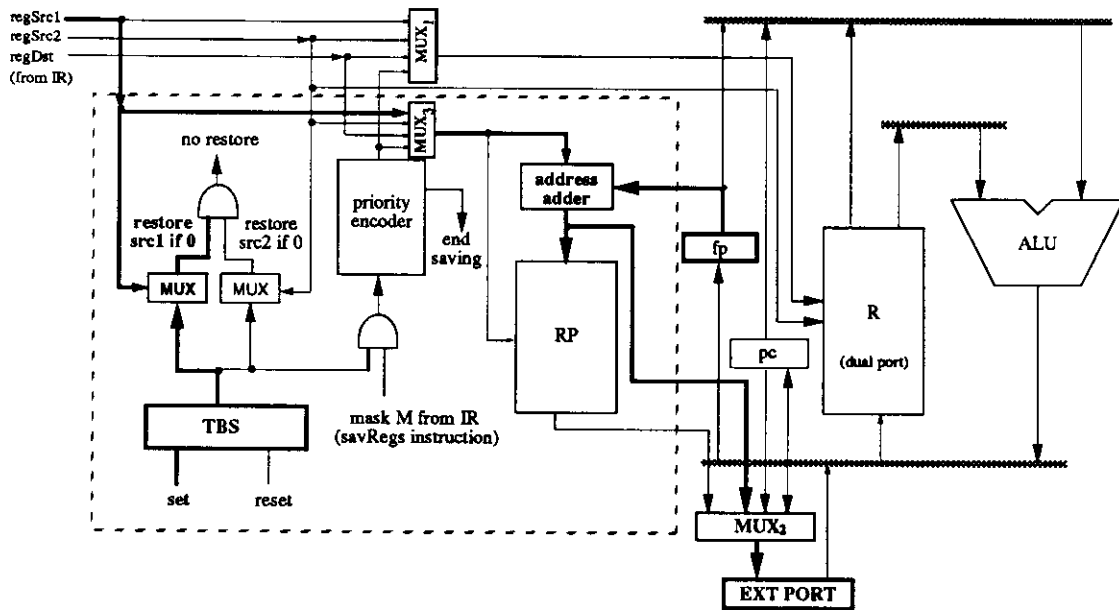


Figure 3.19: Critical Path for a Restoring Operation

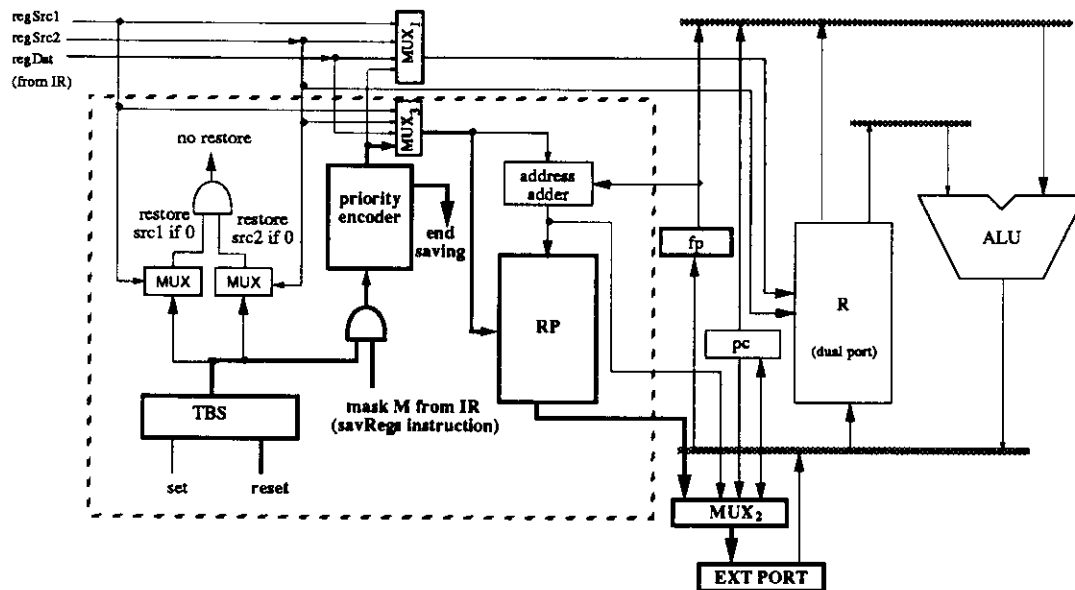


Figure 3.20: Critical Path for a Saving Operation

(i.e., it is read), then the register is loaded and its associated RP register is initialized.

The implementation described indicates that the additional hardware required for this saving/restoring policy consists mainly of a register set for the pointers and of an adder. The cost, in additional modules in a LSI implementation or circuit area in a VLSI implementation, is justified by the reduction in RSR traffic provided by the policy. With a gate-delay analysis it seems that the basic processor cycle is not increased by these additions because they are decoupled from the main datapaths and the operations which have to be done during normal register reads and writes overlap with the normal cycle when they are performed. However, in a VLSI implementation of a specific processor it is possible that the cycle time is affected because of:

1. The increased area which might imply longer buses and, therefore, larger communications delays [Scha89].
2. The increased size of the multiplexors MUX_1 and MUX_2 , which might imply a slight increase in the critical path if these multiplexors are part of the critical path.
3. The increased fan-out of several drivers (fp , $regSrc1$, $regSrc2$, and $regDst$) which have to be directed to the specific modules required by Policy G.

The effect of these factors on the processor cycle time depends on many design requirements and constraints for a given implementation. Although an implementation was performed, the effect of Policy G in this implementation could not be generalized to other alternative implementations—even for the the same architecture. It might be that the added circuitry does not affect the critical path for this specific processor implementation. On the other hand, if it does, the (expected small) increase in the processor cycle time is justified by the reduction of the off-chip references that will increase the overall system performance. Since alternative implementations are possible, the implementation of Policy G to determine this effect on the processor cycle time for a specific implementation has been left open for discussion (see Section 6.2). However, we expect that the information provided above will encourage the reader to consider the inclusion of Policy G in his/her design.

3.6 Summary

When no compiler optimizations are performed to reduce the RSR traffic, our measurements show that:

1. Policy G reduces the register saving and restoring traffic with respect to the conventional static policies used by most processors. Policy G has between 12% (for 24 TBP registers) and 31% (for 6) of the RSR traffic generated by Policy B and between 5% (for 24) and 20% (for 6) of Policy A.

2. As expected, an increase in the number of registers to be preserved across function calls implies an increase in the RSR traffic for the conventional Policies A and B. However, this is not true for the dynamic Policy G. When there are 32 TBP registers, the RSR traffic generated for Policy G is 54% of the one generated when there are 6. On the other hand, in the same situation, the RSR traffic generated for the Policies A and B is 176% and 124%, respectively. Consequently, the overall data memory traffic is not reduced for a larger register set when the conventional static RSR policies are used because the reduction in local scalar traffic is compensated by the increase of the RSR traffic (see Section 4.5).

Our measurements have also shown that when registers are saved/restored at the callee (Policy B), less RSR traffic is generated than when they are saved/restored at the caller. Policy A generates between 152% (for 6 registers) and 216% (for 32) of the RSR traffic produced by Policy B. As we will see in the next chapter, this is not the case when live-variable analysis is performed.

Finally, the implementation of Policy G has been considered. We have shown that the activities required by Policy G are performed in parallel with the main CPU activities (with the exception of one additional gate to generate the register numbers for the `savRegs` instruction) and that the number of cycles required to perform the register saving/restoring operations are the same as the ones required by the conventional static policies. Although the feasibility of the implementation of Policy G depends on many design requirements and constraints, we have not found any major drawback for which its implementation should be discarded. Therefore, we encourage the designer to consider its inclusion in his/her design.

Chapter 4

RSR Intra-Procedural Optimizations

Chapter 4 presents four intra-procedural optimizations to reduce the register saving and restoring (RSR) traffic for single-window architectures. These optimizations can be performed together with the eight RSR policies described in Chapter 3. Our goal is to compare the RSR traffic reduction offered by these optimizations without architectural support (i.e., for the static policies) and with architectural support (i.e., for the dynamic ones). From these eight policies, we only consider four policies for optimization: A, B, C, and G. The static Policies A and B are the conventional ones (see Subsection 2.1.3.4) and several optimizations have already been implemented (discussed below). The dynamic Policy G is our best candidate for implementation because it generates the least memory traffic (see Section 3.3). The dynamic Policy C has also been selected because it requires less architectural support than Policy G (although it has to save/restore a control mask on every function call) and some authors have proposed implementing this RSR policy directly by the compiler without any architectural support at all [Stee80, Cohe88].

For each of these policies we explain how the compiler can help to reduce the RSR traffic and we discuss the resultant reduction in RSR traffic. These measurements are obtained with the same register allocation and assignment approach used to measure the RSR traffic reduction by the architectural policies given in Chapter 3. That is, all local scalar variables are allocated to registers (if enough registers are available) and registers are assigned in a round-robin fashion. Afterwards, we also compare the RSR traffic reduction for other register allocation schemes (Section 4.2). Although two of the optimizations presented are not original (discussed below), we expect that our work will offer a systematic way of measuring the traffic reduction obtained by each optimization. We were not able to find in the current literature this type of evaluation for typical programs. The results available are usually either static measurements or dynamic measurements of small programs which might not be representative of typical programs (see Section 1.3).

Two existing intra-procedural optimizations are discussed and evaluated: live-vari-

able analysis [Hech77, Aho86] for Policy A (renamed **A-live**) and register assignment in leaf functions for Policy B (renamed **B-lf**), Policy C (renamed **C-lf**), and Policy G (renamed **G-lf**). Moreover, two new optimizations—based on live-variable analysis—are also presented. One is for Policy A-live (renamed **A-lvOpt**) and the other for Policies C and G (renamed **C-live** and **G-live**), although the latter could be used for any other dynamic policy.

There exists a third optimization to reduce the register saving/restoring traffic: *inline expansion* (also called *procedure integration*) [Sche77, Alle80, MacL84, Stee87, Hwu89]. This optimization substitutes the call to a function by the function itself. In this case, no RSR traffic is generated since the call itself is eliminated. This optimization is not considered in this dissertation because the performance gain obtained by this optimization cannot be given by the reduction in RSR traffic (as we have done for the architectural policies in Chapter 3 and we do for the other intra-procedural optimizations in this chapter) because several other factors have to be taken into account; for example, the reduction in the number of functions defined, the increase in the number of variables defined per function after the expansion, the variation in the hit/miss ratio and in the page-fault ratio due to the expansion, etc. Moreover, more RSR traffic might be generated for the function that has been expanded since more registers might be required due to the expansion. Constant propagation [Wegm85, Call86] can be used to reduce the number of variables required after the expansion and to reduce the code size generated (because some computations can be performed at compile time).

We show that our dynamic Policy G with leaf-function optimization (Policy G-lf) generates the least RSR traffic with respect to the optimized static policies for any of the three compilers measured (PCC [John79], ACK [Tane83], and GNU [Stal88]). Moreover, the same conclusions have also been reached when, instead of using general-purpose applications (like ASM, NROFF, SORT, and VPCC), we use a numerical application (SPICE) with a heavy floating-point variable usage. (These programs have been briefly described in Section 1.3; see Section 4.3 for an explanation on the differences between these programs.)

Our measurements have also shown that for the static policies, there is not a best approach for performing the register saving and restoring—either at the caller (Policies A, A-live, and A-lvOpt) or at the callee (Policies B and B-lf). Although on the average Policy B-lf performs better than any of the policies of type A, this is only true for some of the programs measured and for some register-set configurations (see Sections 4.1, 4.2, and 4.3).

We compare the RSR traffic generated by our best optimized dynamic Policy G-lf to the one generated by the already-existing schemes for multiple-window architectures: fixed-size windows [Kate83], variable-size windows [Ditz82], and multi-size windows [Hugu85c] (Section 4.4). Since our goal is to have a small register file (32 general-purpose registers) to avoid an increase in the processor cycle time (as discussed in Subsection 2.1.1), we show that a single-window register file with Policy G-lf generates less RSR traffic than a multiple-window register file with multi-size windows, which is the best scheme for a multiple-window register file.

Finally, we measure the overall data memory traffic reduction obtained with the optimized policies and estimate the speed-up factor for a RISC-like processor with Policy G-lf in comparison to the static policies (Sections 4.5 and 4.6). We show that when intra-procedural register allocation and assignment are performed, our measurements do not indicate any reason to have more than 12 to-be-preserved (TBP) registers in the general-purpose register set for non-numeric applications (i.e., ASM, NROFF, SORT, and VPCC). For SPICE, a larger number of TBP registers may be used as we will discuss in Section 4.5.

4.1 Compiler Support to Reduce the RSR Traffic

This section discusses four different optimizations performed at function level. The compiler generates code per function without any knowledge of the register usage required by the functions which might be called or the functions which might call the current one. Two of the optimizations (live-variable analysis and live-variable analysis optimized) are performed on Policy A and discussed in Subsections 4.1.1 and 4.1.2. Leaf-function optimization is applied on the static Policy B and on the dynamic Policies C and G (Subsection 4.1.3). Finally, an optimization based on live-variable analysis is presented for the dynamic Policies C and G (Subsection 4.1.4).

As we said above, the compiler used to evaluate these optimizations is the same used to perform the evaluations for the architectural policies. The compiler has not been modified to incorporate these optimizations since our goal is not to obtain a production compiler, but to measure the performance of the different compiler optimizations. These have been measured using BKGGEN (see Section 1.2) which has turned out to be a flexible tool to let us measure not only several architectural configurations, but also these intra-procedural optimizations. For instance, to measure the leaf-function optimization, the actions file (which indicates the events to measure per block) is modified as follows: the masks that indicate the read and write operations performed on the TBP registers (per block) are cleared if these registers can be transferred to the to-be-destroyed (TBD) registers (for the whole leaf function). Thus, the same actions file generated to measure the RSR traffic without leaf-function optimizations can be used without having to recompile the programs. Also, the same BKGGEN function available to measure the architectural policies can also be used. (A detailed discussion on how BKGGEN has been used to measure each intra-procedural optimization is outside of the scope of this dissertation.)

4.1.1 Policy A-live

This is the standard intra-procedural register allocation for Policy A implemented by an optimizing compiler when variables are assigned to registers for the whole scope of the function [Aho86]. While the unoptimized Policy A saves and restores all the registers defined by the function, Policy A-live only saves and restores a subset of them: the ones

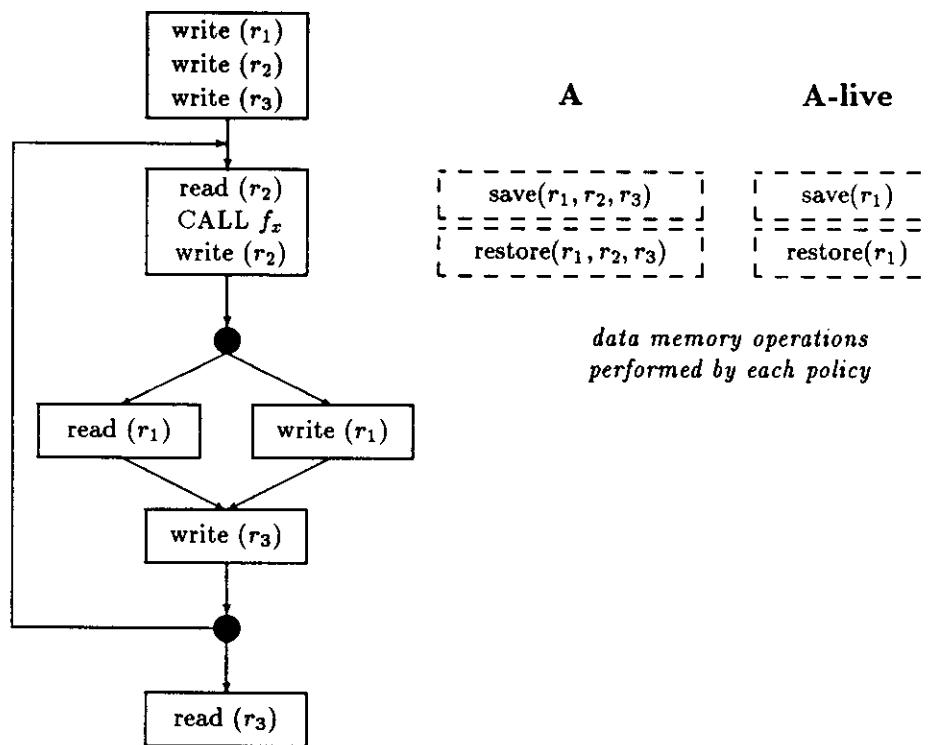


Figure 4.1: Policy A-live versus Policy A

which are needed (i.e., read) after the call (once the function has returned). Registers that are written before being read (in all the possible paths after the call) do not need to be saved/restored. These registers are called *dead* or *killed*. In contrast, registers that are first read in any of the possible paths after the call are called *live registers*. An optimizing compiler performs live-variable analysis to find the live/dead information and only saves/restores the live registers (see Figure 4.1).

Table 4.1 shows the RSR traffic for Policy A-live with respect to the standard Policy B (as given in Table 3.2). Policy A-live always generates less RSR traffic than the unoptimized Policy A. Policy A-live has from 38% of the RSR traffic generated by Policy A when there are 32 TBP registers available to 53% when there are 6. We deduce that the RSR traffic reduction is larger for a larger register set because the number of live variables at the call does not increase at the same rate as the number of variables allocated to registers per function. This is a consequence of having all possible local scalar variables allocated to registers.

Although Policy A-live generates, on the average, less RSR traffic than Policy B, this is not true for every measured program. Policy A-live always generates less RSR traffic than Policy B for NROFF and VPCC because almost all the executed functions in these programs require a small number of registers. For instance, 99% of the functions executed in NROFF require 6 TBP registers and 94% of the functions executed in VPCC require 8. As a consequence, the RSR traffic generated by Policy A-live is about half of

no. regs.	program	Policies			Policy B	
		A	A-live	A-lvOpt		
6	ASM	1.20	0.63	0.33	1.0	(9.67)
	NROFF	1.36	0.44	0.24	1.0	(4.71)
	SORT	2.87	2.54	1.63	1.0	(4.18)
	VPCC	1.18	0.50	0.40	1.0	(7.60)
	4 P.	1.52	0.80	0.51	1.0	(5.44)
8	ASM	1.49	0.71	0.39	1.0	(10.11)
	NROFF	1.43	0.50	0.28	1.0	(4.73)
	SORT	3.20	2.13	1.37	1.0	(5.00)
	VPCC	1.19	0.49	0.37	1.0	(8.73)
	4 P.	1.66	0.79	0.50	1.0	(5.88)
12	ASM	2.05	0.80	0.45	1.0	(10.22)
	NROFF	1.44	0.50	0.28	1.0	(4.74)
	SORT	3.52	1.61	1.04	1.0	(6.60)
	VPCC	1.25	0.49	0.37	1.0	(8.91)
	4 P.	1.86	0.75	0.48	1.0	(6.25)
16	ASM	2.52	1.10	0.51	1.0	(10.22)
	NROFF	1.44	0.50	0.28	1.0	(4.74)
	SORT	3.62	1.31	0.85	1.0	(8.15)
	VPCC	1.29	0.48	0.36	1.0	(9.03)
	4 P.	2.01	0.74	0.46	1.0	(6.59)
24	ASM	3.31	1.14	0.53	1.0	(10.22)
	NROFF	1.44	0.50	0.28	1.0	(4.74)
	SORT	3.65	1.55	0.91	1.0	(8.93)
	VPCC	1.31	0.48	0.36	1.0	(9.10)
	4 P.	2.11	0.81	0.48	1.0	(6.76)
32	ASM	4.08	1.33	0.57	1.0	(10.22)
	NROFF	1.44	0.50	0.28	1.0	(4.74)
	SORT	3.65	1.55	0.91	1.0	(8.93)
	VPCC	1.31	0.48	0.36	1.0	(9.10)
	4 P.	2.16	0.83	0.49	1.0	(6.76)

Table 4.1: RSR Traffic for Policy A Optimizations Relative to Policy B

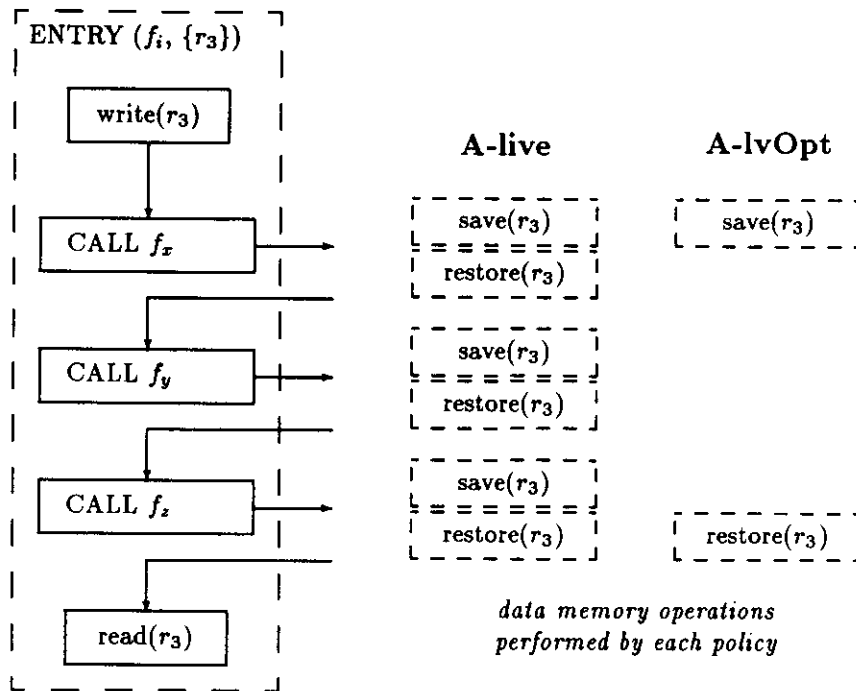


Figure 4.2: Policy A-lvOpt versus Policy A-live

the one generated by Policy B for any of the register-set configurations.

On the other hand, Policy B always generates less RSR traffic than Policy A-live for SORT. As we mentioned in Section 3.2, SORT has a function which requires 18 TBP registers. This function accounts for 20% of the calls and generates 70% of the calls. Since these 70% calls are to functions with only one TBP register, Policy A-live generates more RSR traffic than Policy B because, on the average, there are more live registers in a call for this 18-defined-register function than registers defined for the other ones (only 1).

Policy B for ASM also generates less RSR traffic than Policy A-live for larger register sets (16 TBP registers or more). It does not behave exactly like SORT because the functions being called have a larger number of registers defined so that Policy B itself generates a significant amount of RSR traffic (the largest among the four programs). Thus, only for larger register sets is the average number of live registers in a call greater than the average number of registers defined per function.

4.1.2 Policy A-live Optimized (A-lvOpt)

Policy A-live generates some unnecessary RSR traffic. For instance, let us consider the situation shown in Figure 4.2. The function currently being parsed has three *consecutive calls*, that is, three calls without unconditional or conditional branches between them.

program	static	dynamic
ASM	13.9%	1.3%
NROFF	21.1%	5.5%
SORT	18.9%	19.3%
VPCC	23.3%	15.9%
4 P.	20.5%	10.4%

Table 4.2: Percentage of Consecutive Calls

Register r_3 is written before the first call and it is read after the third one. Since it is alive at each call, Policy A-live saves/restores it in every consecutive call. However, it is only necessary to save it at the first call and to restore it upon return from the third one. This is done by Policy A-live Optimized (renamed **A-lvOpt**). This optimization can be applied not only to the consecutive calls, but also to the non-consecutive calls obtained following all possible paths, as we will discuss next.

Policy A-lvOpt saves only the registers with live variables that have been written since the last time that they were saved (following all possible paths to the specific call instruction in the function currently being processed) and restores only the ones that will be read after the function returns (again following all possible paths until the next call or return instructions). Thus, Policy A-lvOpt eliminates the unnecessary RSR traffic of saving a register whose content has already been saved and of restoring a register that will not be read by the instructions that might be executed before the next call.

This optimization is easier to implement for consecutive calls. In this case, a *peephole optimizer* could do it without having to collect control-flow information to follow all possible paths. However, the number of consecutive calls is small: 21% of the calls are consecutive and those account for 10% of the executed calls (see Table 4.2). For this reason, we prefer to implement this optimization following all possible paths.

While in Policies A and A-live, registers can be saved at the top of the stack (from where they are restored after return), Policy A-lvOpt requires that registers always be saved/restored at/from a compiler-fixed displacement in the frame (for a given function). This is necessary to restore a register saved by a previous call. Since the *activation record* or *frame* of an active function is fixed during all its lifetime,¹ this additional requirement does not increase the compiler complexity.

Table 4.1 shows the RSR traffic for Policy A-lvOpt with respect to the standard Policy B. On the average, Policy A-lvOpt generates between 59% and 64% of the RSR traffic produced by the standard Policy A-live and between 49% and 51% of Policy B. Notice that for ASM Policy A-lvOpt always generates less RSR traffic than Policy B and that for SORT it generates less RSR traffic when at least 16 TBP registers are available (in contrast with Policy A-live).

Figure 4.3 shows the RSR traffic for Policies A, B, A-live, and A-lvOpt. Policy

¹This is necessary to use the stack pointer as frame pointer and argument pointer (see Subsection 2.1.3.5).

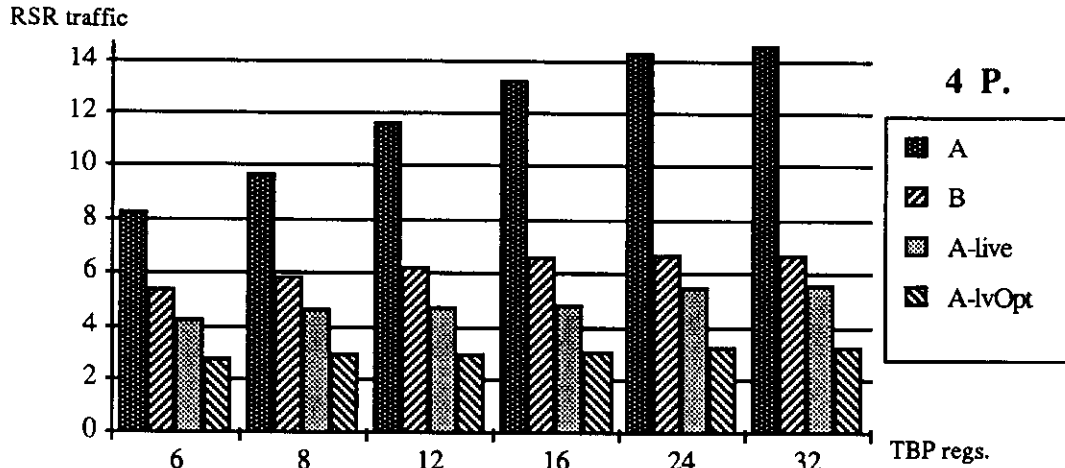


Figure 4.3: RSR Traffic for Policies A-live and A-lvOpt

A-lvOpt is not only the static policy that generates the least RSR traffic, but also the one whose RSR traffic generated grows slower when the register set size is increased. When there are 32 TBP registers, the RSR traffic generated is 118% of the one generated when there are 6. On the other hand, for Policies A, A-live, and B, the RSR traffic for 32 registers is 176%, 129%, and 124% of the one for 6, respectively.

4.1.3 Policies B, C, and G with Leaf Functions

An optimizing compiler can reduce the register saving/restoring traffic during function calls by changing the assignment of variables to registers for *leaf functions*. A function is said to be a leaf when it does not have any call instruction in its definition body. Since these functions are not going to generate any other call, the variables selected for allocation are assigned to registers that can be destroyed across functions rather to registers that must be preserved. In this case, if enough TBD registers are available, no RSR traffic is generated. Let us discuss first how the traffic generated by the environment registers (ER) can be reduced by this intra-procedural optimization and, afterwards, how the traffic produced by the TBP registers can be reduced.

ER Traffic Reduction

The ER traffic for Policies A, B, and G is caused by the *return address* (RA) traffic and for Policy C by the RA traffic and the *TBS* (To Be Saved; see Section 3.2) mask traffic. To reduce the RA traffic the architecture can provide call and return instructions that save/restore the return address in a register (rather than to the stack). This register is called the *link* register. When the callee is not a leaf function, the return address (i.e., the link register) is saved to the stack (to free the register for the calls that this function might perform). Notice that during execution it might happen that no call is generated.

program	static	dynamic
ASM	30.7%	19.4%
NROFF	24.7%	15.4%
SORT	45.6%	77.9%
VPCC	22.6%	18.1%
4 P.	26.7%	28.8%

Table 4.3: Percentage of Leaf Functions

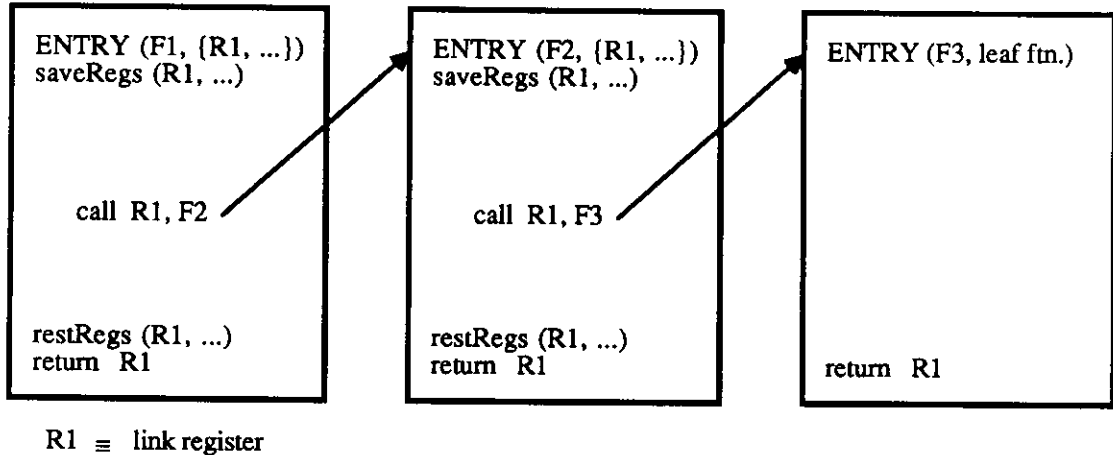


Figure 4.4: Leaf-Function Optimization for the Return Address

However, even in this case, the link register is saved anyhow because Policies B, C, and G save the registers at function entry.

On the other hand, when the callee is a leaf function, the return address can remain in the register and no RA traffic is generated (see Figure 4.4). This optimization can also be performed for Policies A-live and A-lvOpt if the register with the return address is saved at the function entry and restored before return for non-leaf functions, i.e., this register is saved/restored with a Policy B-lf approach rather than with the respective A-live or A-lvOpt. As we can see in Table 4.3, on the average, 29% of the functions do not need to save/restore the return address when it is stored in a register by the call instruction. This optimization can also be applied to the TBS mask for Policy C.

TBP Registers Traffic Reduction

Table 4.4 shows the RSR traffic generated by Policies B, C, and G with leaf-function optimization relative to the standard Policy B. Policy C includes the saving/restoring traffic of the TBS mask for every function (2 data memory accesses per function call) while Policy C with leaf-function optimization includes it only for non-leaf functions. Leaf measurements have been taken for 2 different numbers of TBD registers, 4 and 6. The suffixes -lf4 and -lf6 have been added to each policy to denote the number of TBD

no. regs.	program	Policies			Policies			Policies		
		B	B-lf4	B-lf6	C	C-lf4	C-lf6	G	G-lf4	G-lf6
6	ASM	1.0 (9.67)	0.89	0.85	0.95	0.80	0.77	0.23	0.19	0.19
	NROFF	1.0 (4.71)	0.86	0.84	1.07	0.92	0.91	0.19	0.17	0.17
	SORT	1.0 (4.18)	0.63	0.63	1.37	0.72	0.72	0.56	0.28	0.28
	VPCC	1.0 (7.60)	0.92	0.88	1.07	0.94	0.92	0.39	0.34	0.33
	4 P.	1.0 (5.44)	0.84	0.82	1.11	0.89	0.87	0.31	0.24	0.24
8	ASM	1.0 (10.11)	0.89	0.86	0.94	0.80	0.76	0.23	0.20	0.19
	NROFF	1.0 (4.73)	0.86	0.84	1.09	0.97	0.96	0.17	0.16	0.16
	SORT	1.0 (5.00)	0.69	0.69	1.31	0.69	0.69	0.47	0.28	0.28
	VPCC	1.0 (8.73)	0.93	0.90	1.01	0.92	0.90	0.36	0.32	0.32
	4 P.	1.0 (5.88)	0.86	0.84	1.09	0.89	0.88	0.29	0.24	0.23
12	ASM	1.0 (10.22)	0.89	0.86	0.91	0.78	0.75	0.20	0.19	0.19
	NROFF	1.0 (4.74)	0.86	0.84	1.03	0.92	0.91	0.12	0.11	0.11
	SORT	1.0 (6.60)	0.76	0.76	1.12	0.71	0.71	0.40	0.22	0.22
	VPCC	1.0 (8.91)	0.93	0.90	0.96	0.87	0.84	0.25	0.22	0.20
	4 P.	1.0 (6.25)	0.86	0.84	1.02	0.85	0.83	0.23	0.17	0.17
16	ASM	1.0 (10.22)	0.89	0.86	0.83	0.75	0.75	0.16	0.15	0.15
	NROFF	1.0 (4.74)	0.86	0.84	0.83	0.71	0.70	0.10	0.09	0.08
	SORT	1.0 (8.15)	0.81	0.81	0.89	0.69	0.69	0.18	0.18	0.18
	VPCC	1.0 (9.03)	0.93	0.90	0.98	0.90	0.88	0.22	0.22	0.22
	4 P.	1.0 (6.59)	0.87	0.85	0.89	0.77	0.76	0.16	0.15	0.15
24	ASM	1.0 (10.22)	0.89	0.86	0.82	0.76	0.72	0.14	0.13	0.13
	NROFF	1.0 (4.74)	0.86	0.84	0.81	0.71	0.71	0.06	0.06	0.06
	SORT	1.0 (8.93)	0.83	0.83	0.78	0.52	0.52	0.13	0.11	0.11
	VPCC	1.0 (9.10)	0.93	0.90	0.82	0.76	0.76	0.16	0.16	0.16
	4 P.	1.0 (6.76)	0.87	0.86	0.81	0.68	0.67	0.12	0.11	0.11
32	ASM	1.0 (10.22)	0.89	0.86	0.79	0.72	0.72	0.15	0.14	0.14
	NROFF	1.0 (4.74)	0.86	0.84	0.74	0.63	0.62	0.05	0.05	0.04
	SORT	1.0 (8.93)	0.83	0.83	0.89	0.63	0.63	0.21	0.16	0.16
	VPCC	1.0 (9.10)	0.93	0.90	0.86	0.79	0.79	0.17	0.17	0.17
	4 P.	1.0 (6.76)	0.87	0.86	0.82	0.68	0.68	0.14	0.12	0.12

Table 4.4: RSR Traffic for Leaf-Function Optimization Relative to Policy B

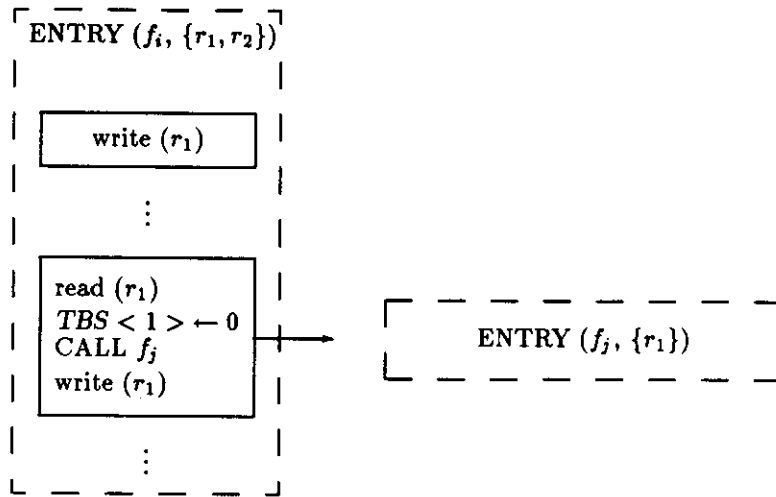


Figure 4.7: Example for Policy G-live

sets (see Figure 4.6). Policy B-lf has 94% (97%) of the traffic generated by Policy C-lf when there are 6 (8) TBP registers available. However, for a larger register set Policy C-lf performs better: on the average, it has from 99% of the Policy B-lf traffic when 12 TBP registers are available to 78% when there are 24 and 32. Notice that Policy C-lf generates less traffic than Policy B-lf for every single measured program when there are at least 16 TBP registers.

As we have already discussed in Chapter 3, less RSR traffic is generated for the dynamic Policy G when more TBP registers are available, but not for the static Policy B. This is still true for the optimized RSR traffic. Leaf-function optimization benefits both policies so that the RSR traffic reduction for G with respect to B (69%–86%) is similar to the one for G-lf with respect to B-lf (71%–88%).

4.1.4 Policies C-live and G-live with Leaf Functions

The dynamic Policies C and G save the registers defined in the function if they have been used in the exterior levels. It is possible that, although the register has already been used, it is a dead register (as defined in Subsection 4.1.1). Thus, its content can be destroyed and it is not necessary to save/restore the register. Notice that Policy C saves and restores a dead register, but Policy G only saves it (it does not restore it because it will never be read).

The callee does not know which registers are alive for the caller because code is generated per function (as we discussed in Subsection 2.1.3; see Figures 2.5 and 5.2). Thus, the compiler has to generate an extra instruction before the call to clean the bits of the *TBS* mask associated with the dead registers. For example, in Figure 4.7 the register r_1 used in f_i is dead when this function calls f_j . To avoid the saving operation (since the register was previously written), the bit associated to this register in the *TBS*

no. regs.	program	Policy C					Policy G				
		-lf	-lv1	-lv2	-lv3	-lv4	-lf	-lv1	-lv2	-lv3	-lv4
6	ASM	4.29	2.86	3.07	3.63	3.76	(1.81)	0.60	0.69	0.73	0.76
	NROFF	5.47	3.41	3.67	4.90	5.08	(0.79)	0.51	0.60	0.88	0.96
	SORT	2.54	2.41	2.41	2.47	2.47	(1.18)	0.95	0.95	0.96	0.96
	VPCC	2.74	1.56	1.68	2.24	2.45	(2.62)	0.61	0.65	0.86	0.91
	4 P.	3.68	2.39	2.54	3.22	3.38	(1.31)	0.64	0.69	0.88	0.92
8	ASM	4.07	2.61	2.63	2.81	3.12	(1.98)	0.55	0.56	0.63	0.74
	NROFF	5.92	3.50	4.09	5.17	5.27	(0.78)	0.59	0.68	0.91	0.96
	SORT	2.46	2.04	2.28	2.28	2.28	(1.40)	0.82	0.94	0.94	0.94
	VPCC	2.84	1.66	1.79	2.22	2.30	(2.83)	0.65	0.69	0.86	0.89
	4 P.	3.76	2.34	2.62	3.15	3.23	(1.40)	0.66	0.73	0.88	0.91
12	ASM	4.09	2.37	2.40	2.53	2.74	(1.95)	0.47	0.49	0.51	0.61
	NROFF	8.42	5.33	5.72	7.96	8.06	(0.52)	0.56	0.66	0.93	0.98
	SORT	3.17	1.96	2.01	2.01	2.68	(1.47)	0.78	0.78	0.78	0.94
	VPCC	3.88	2.10	2.40	2.88	2.99	(1.99)	0.63	0.66	0.83	0.86
	4 P.	4.86	2.90	3.13	3.90	4.17	(1.09)	0.64	0.68	0.82	0.90
16	ASM	4.99	3.10	3.11	3.24	3.26	(1.54)	0.57	0.57	0.58	0.59
	NROFF	8.00	5.19	5.59	6.93	7.02	(0.42)	0.49	0.61	0.94	0.98
	SORT	3.86	2.21	2.47	2.47	3.14	(1.47)	0.78	0.78	0.78	0.94
	VPCC	4.13	2.07	2.37	3.10	3.22	(1.97)	0.69	0.72	0.83	0.86
	4 P.	4.96	2.87	3.16	3.78	4.04	(1.02)	0.66	0.70	0.83	0.90
24	ASM	5.59	3.32	3.32	3.46	3.46	(1.38)	0.45	0.45	0.45	0.45
	NROFF	11.76	8.04	9.39	11.20	11.36	(0.28)	0.55	0.68	0.90	0.98
	SORT	4.82	2.47	2.88	2.95	2.95	(0.96)	0.92	0.92	0.92	0.92
	VPCC	4.87	2.52	2.95	3.61	3.78	(1.43)	0.62	0.66	0.79	0.83
	4 P.	6.39	3.75	4.34	5.04	5.15	(0.72)	0.67	0.72	0.82	0.85
32	ASM	5.07	3.25	3.26	3.31	3.33	(1.46)	0.59	0.60	0.60	0.61
	NROFF	12.36	8.18	8.64	9.98	10.12	(0.24)	0.60	0.71	0.90	0.97
	SORT	3.82	2.65	2.67	2.92	3.59	(1.47)	0.78	0.78	0.78	0.94
	VPCC	4.57	2.30	2.64	3.21	3.30	(1.56)	0.66	0.70	0.80	0.83
	4 P.	5.56	3.42	3.64	4.17	4.47	(0.83)	0.69	0.72	0.79	0.88

Table 4.5: RSR Traffic for Policies C-live and G-live Relative to Policy G-lf

no. regs.	program	minimum number of bits to be cleared in the <i>TBS</i>							
		1 bit		2 bits		3 bits		4 bits	
		def.	exe.	def.	exe.	def.	exe.	def.	exe.
6	ASM	43.6	68.3	27.2	47.9	15.2	15.1	8.6	12.9
	NROFF	45.3	69.2	22.9	55.7	7.2	23.0	3.4	1.7
	SORT	46.3	22.2	21.3	22.1	7.9	1.1	4.9	1.1
	VPCC	58.4	70.6	31.7	55.5	19.7	16.7	10.9	7.8
	4 P.	51.4	60.0	28.1	48.6	15.1	16.9	8.2	3.4
8	ASM	59.2	93.3	34.7	65.5	20.9	54.0	14.1	21.4
	NROFF	46.4	69.3	23.5	55.7	8.9	23.5	3.7	1.7
	SORT	52.4	41.6	26.2	22.1	12.8	22.1	7.3	22.1
	VPCC	59.1	72.2	33.0	56.6	21.3	17.5	14.8	12.0
	4 P.	55.6	65.3	30.7	49.5	17.7	23.2	11.5	8.9
12	ASM	61.9	93.6	36.9	66.0	25.0	59.5	17.3	51.9
	NROFF	46.5	69.3	23.5	55.7	9.0	23.5	4.1	1.7
	SORT	54.9	41.6	28.7	40.0	16.5	40.0	9.1	22.1
	VPCC	59.5	72.2	33.6	56.6	21.6	17.5	15.5	12.2
	4 P.	56.5	65.3	31.5	53.2	18.9	27.0	12.7	10.2
16	ASM	62.2	93.6	37.4	70.0	25.3	63.2	19.4	57.4
	NROFF	46.5	69.3	23.5	55.7	9.0	23.5	4.1	1.7
	SORT	54.9	41.6	28.7	40.0	16.5	40.0	9.1	22.1
	VPCC	59.6	72.2	33.6	56.6	21.7	17.5	15.5	12.2
	4 P.	56.6	65.3	31.6	53.3	19.0	27.2	13.2	10.4
24	ASM	62.8	93.6	37.8	70.1	25.5	63.3	19.9	61.4
	NROFF	46.5	69.3	23.5	55.7	9.0	23.5	4.1	1.7
	SORT	54.9	41.6	28.7	40.0	16.5	40.0	9.1	22.1
	VPCC	59.6	72.2	33.6	56.6	21.7	17.5	15.5	12.2
	4 P.	56.8	65.3	31.7	53.3	19.1	27.2	13.3	10.6
32	ASM	63.5	93.7	38.8	70.1	26.5	63.5	20.3	61.6
	NROFF	46.5	69.3	23.5	55.7	9.0	23.5	4.1	1.7
	SORT	54.9	41.6	28.7	40.0	16.5	40.0	9.1	22.1
	VPCC	59.6	72.2	33.6	56.6	21.7	17.5	15.5	12.2
	4 P.	56.9	65.4	31.9	53.3	19.3	27.2	13.4	10.6

Table 4.6: Percentage of Calls which Issue a *Clear-TBS* Instruction

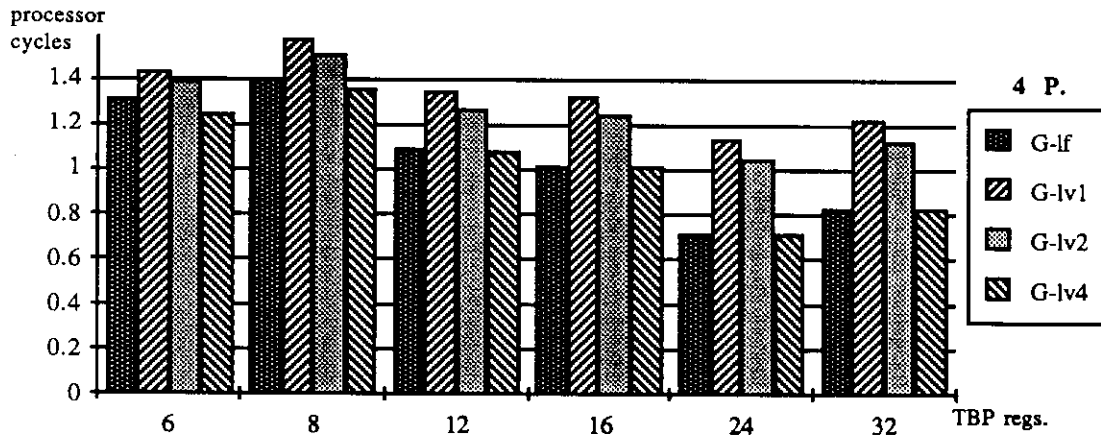


Figure 4.9: RSR Traffic for Policy G-live with *Clear-TBS* Instruction Traffic

smaller than the one generated by Policy G-lf (see Figure 4.8). However, when we account the number of cycles required to execute the extra instructions, there is no reduction in the number of cycles required (see Figure 4.9). This is true for every measured program when there are at least 16 TBP registers. Thus, we conclude that our measurements do not justify the implementation of this optimization for Policy G-lf.

2. Policy C-lv1 generates between 58% and 65% of the Policy C-lf RSR traffic (including the optimized traffic caused by the dynamic mask). If we compare the traffic generated by Policies C-lf and C-lv1 with the RSR traffic for Policies B-lf and A-lvOpt, our measurements do not justify the implementation of either Policy C-lf or C-lv1 (see Figure 4.10), because:
 - (a) For small register sets, both static Policies B-lf and A-lvOpt generate less RSR traffic than Policy C-lf and Policy A-lvOpt generates less traffic than Policy C-lv1. The lack of any real reduction in RSR traffic is a consequence of having to save/restore the dynamic mask.
 - (b) For larger register sets, Policy A-lvOpt still generates less traffic than Policy C-lf and although Policy C-lv1 generates less RSR traffic than Policy A-lvOpt, the total number of execution cycles required by Policy C-lv1 is larger than the one required by Policy A-lvOpt because of the additional instructions needed to manipulate the dynamic mask. This is also true for Policies C-lv2, C-lv3, and C-lv4.

Note: without architectural support the dynamic mask has to be updated by explicit instructions generated by the compiler. These instructions should perform the operations given in Figure 3.6. In this case, the implementation of Policy C is even less attractive because of the additional cycles required to execute the operations that would be performed in parallel if architectural support were provided

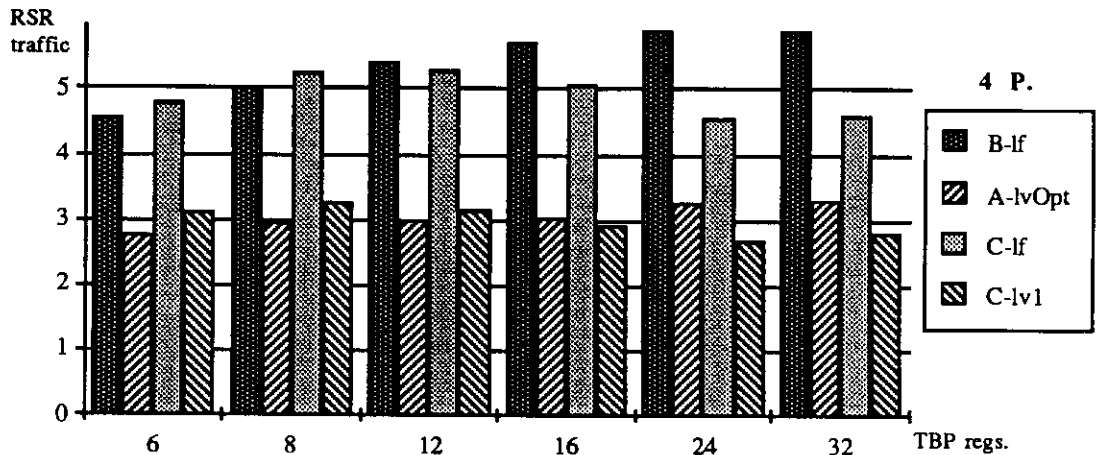


Figure 4.10: RSR Traffic for Policy C-live

(e.g., the set/reset operations on the *TBS* mask). Thus, we do not agree with others [Stee80, Cohe88] who would implement this policy.

One additional advantage of this intra-procedural optimization is that it reduces the average number of registers being used and, therefore, the average number of registers to be saved/restored during context switching. This is shown in Figure 4.11. However, this does not change our previous conclusions since context switching is performed infrequently.

4.1.5 Summary

The RSR traffic for the optimized Policies B-lf, A-live, A-lvOpt and G-lf is shown in Figure 4.12. This figure does not show the RSR traffic for Policies C-live and G-live because of the increase in processor cycles required by the additional instructions to be executed by this optimization compared to the cycles eliminated by the RSR operations which the compiler is able to remove. When we compare the optimized policies, we conclude the following:

1. When an optimizing compiler is used with live-variable analysis, it is better to save/restore registers at the caller rather than at the callee. Policy A-live generates between 85% and 95% of Policy B-lf RSR traffic and Policy A-lvOpt between 53% and 61%.

However, as we have discussed in Subsection 4.1.1, this is not true for every measured program. Policy B-lf for SORT performs better than Policy A-live for any register-set configuration and better than Policy A-lvOpt for a small register set. For ASM, Policy A-lvOpt always has less RSR traffic than Policy B-lf, but Policy A-live generates more RSR traffic when 16 or more TBP registers are available (see

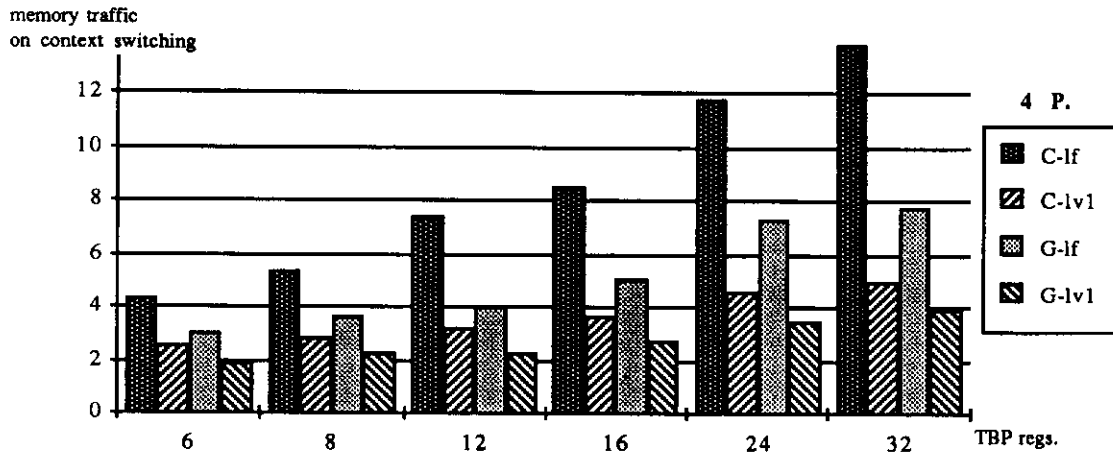


Figure 4.11: Average Number of Registers To Be Saved During Context Switching

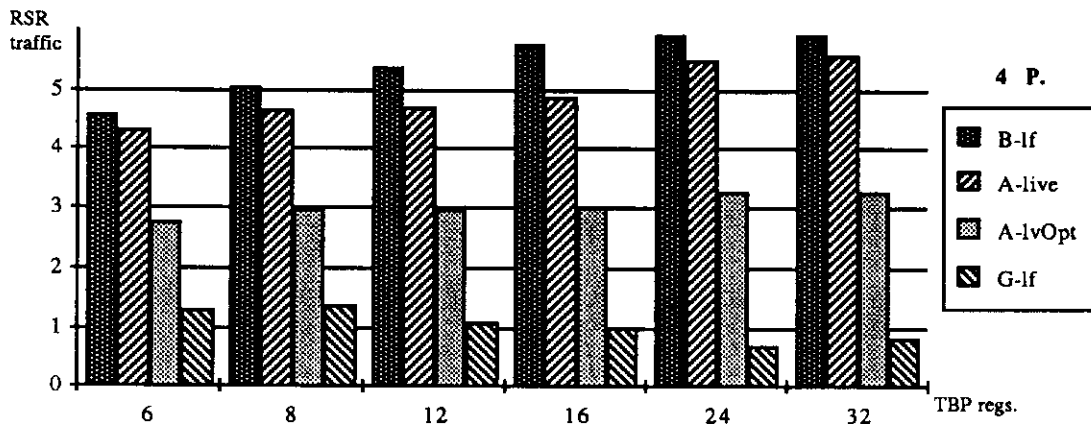


Figure 4.12: RSR Traffic for Optimized Policies

Figure 4.13). For NROFF and VPCC, both Policies A-live and A-lvOpt reduce the RSR traffic significantly with respect to Policy B-lf. Therefore, we would like the compiler to decide which policy would be the most appropriate for a given program. Since this cannot be done with intra-procedural register allocation and assignment because the compiler does not have a global view of the program, the compiler should provide an option so that the programmer could specify the policy to be selected.

2. An optimizing compiler with one of the static policies generates, in general, less RSR traffic than an optimizing compiler with the dynamic Policy C (not shown in Figure 4.12). Policy A-lvOpt has between 51% and 64% of the traffic generated by C-lf. Policy B-lf has between 85% and 91% of the C-lf traffic when up to 12 TBP registers are available; however, Policy C-lf performs better than B-lf for 16

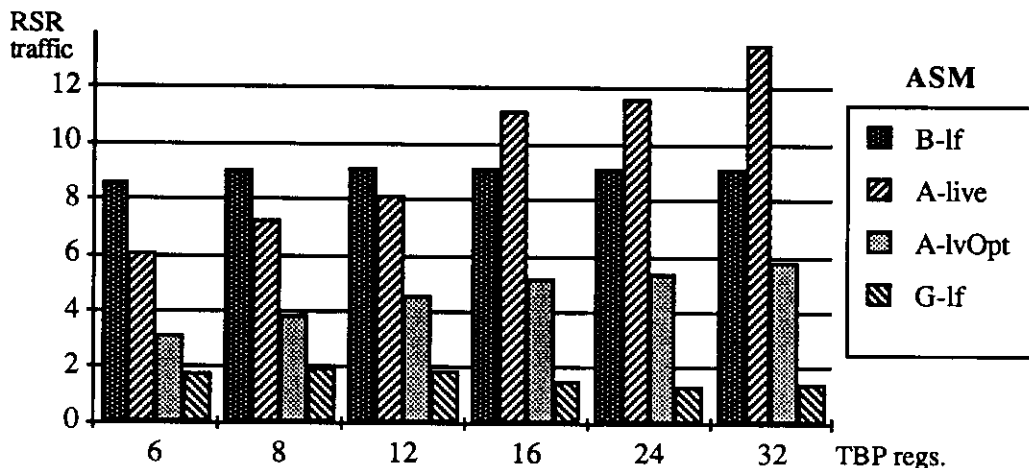


Figure 4.13: ASM RSR Traffic for Optimized Policies

(98%), 24 (86%), and 32 (88%) registers. Therefore, the register saving/restoring traffic reduction obtained by the *cheap* dynamic Policy C does not justify its implementation.

3. The optimized dynamic Policy G, G-lf, generates less RSR traffic than the best static one. Policy G-lf has between 47% (for 6 TBP registers) and 22% (for 24) of the RSR traffic generated by Policy A-lvOpt.
4. Policy G-lf is the only optimized policy whose RSR traffic decreases for larger register sets. When there are 32 registers to be preserved across function calls, the RSR traffic generated is 63% of the one generated when there are 6.

Notice that the last two conclusions were also reached in Section 3.6 where no optimizations were performed.

Since the reduction in RSR traffic for Policy G-lf is significant with respect to Policy A-lvOpt (22–47%), it seems reasonable to implement the dynamic Policy G.

4.2 A Comparison with Other Register Allocators

To verify that our previous conclusions are still valid for other register allocation approaches, we have measured the RSR traffic reduction with two other compilers: the Amsterdam Compiler Kit (ACK) [Tane83, ACK85] and the GNU C Compiler [Stal88]. In this section, we first describe the register allocation approaches taken by these compilers⁴ and afterwards we compare the RSR traffic generated.

⁴Both compilers have been modified to generate code for different register sets and to perform round-robin register assignment as it has been described for PCC.

ACK has an intra-procedural register allocator with Policy B-lf as the conventional register saving/restoring policy. Local scalar variables of type `char`, `short`, and `double` as well as other local scalar variables with an alias are not considered for allocation. The selection of which local variables are allocated to registers is based on *static linear priority*,⁵ i.e., frequently-used variables have a higher priority to be assigned to registers (see Section A.1). As in PCC, there is *no sharing* of registers among local variables per function. That is, once a variable is assigned to a register, the register is exclusively dedicated to this variable although the variable may not be alive during the whole scope of the function.

GNU has also an intra-procedural register allocator with Policy B-lf as the conventional register saving/restoring policy. All local scalar variables without aliases are considered for allocation. The register allocator performs live-variable analysis so that local variables which are alive in any of the function calls are assigned to TBP registers; otherwise, to TBD registers. Notice that since a leaf function does not have any call, local variables are assigned to TBD registers directly by the allocator and not by a peephole optimizer, as it is done in ACK. Since the allocator knows the live/dead variables, local variables with disjoint lifetimes *share* the same register (see Section A.2). The selection of which local variables are allocated to registers is a function based on the static linear priority of the variables and their live range.

Table 4.7 shows the RSR traffic for Policies B-lf and G-lf for both compilers and for Policies A-live and A-lvOpt for the GNU compiler⁶ relative to PCC Policy B-lf. Let us discuss first each policy separately.

Figure 4.14 shows the RSR traffic generated by Policy B-lf for the three compilers. As we can see, the RSR traffic generated by ACK is very similar to the one generated by PCC. On the average, ACK generates between 93% (for 24 TBP registers) and 96% (for 6) of RSR traffic produced by PCC. This traffic reduction occurs because ACK does not allocate local scalar variables of type `char` and `short` to registers. ACK generates less RSR traffic than PCC for all programs but SORT (see Table 4.7), because ACK fails to perform leaf-function optimization for a parameter defined of class `register` by the programmer in a function which receives 40% of the calls.

On the other hand, GNU generates between 54% (for 32) and 61% (for 6) of the RSR traffic produced by PCC. GNU reduces the RSR traffic further because it shares registers among local scalar variables. Thus, we conclude that live-variable analysis should be performed to reduce the average number of registers used in a function since this reduces significantly the RSR traffic caused by Policy B.

⁵In *static linear priority*, each occurrence of the variable is counted as one independently of the place where it occurs (i.e., whether the variable is inside a loop). In contrast, when variables are selected by a *static weighted priority*, variables inside a loop have more weight than variables outside. In Section A.1 we evaluate the selection of variables by static linear priority and we conclude that our measurements do not show the need for having a more complex selection.

⁶The ACK RSR traffic for Policies A-live and A-lvOpt could not be measured because the compiler uses global jumps to implement the *switch* statement. These jumps cannot be traced by our BKGGEN to obtain the live-variable information.

no. regs.	program	PCC		ACK		GNU			
		B-lf		B-lf	G-lf	A-live	A-lvOpt	B-lf	G-lf
6	ASM	1.0	(8.57)	0.78	0.29	0.65	0.46	0.64	0.26
	NROFF	1.0	(4.06)	0.93	0.17	0.67	0.41	0.43	0.11
	SORT	1.0	(2.62)	1.29	0.46	3.62	2.56	0.99	0.48
	VPCC	1.0	(6.97)	0.93	0.35	0.70	0.56	0.72	0.39
	4 P.	1.0	(4.59)	0.96	0.27	1.02	0.71	0.61	0.26
8	ASM	1.0	(9.02)	0.77	0.27	0.79	0.53	0.65	0.25
	NROFF	1.0	(4.09)	0.93	0.18	0.67	0.41	0.43	0.06
	SORT	1.0	(3.44)	1.22	0.38	3.24	2.14	0.98	0.37
	VPCC	1.0	(8.09)	0.89	0.27	0.71	0.55	0.65	0.31
	4 P.	1.0	(5.04)	0.94	0.24	1.05	0.71	0.60	0.20
12	ASM	1.0	(9.12)	0.77	0.24	1.17	0.63	0.65	0.25
	NROFF	1.0	(4.09)	0.93	0.14	0.68	0.42	0.43	0.03
	SORT	1.0	(5.04)	1.15	0.30	2.84	1.70	0.82	0.19
	VPCC	1.0	(8.27)	0.89	0.25	0.71	0.55	0.64	0.23
	4 P.	1.0	(5.40)	0.95	0.21	1.13	0.72	0.59	0.14
16	ASM	1.0	(9.12)	0.77	0.16	1.43	0.69	0.65	0.23
	NROFF	1.0	(4.09)	0.93	0.12	0.68	0.42	0.43	0.02
	SORT	1.0	(6.60)	1.11	0.22	2.17	1.30	0.63	0.18
	VPCC	1.0	(8.39)	0.89	0.23	0.70	0.54	0.63	0.23
	4 P.	1.0	(5.74)	0.95	0.18	1.08	0.68	0.55	0.14
24	ASM	1.0	(9.12)	0.77	0.21	1.43	0.69	0.65	0.22
	NROFF	1.0	(4.09)	0.93	0.10	0.68	0.42	0.43	0.01
	SORT	1.0	(7.37)	1.00	0.20	1.94	1.16	0.56	0.02
	VPCC	1.0	(8.46)	0.92	0.16	0.69	0.53	0.62	0.13
	4 P.	1.0	(5.92)	0.93	0.15	1.05	0.66	0.54	0.07
32	ASM	1.0	(9.12)	0.77	0.14	1.43	0.69	0.65	0.22
	NROFF	1.0	(4.09)	0.93	0.08	0.68	0.42	0.43	0.01
	SORT	1.0	(7.37)	1.00	0.09	1.94	1.16	0.56	0.01
	VPCC	1.0	(8.46)	0.95	0.21	0.69	0.53	0.62	0.17
	4 P.	1.0	(5.92)	0.94	0.13	1.05	0.66	0.54	0.07

Table 4.7: RSR Traffic Generated by ACK and GNU Relative to PCC Policy B-lf

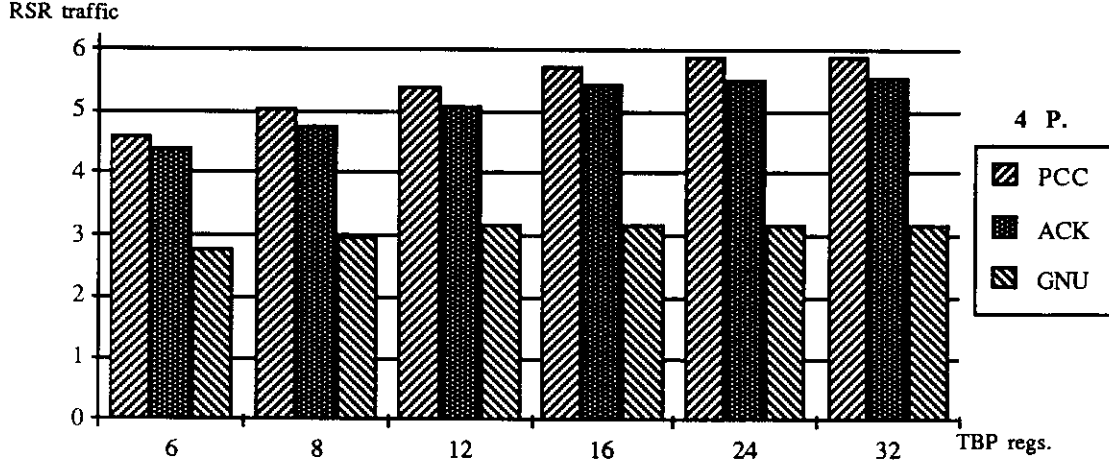


Figure 4.14: RSR Traffic for Policy B-lf

Figure 4.15 shows the RSR traffic generated by Policy G-lf for the three compilers. As we can see in the figure, for some register configurations ACK generates less RSR traffic than PCC and for others the opposite is true. These small differences in RSR traffic are due to the register assignment performed by each compiler and the dynamic behavior of Policy G. As for Policy B-lf, GNU generates the smallest traffic. The traffic reduction is more significant for larger register sets: GNU generates 90% of the RSR traffic produced by PCC when 6 TBP registers are available, 70% for 12, and 53% for 24, because with GNU registers are shared by the variables with disjoint lifetimes. Since the average number of registers used per function decreases, the probability of having a disjoint register assignment between caller and callee increases.

Figure 4.16 shows the RSR traffic generated by Policies A-live and A-lvOpt for PCC and GNU. GNU generates more RSR traffic than PCC for both policies. GNU has between 108% and 129% of the RSR traffic produced by PCC for Policy A-live and between 118% and 129% for Policy A-lvOpt. Thus, even though GNU uses, on the average, fewer registers per function, more registers are alive (i.e., the variables assigned to registers) for GNU than PCC. The increase of the number of live registers by GNU is due to the compiler-defined variables generated by GNU, but not by PCC (e.g., for common expressions, loop-invariant removal).

If we compare the RSR traffic generated by the optimized Policies B-lf, A-live, A-lvOpt, and G-lf for the GNU compiler, as we did in Subsection 4.1.5 for PCC, we obtain similar conclusions (see Figure 4.17):

1. Although on the average Policy B-lf generates less RSR traffic than both Policies A-live and A-lvOpt, this is not the case for every single program as it was not for PCC. Policy A-lvOpt is the static policy which generates the least RSR traffic for ASM, NROFF, and VPCC for both compilers PCC and GNU. As we said, a compiler with an intra-procedural register allocator should provide an option to let

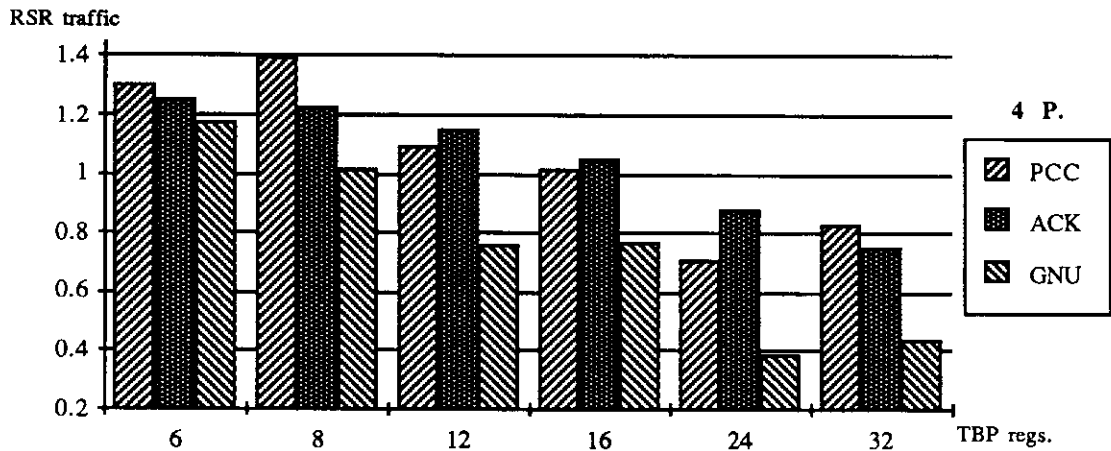


Figure 4.15: RSR Traffic for Policy G-lf

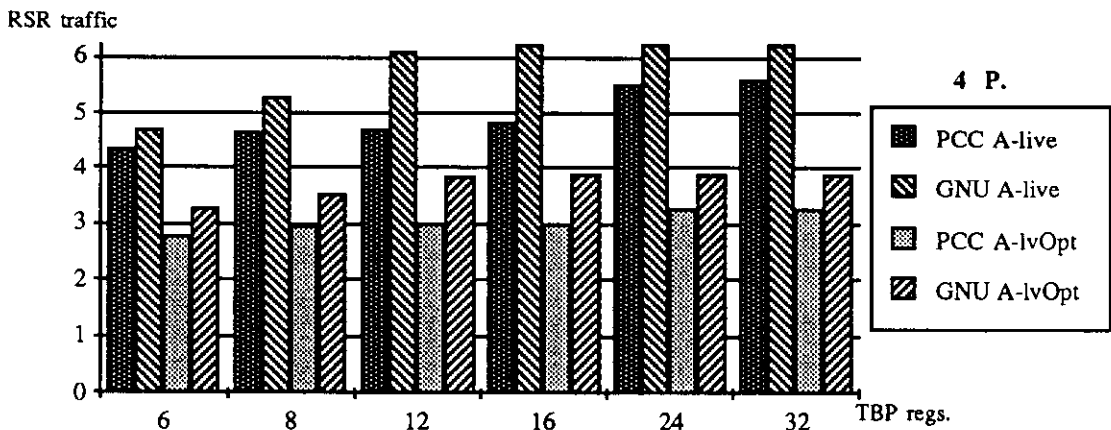


Figure 4.16: RSR Traffic for Policies A-live and A-lvOpt

the programmer decide which is the most appropriate policy to be used for each specific application.

2. The dynamic Policy G-lf generates less RSR traffic than the static policies. Policy G-lf has between 13% (for 24 TBP registers) and 42% (for 6) of the RSR traffic produced by Policy B-lf and between 10% (for 24) and 36% (for 6) by Policy A-lvOpt.

Moreover, this last conclusion is also true for ACK with respect to Policy B-lf (since Policies A-live and A-lvOpt are not available). Policy G-lf generates between 13% (for 32) and 28% of the RSR traffic produced by Policy B-lf.

In conclusion, the intra-procedural optimizations presented in Section 4.1 can be used by any intra-procedural register allocator to obtain a reduction in the RSR traffic

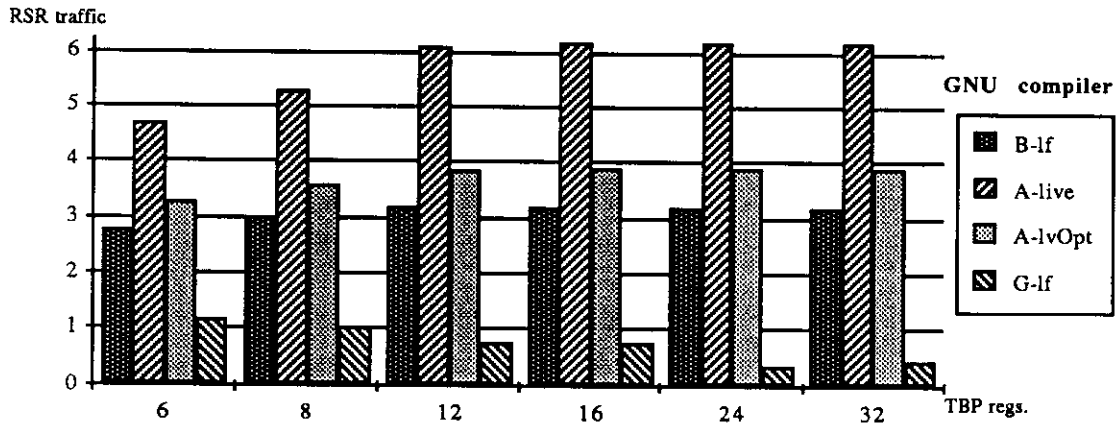


Figure 4.17: RSR Traffic for GNU Compiler

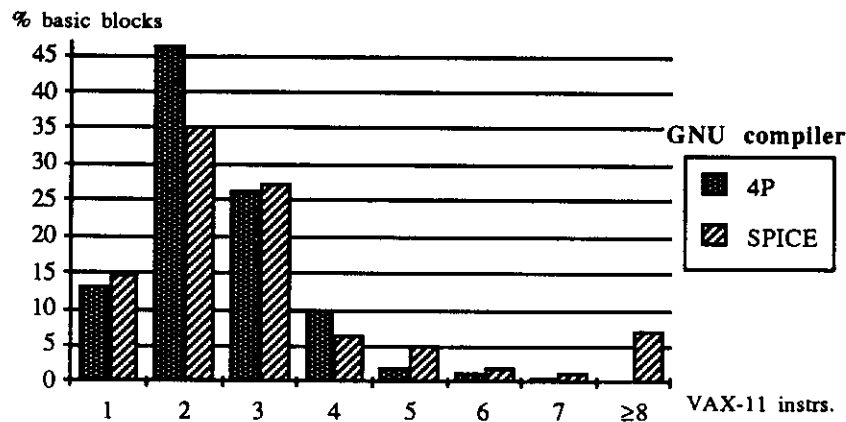


Figure 4.18: VAX-11 Instructions per Executed Basic Block

generated. The conclusions obtained for the intra-procedural optimizer are valid for all three register allocators used in this dissertation, although the exact RSR traffic eliminated by each policy is slightly different.

4.3 A Comparison with SPICE

In this section we compare the best static optimized Policies (A-live, A-lvOpt, and B-lf) with the best dynamic one (G-lf) using **SPICE** and the GNU C Compiler. **SPICE** is a circuit simulator which was originally written in **FORTRAN** and has now been recoded in **C**. We have separated this program from the other four (**ASM**, **NROFF**, **SORT**, and **VPCC**) because:

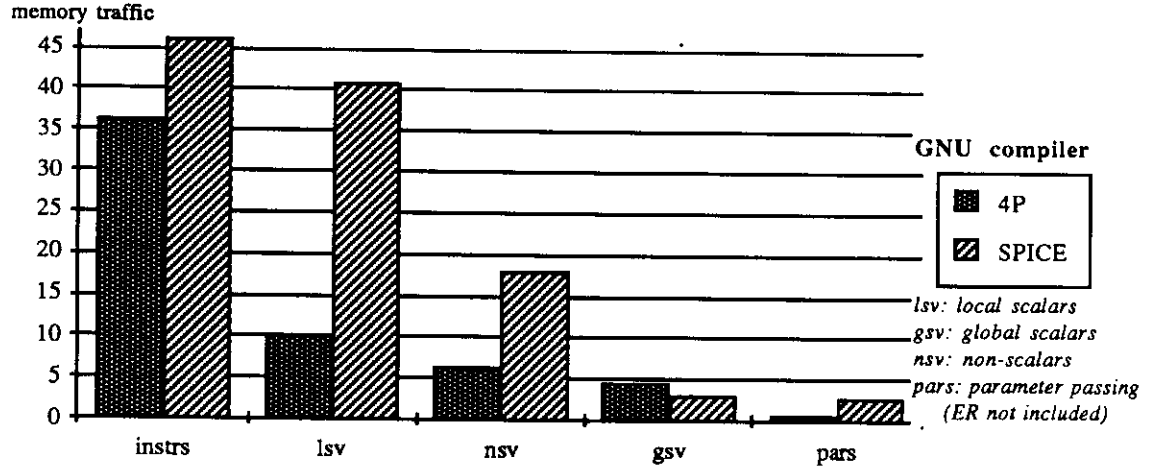


Figure 4.19: Data Memory Traffic per Executed Function

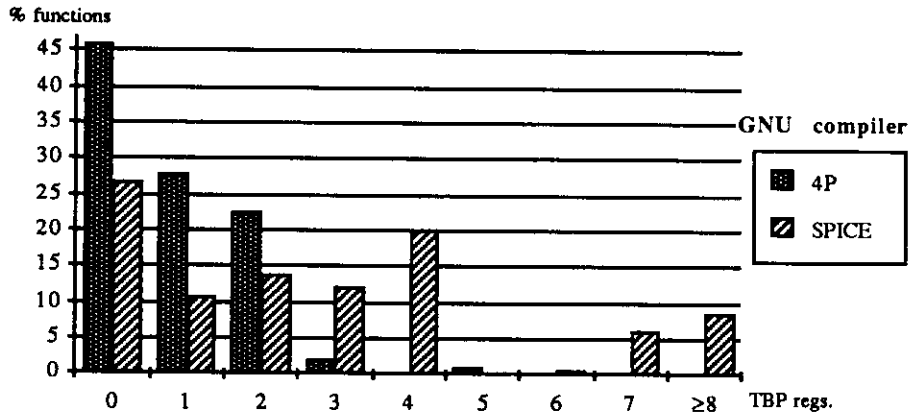


Figure 4.20: To-Be-Preserved Registers Required per Executed Function

- Since SPICE was originally programmed in FORTRAN, it has larger basic blocks (3.7 VAX instructions for SPICE versus 2.5 for the four programs; see Figure 4.18) and fewer function calls (46 VAX instructions per call for SPICE versus 37 for the four programs).
- SPICE has four times the local scalar variable traffic generated by the four programs (see Figure 4.19). Thus, as we will discuss in Section 4.6, a more significant data memory traffic reduction is obtained for SPICE with a larger register set.
- SPICE relies heavily on floating-point variables. Since most of these variables are local doubles, more registers are required per function (see Figure 4.20). On the average, the GNU C compiler uses 5.9 registers per executed function for SPICE and 3.2 for the other four programs. Moreover, functions that require more than 32 registers generate 64% of the local scalar variable traffic for SPICE while only 1% for the other four.

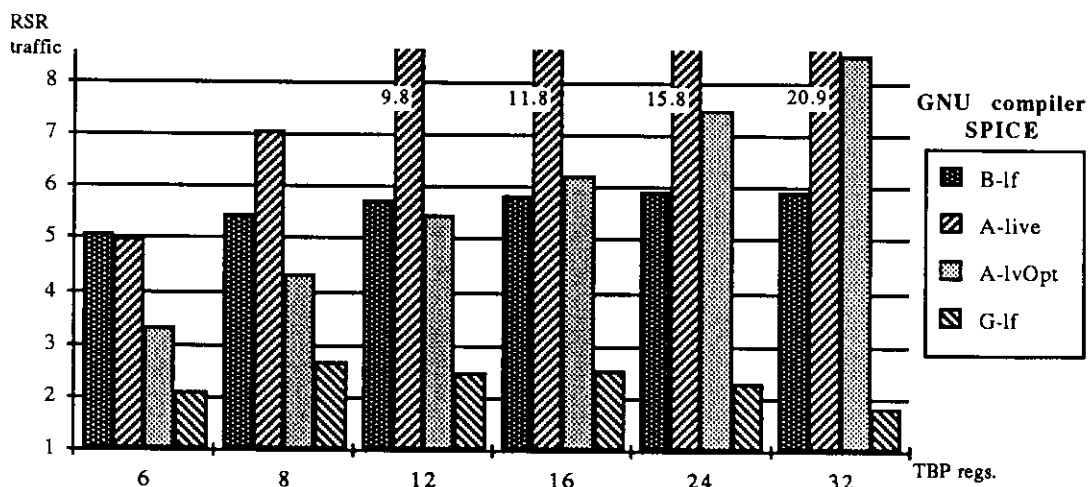


Figure 4.21: RSR Traffic Generated by the GNU C Compiler for SPICE

The reason for selecting only the GNU C Compiler to perform this comparison is because GNU is the only compiler designed to assign both float and double local variables to registers. The Portable C Compiler does not assign any floating-point variable to registers, and ACK only assigns floats.

Figure 4.21 shows the RSR traffic for the four policies. From the figure, we conclude the following:

1. Policy B-lf generates less RSR traffic than both Policies A-live and A-lvOpt for larger register sets (16 or more registers); however, Policy A-lvOpt generates less RSR traffic for smaller register sets. This can be explained as follows:
 - (a) For small register sets, the average number of variables assigned to registers alive at each call is greater than the average number of registers used per function. Usually it is the case that the callers use more registers than the callees and, therefore, Policies A and A-live generate more RSR traffic than Policies B and B-lf (see Sections 3.1 and 4.2). However, since not all of the registers with live variables are needed between calls, Policy A-lvOpt produces less RSR traffic than both Policies A-live and B-lf.
 - (b) For larger register sets, more local variables are assigned to registers and, therefore, the number of registers with live variables keeps increasing when more registers are available. Moreover, as we said above, since callers tend to use more registers than callees, policies of type A need to save/restore more registers than policies of type B. Policy A-live has 200% of the RSR traffic generated by Policy B-lf for 16 TBP registers, 267% for 24, and 350% for 32. Although not all the registers need to be saved/restored between calls, Policy A-lvOpt also has to save/restore, on the average, more registers than Policy B-lf. However, the RSR traffic generated does not increase as significantly as with Policy A-live: Policy A-lvOpt has between 107% of the RSR traffic

generated by Policy B-lf (for 16 TBP registers) and 144% (for 32).

2. The RSR traffic for both Policies A-live and A-lvOpt grows rapidly when more TBP registers are available. This was not the case for the other four programs (compare Figure 4.21 with Figure 4.17). When there are 32 TBP registers, Policy A-live for SPICE has 419% of the traffic produced when there are 6, while for the other programs it has 132%, and Policy A-lvOpt has 257% for SPICE versus 119% for the other four. However, this is not the case for Policy B-lf (117% for SPICE versus 114% for the other four): since the callers have more register usage than the callees, more registers have to be saved/restored for policies of type A.
3. With SPICE, Policy G-lf is still the policy that generates the least RSR traffic. Policy G-lf has between 21% (for 32 TBP registers) and 63% (for 6) of the RSR traffic generated by Policy A-lvOpt and between 31% (for 32) and 48% (for 8) by Policy B-lf. Moreover, an increase in the number of TBP registers does not imply an increase in the RSR traffic for Policy G-lf. This is also the case concerning the four other programs (see Subsection 4.1.5 and Section 4.2). However, the RSR traffic reduction is not as significant for SPICE as for the average of the other programs due to SPICE's heavier register. When there are 32 TBP registers available, Policy G-lf for SPICE has 87% of the traffic produced when there are 6 while for the four programs it has 37%.

In conclusion, even for a program with a heavy register usage, the dynamic Policy G-lf generates less RSR traffic than the static Policies A-live, A-lvOpt, and B-lf. Moreover, the heavier register usage does not prevent the round-robin register assignment algorithm from using disjoint registers between caller and callee so that the RSR traffic is reduced when more TBP registers become available.

4.4 A Comparison with Multiple-Window Register Files

In this section we compare the RSR traffic generated by our best policy for single-window architectures (Policy G-lf) with the three existing schemes for multiple-window architectures: fixed-size windows [Kate83], variable-size windows [Ditz82], and multi-size windows [Hugu85c]. As we mentioned in Subsection 2.1.1, a larger register file with multiple windows almost eliminates the RSR traffic [Hitc85, Hugu85a, Eick87], but this file has a negative effect on the processor cycle time [Sher84, Ditz87b]. To prevent this negative effect, a 32-register file might be more convenient and, in this case, we will show that our Policy G-lf generates the least RSR traffic.

The RSR traffic has been measured with two C compilers: PCC and GNU. The reason for selecting both compilers is that PCC assigns one local scalar variable per register while GNU performs live-variable analysis and allows register sharing for local scalar variables with disjoint lifetimes (see Section 4.2). Thus, we can evaluate the RSR traffic reduction offered when live-variable analysis is performed and registers are shared.

no. regs.	program	PCC			GNU		
		fs12	ms3	vs	fs12	ms3	vs
32	ASM	4.060	2.439	0.520	4.060	2.248	0.399
	NROFF	11.898	1.389	0.193	11.898	1.153	0.018
	SORT	4.754	0.201	0.040	4.754	1.086	0.006
	VPCC	8.578	3.470	0.786	8.559	2.378	0.379
	4 P.	9.404	1.647	0.305	9.400	1.452	0.110
64	ASM	0.183	0.008	0.002	0.183	0.005	0.000
	NROFF	2.041	0.147	0.000	2.041	0.033	0.000
	SORT	0.008	0.008	0.001	0.008	0.001	0.000
	VPCC	1.424	0.529	0.161	1.411	0.273	0.082
	4 P.	1.419	0.196	0.036	1.416	0.078	0.018

Table 4.8: RSR Traffic for Multiple-Window Register Files

The RSR traffic have been measured for the following configurations (see Table 4.8):

- 32 and 64-register files.
- Fixed-size windows of 12 registers (labeled *fs12*). One window is transferred to (from) memory on overflow (underflow). Our measurements have shown that a 12-register window is enough for 96% of the executed functions for the PCC and for 100% for the GNU compiler. However, most of the registers are being unused: on the average, PCC has 7.9 unused registers per window and GNU, 9.4.
- 3-size windows of 4, 8, and 12 registers (labeled *ms3*). On overflow, only the smallest window required is transferred to memory.⁷ On underflow, the largest window (12 registers in this case) is always transferred. Multi-size windows reduce the average number of unused registers to 1.31 for the PCC and to 2.18 for the GNU compiler.
- Variable-size windows (labeled *vs*) of maximum size 12. In this case, the exact number of registers is transferred since both instructions to expand the window and instructions to check for underflow are made explicit by the compiler. No compiler optimizations like the ones proposed by [Band87] have been implemented to reduce the RSR traffic even further (for instance, the instruction to check for underflow can be eliminated if the caller does not need to use the TBP registers because it is going to return).

⁷This policy generates less RSR traffic than the policy of always saving the largest window on overflow, the policy proposed originally [Hugu85a]. However, it generates more overflows. For instance, for a 32-register file, 19% of the calls generate an overflow exception for the former policy versus 17% for the latter (using PCC). To decide which policy is the best we should consider not only the RSR traffic, but also the number of cycles taken by the exception handler. However, in this section we show only the results for the first policy to simplify the raw data discussed.

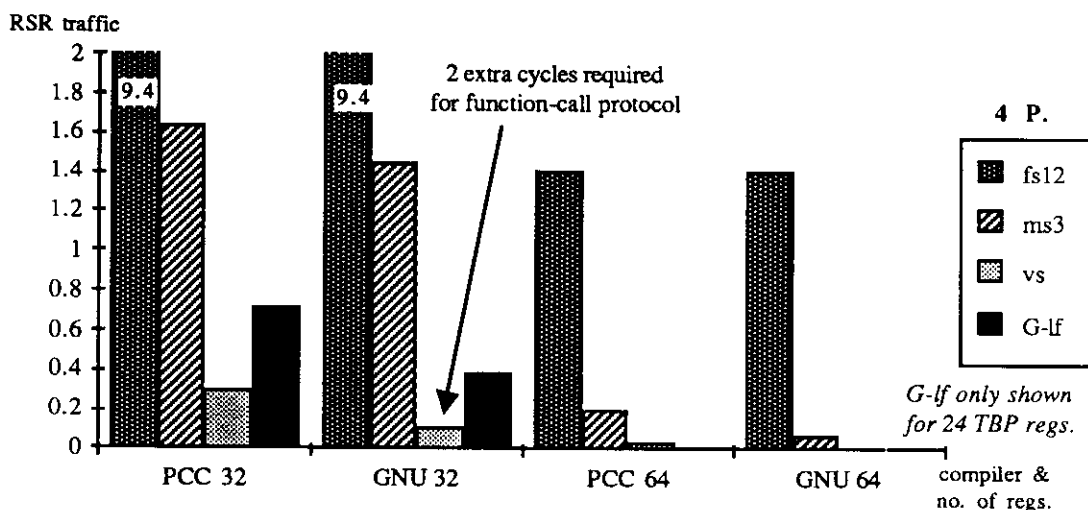


Figure 4.22: RSR Traffic for Multiple-Window Register Files

program	PCC	GNU
ASM	93.9%	26.4%
NROFF	10.0%	1.5%
SORT	22.1%	22.1%
VPCC	56.4%	37.0%
4 P.	26.1%	14.4%

Table 4.9: Percentage of Functions which Must Expand the Window Size

Figure 4.22 shows the RSR traffic generated by the four programs for the above configurations plus the RSR traffic generated by Policy G-lf when 24 TBP registers are available.⁸ As we can see, variable-size windows generate the least RSR traffic. However, if we consider that each function requires almost two extra memory cycles due to the two extra instructions,⁹ then the overall number of cycles required will be worse than the ones required by both multi-size windows and Policy G-lf.

For multi-size windows our measurements show that, on the average, 26% of the executed functions are required to issue an instruction to expand the window size to more than 4 registers for the PCC and 14% for the GNU compiler (see Table 4.9). Thus, most of the time only two instructions are necessary (as in fixed-size windows) to implement the function-call protocol when the allocator does not require more registers for a function. For this reason, the number of cycles necessary to execute the instructions for handling the register file plus the ones to perform the RSR traffic required by multi-size windows is smaller than for variable-size windows. For instance, for the GNU compiler, multi-size windows have 75% of the cycles required by variable-size windows for a 32-register file,

⁸We assume that a 32-register file is divided into 24 TBP registers and 8 environment and TBD registers. For this reason Figure 4.22 only shows the RSR traffic for Policy G-lf with 24 TBP registers.

⁹We said “almost” because as we mentioned above, some optimizations have been proposed to eliminate the instruction to check for underflow [Band87].

no. regs.	program	PCC			GNU		
		fs12	ms3	vs	fs12	ms3	vs
32	ASM	16.9%	28.5%	7.1%	16.9%	22.4%	6.4%
	NROFF	49.6%	17.3%	4.8%	49.6%	14.4%	0.6%
	SORT	19.8%	1.7%	0.6%	19.8%	9.1%	0.1%
	VPCC	35.7%	37.7%	9.3%	35.7%	28.3%	6.9%
	4 P.	39.2%	19.1%	5.0%	39.2%	16.7%	2.2%
64	ASM	0.8%	0.1%	0.0%	0.8%	0.1%	0.0%
	NROFF	8.5%	1.8%	0.0%	8.5%	0.4%	0.0%
	SORT	0.0%	0.1%	0.0%	0.0%	0.0%	0.0%
	VPCC	5.9%	5.4%	1.4%	5.9%	3.4%	1.1%
	4 P.	5.9%	2.2%	0.3%	5.9%	1.0%	0.2%

Table 4.10: Percentage of Overflows in Multiple-Window Register Files

and 11% for a 64-register file.

When registers are shared by local scalar variables with disjoint lifetimes, more functions fit in the smallest window size so that fewer instructions to expand the window size have to be issued for multi-size windows (see Table 4.9; also, notice the large difference that sharing registers makes for ASM). Consequently, fewer overflows are generated (see Table 4.10). The GNU compiler has 88% of the RSR traffic generated by PCC for a 32-register file and 40% for a 64-register file. This RSR traffic reduction is also significant for variable-size windows: 36% and 50%, respectively. On the other hand, the RSR traffic generated for fixed-size windows should be identical for both compilers. However, there is a slight difference due to the RSR traffic generated by VPCC, which is caused by a different implementation of a few library functions in both compilers. Thus, architectures with fixed-size windows cannot benefit from sharing registers, except when there are more local scalars than registers.

Therefore, although variable-size windows generate the least RSR traffic, multi-size windows require less cycles to implement the function-call protocol and, as a result, they provide the best performance for multiple-window register files. Let us now compare multi-size windows with Policy G-lf. This comparison is performed in two steps: first, we consider only the RSR traffic and, second, we evaluate the cycles which perform these RSR operations.

If we just consider the RSR traffic generated for a 32-register file, Policy G-lf has 44% of the RSR traffic produced by 3-size windows for PCC and 27% for GNU.

Let us now consider the processor cycles required by these two schemes. We already discussed above that 3-size windows require 0.26 extra cycles per function call for the PCC and 0.14 for the GNU compiler due to the extra instruction which must be issued to expand a window (see Table 4.9). For Policy G, no extra cycles are required for the saving/restoring operations (see Section 3.5). In this case, Policy G-lf requires 38% of the processor cycles necessary for 3-size windows with PCC and 24% with GNU.

no. regs.	program	PCC			GNU		
		fs12	ms3	vs	fs12	ms3	vs
32	ASM	28.00	28.72	21.11	28.00	25.44	18.44
	NROFF	28.00	27.81	22.00	28.00	27.70	17.53
	SORT	28.00	30.99	26.64	28.00	30.18	22.45
	VPCC	28.00	29.13	23.81	28.00	29.44	22.42
	4 P.	28.00	28.78	23.30	28.00	28.48	19.63
64	ASM	34.50	35.09	29.47	34.50	35.49	25.48
	NROFF	47.77	43.37	27.74	47.77	40.86	18.34
	SORT	43.08	45.15	34.54	43.08	42.76	34.58
	VPCC	47.61	44.48	33.37	47.61	44.27	32.67
	4 P.	46.24	43.63	30.41	46.24	41.76	25.05

Table 4.11: Average Number of Registers To Be Saved During Context Switching

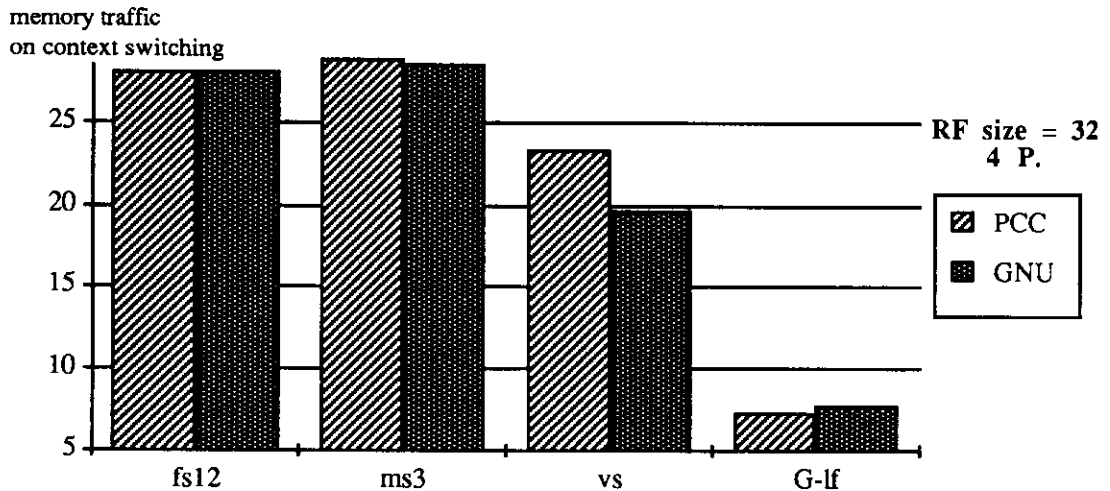


Figure 4.23: Average Number of Registers To Be Saved During Context Switching

Policy G-lf has an additional advantage over the other configurations: the smallest average number of registers which need to be saved during context switching. Table 4.11 shows the average number of registers to be saved during context switching for each program. Notice that for a 32-register file with fixed-size windows of 12 registers, exactly 28 registers need to be saved, that is, 2 windows of 12 registers each plus 4 overlapped registers. As we can see Figure 4.23, for a 32-register file, Policy G-lf has 26% of the traffic for 3-size windows with PCC and 28% with GNU. The numbers shown in the table and in the figure do not include either the TBD registers, which will have to be saved for each configuration, or the return address for Policy G-lf, because specific architectural or compiler support can be provided for the return address (see Section 5.3).

In conclusion, when a small 32-register file is available, a single-window register file with Policy G-lf generates less RSR traffic than a multiple-window register file with

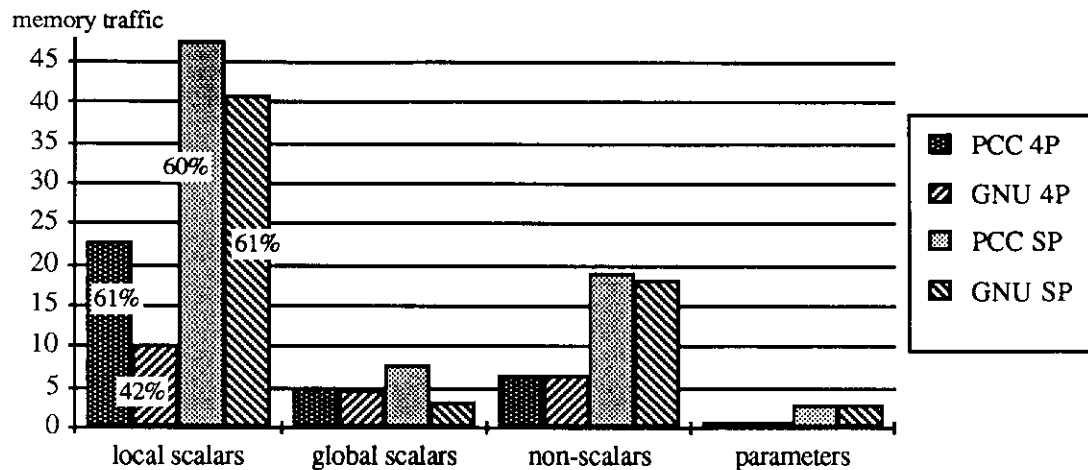


Figure 4.24: Distribution of Data Memory Traffic

multi-size windows, which is the best scheme for a multiple-window register file.

4.5 Overall Data Memory Traffic Generated

In this section the overall data memory traffic reduction is compared for Policies A-live, A-lvOpt, B-lf, and G-lf. This evaluation is performed using two compilers, PCC and GNU, and the two sets of programs: the average for the four programs (ASM, NROFF, SORT, and VPCC) and SPICE. ACK has not been included in this discussion because the RSR traffic generated by Policy B-lf for ACK is similar to the one for PCC (see Figure 4.14). This is also the case for the global scalar traffic, but not for the local scalar traffic or for the traffic caused by the non-scalar variables.

- ACK generates less local scalar traffic than PCC when the number of TBP registers is small (i.e., up to 12), because ACK selects the variables to allocate based on static linear priority, while PCC does it by the order of definition specified by the programmer (see Section A.1). For larger register sets, the traffic is similar for both compilers.
- ACK generates more than twice the PCC (and GNU) traffic caused by non-scalar variables. This can be justified because the goal of ACK is to be portable across different languages and machines.

For these reasons, we decided not to include ACK in our evaluation.

Figure 4.24 shows the distribution of data memory references for the four programs measured and for SPICE when no register allocation is performed for local scalar variables: local scalar variables, global scalar variables, non-scalar variables (i.e., arrays and

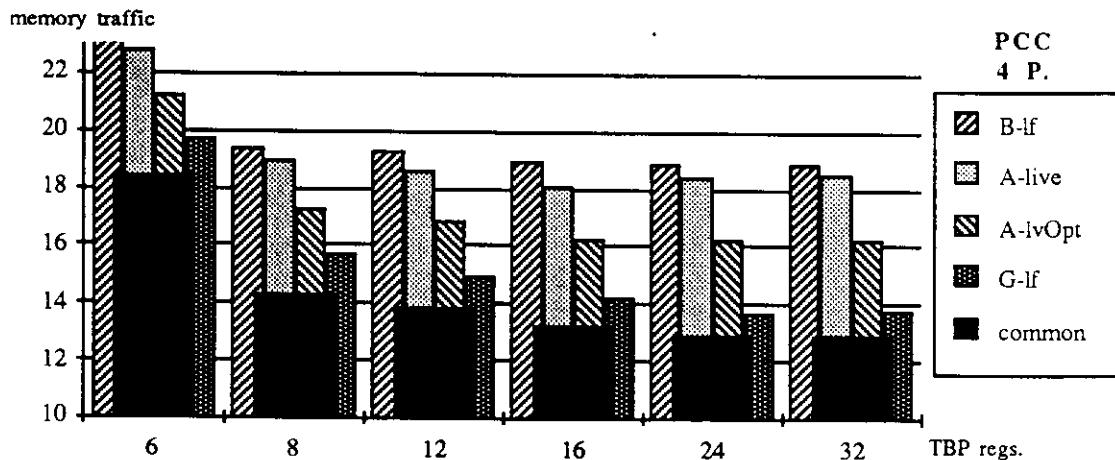


Figure 4.25: Overall Data Memory Traffic with PCC for the Four Programs

structures), and parameter passing. The figure does not show the traffic caused by the saving and restoring of the return address because it is always two memory references per executed function, except when leaf-function optimization is performed (as we discussed in Subsection 4.1.3) or when multiple PCs are provided or inter-procedural optimization is performed (as we will discuss in Section 5.3). No traffic is taken into account for temporary variables since these operations are performed in a small number of registers.¹⁰ As we can see in the figure, when local scalar variables are not allocated to registers (i.e., only local allocation is performed per basic block; see Subsection 2.1.3.2), the largest portion of the traffic is caused by local scalar variables. For the four programs, local scalar traffic accounts for 61% for PCC and 42% for GNU over the total data memory traffic while for SPICE, it is 60% and 61%, respectively.¹¹ Therefore, a compiler with an efficient intra-procedural register allocation and assignment policy (which also generates little register saving and restoring traffic) can cut in about half the memory traffic required by a program.

Figures 4.25, 4.26, and 4.27 show the overall traffic generated per function when we eliminate the traffic caused by the pushing of parameters in the stack by passing them through registers¹² for the average of the four programs with PCC and GNU and for SPICE with GNU. The *common* traffic (black boxes) corresponds to the traffic caused by the non-allocated local scalars, the global scalars, the non-scalar (arrays and structures) variables and to the return-address traffic; that is, to the overall traffic minus the RSR traffic. Notice that the reduction in the common traffic seen in the figure corresponds to the local scalar traffic reduction since this is the only traffic which decreases for a larger

¹⁰Our previous measurements on three of the five programs measured (NROFF, SORT, and VPCC) showed that two registers are enough to evaluate 95% of the arithmetic expressions [Hugu85a].

¹¹The GNU compiler generates less local scalar traffic than PCC because the latter only assigns locals to TBP registers, but the former assigns dead locals to TBD registers while using the TBP registers for live locals (see Section 4.2). Thus, even when no TBP registers are available, the GNU compiler can reduce the local traffic by assigning dead locals to TBD registers.

¹²A small number of registers (4) is enough to pass parameters through registers [Hugu85a, Karg89].

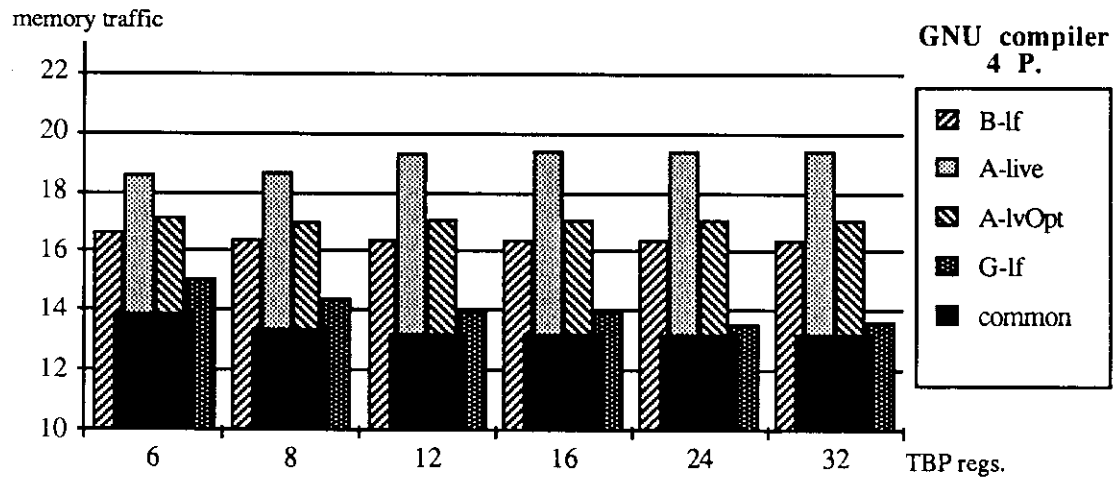


Figure 4.26: Overall Data Memory Traffic with GNU for the Four Programs

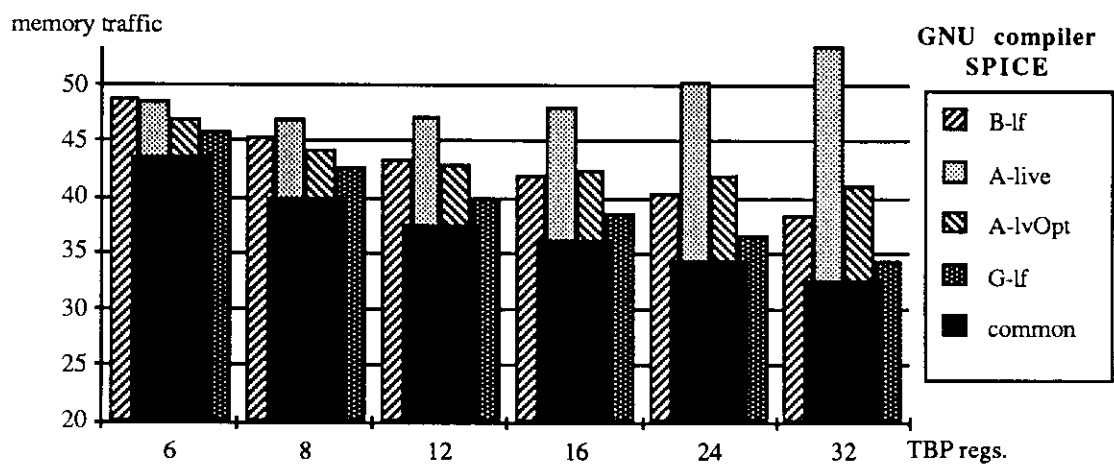


Figure 4.27: Overall Data Memory Traffic with GNU for SPICE

number of TBP registers. Let us comment first the measurements for the average of the four programs and, afterwards, for SPICE.

Since only local scalar variables are allocated by PCC (see the introduction of Chapter 3), the traffic reduction in Figure 4.25 corresponds to these when more of them are assigned to registers. Notice that almost all of them have been allocated when 16 TBP registers are available (since there is only a slight traffic reduction for the *common* traffic).

On the other hand, Figure 4.26 shows that when local scalar variables with disjoint lifetimes share the same register, even fewer TBP registers are required (see Section A.2). As we can see in the figure, the traffic remains almost constant for Policies A-lvOpt and B-lf; however, this is not the case for Policy A-live due to the significant increase in RSR traffic that we already mentioned in Section 4.2 (see Figure 4.17). Also, notice that although the *common* traffic gets smaller when the number of TBP registers is increased from 6 to 12, there is no overall data memory traffic reduction for Policies A-lvOpt and B-lf because the reduction in traffic for the non-allocated locals is compensated for by the increase in RSR traffic.

Therefore, it seems that for the four programs, both compilers do not require a register set with more than 8 TBP registers when the static policies are used to perform register saving and restoring. A larger number of TBP registers (for the measured programs) can only be justified when the dynamic Policy G is implemented. For instance, the overall traffic generated by Policies A-lvOpt, B-lf, and G-lf with the GNU compiler when 24 TBP registers are available is 101%, 100%, and 94%, respectively, compared to when there are only 8 TBP registers available.

Moreover, if we compare Policy B-lf (the best static policy) with Policy G-lf using the GNU compiler, there is also no reason to increase the number of TBP registers beyond 12. This is because the RSR traffic generated by Policy G-lf for 12 is already small (for instance, 0.76 memory references per function for the average of the four programs) and, therefore, the traffic ratio with respect to Policy B-lf remains almost the same when more TBP registers are available. For instance, for the average of the four programs, the overall traffic generated by Policy G-lf is 85% of the one produced by Policy B-lf for 12 TBP registers, and 83% for 24.

However, this is not the case for SPICE as we can see in Figure 4.27. The more TBP registers are available, the more **double** variables GNU can assign to registers. In this case, the compiler can make use of a larger number of TBP registers. Even though the RSR traffic increases for the static policies when a larger number of TBP registers is available (as we have discussed in Section 4.3; see Figure 4.21), the overall traffic becomes less and less for Policies A-lvOpt, B-lf, and G-lf because the compiler can assign more local scalars to registers (see Figures A.2 and A.4). For instance, the overall traffic generated by Policies A-lvOpt, B-lf, and G-lf when 24 TBP registers are available is of 95%, 89%, and 86%, respectively, compared to when there are only 8 TBP registers available.

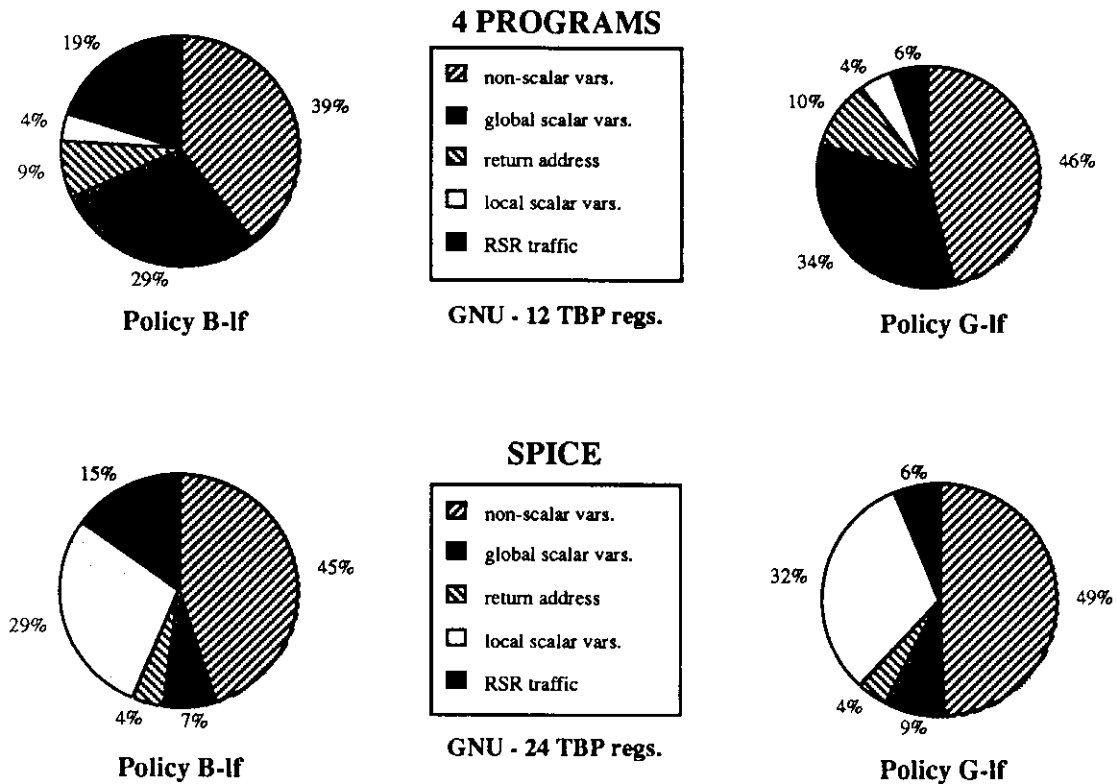


Figure 4.28: Overall Data Memory Traffic Distribution

Figure 4.28 shows the distribution of the overall data memory traffic for the average of the four programs and SPICE when code is generated with the GNU C Compiler with 12 and 24 TBP registers. Only two RSR policies are shown in the figure: Policy B-lf¹³ and Policy G-lf. When only compiler support is provided, i.e., the best of the two static policies (Policy A-lvOpt or Policy B-lf), 33% of the data memory traffic generated by the average of the four programs and 40% by SPICE is eliminated. Notice that the non-allocated local scalar traffic plus the RSR traffic is 23% for the average of the four programs and 44% for SPICE of the overall data memory traffic; in contrast, when no intra-procedural register allocation is performed for local scalar variables, the local scalar traffic accounts for 42% and 61%, respectively (see Figure 4.24).

When both architectural and compiler support are provided (i.e., Policy G-lf), 43% of the data memory traffic generated by the average of the four programs and 45% by SPICE is eliminated. In this case, the local scalar traffic plus the RSR traffic accounts for 10% and 38% of the overall traffic, respectively. The data memory traffic generated with both architectural and compiler support is 85% for the average of the four programs and 91% for SPICE of the one produced with only compiler support.

¹³The reason for mentioning Policy B-lf rather than Policies A-live or A-lvOpt is that Policy B-lf generates the least RSR traffic for the average of the four programs and for SPICE (see Figures 4.26 and 4.27).

In conclusion, our measurements have shown that for non-numeric applications (as a compiler, a sort program, a word processor, and an assembler) a register set with between 8 and 12 TBP registers should be sufficient for intra-procedural optimizers with (i.e., Policy G-lf) or without architectural support (i.e., the best static Policy A-lvOpt or B-lf for a given application). For numerical applications (such as SPICE), if the compiler can assign floating-point variables to general-purpose registers, a larger number of TBP registers is justified.

4.6 Speed-Up

In this section we evaluate the speed-up that we might obtain for a RISC-like machine. This evaluation is performed using the PCC and the GNU compilers and the two sets of programs, the average for the four programs and SPICE, as we did in the previous section. First, we comment on the model we have used to evaluate the speed-up for a RISC-like machine, and, second, we discuss the speed-up factor for Policies A-lvOpt, A-live, B-lf, and G-lf.

To estimate the speed-up obtained by Policy G-lf we assume that each register-to-register instruction is executed in one cycle, that load/store instructions require one or two extra cycles to access memory, and that the processor cycle time is not affected by the implementation of Policy G (see Section 3.5). The reason for considering the two memory speeds is that if the processor cycle keeps getting reduced, then it is possible that two extra cycles will be required to access memory because of chip bandwidth limitations. The execution time (T) of a program with N instructions is given by:

$$T = N(1 + c\{\text{non-allocated local scalars} + \text{global scalars} + \text{non-scalar variables} + \text{RSR}\})$$

where c is the number of extra cycles required to access memory.

As we mentioned in the previous section, no traffic is taken into account for parameter passing. Figure 4.29 shows the speed-up obtained by Policies B-lf, A-live, A-lvOpt, and G-lf when 12 TBP registers are available with respect to a machine without TBP registers. The speed-up factor is shown for the two different memory speeds mentioned above. The maximum speed-up shown is obtained when the RSR traffic has been completely eliminated as well as the traffic caused by the return address. For instance, this would be the case when an infinite register file is provided with 13-register, fixed-size windows (12 for the TBP registers and one for the return address). From the figure, we conclude the following:

1. The speed-up factor is more significant for PCC than for GNU, because PCC does not assign any local scalar variables to registers when there are no TBP registers available. Once TBP registers become available, locals are assigned and, therefore, the data memory traffic is reduced and the program is executed more quickly. On the other hand, GNU assigns dead local scalars to TBD registers so that even when no TBP registers are available, the local scalar memory traffic is less for GNU than

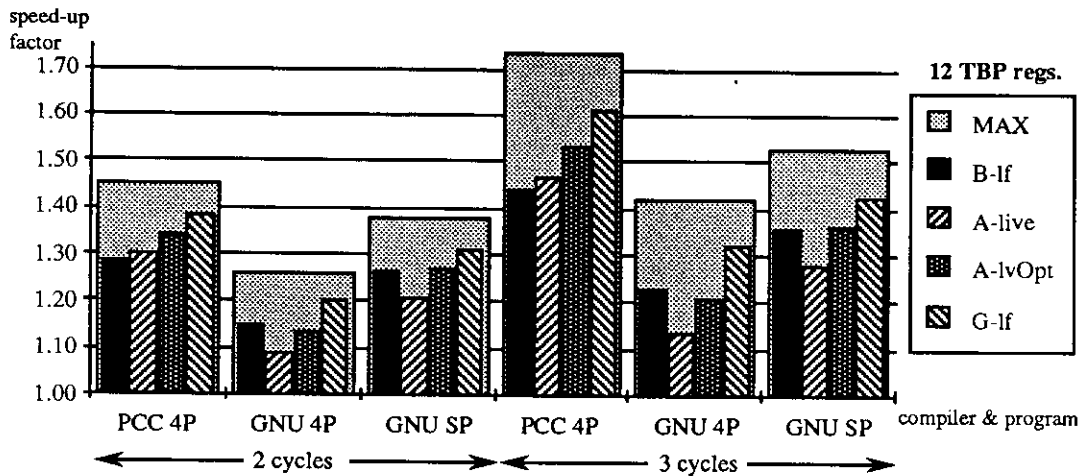


Figure 4.29: Speed-Up for Intra-Procedural Optimizations w.r.t. 0 TBP registers

for PCC (see Figure 4.24). Thus, the overall data memory traffic reduction is less for GNU when TBP registers become available and the speed-up factor is smaller. If GNU did not assign local scalar variables to TBD registers, then the speed-up factor obtained by GNU would be similar to the one obtained by PCC.

2. The dynamic Policy G-lf provides the higher speed-up factor and is the one which comes closer to the maximum possible speed that can be obtained.

The speed-up of Policy G-lf with respect to the static policies is better appreciated in Figure 4.30. As basis for comparison we have selected Policy B-lf since this is the static policy which generates the least traffic with GNU for the average of the four programs and for SPICE. When 2 cycles are required to access memory, Policy G-lf has a speed-up factor of between 1.04 (for GNU SP) and 1.08 (for PCC 4P) and when 3, between 1.05 (for GNU SP) and 1.12 (for PCC 4P).

Finally, let us consider the speed-up factor for a greater number of TBP registers. This is shown in Figure 4.31 for the four programs and for SPICE for 12, 16, and 24 TBP registers with respect to Policy B-lf with 12 TBP registers. From the figure, we conclude the following:

1. The speed-up factor for the static Policies A-live and A-lvOpt with respect to Policy B-lf becomes worse due to the increase in register saving/restoring traffic (see Figures 4.17 and 4.21).
2. The speed-up factor for the dynamic Policy G-lf increases with a larger register set due to the RSR traffic reduction. This increase is significant for SPICE (from 1.04 for 12 TBP registers to 1.08 for 24), but not for the four programs (from 1.05 to 1.06), because SPICE is the only program for which the overall data memory traffic becomes smaller for more than 12 TBP registers (see Figure 4.27).

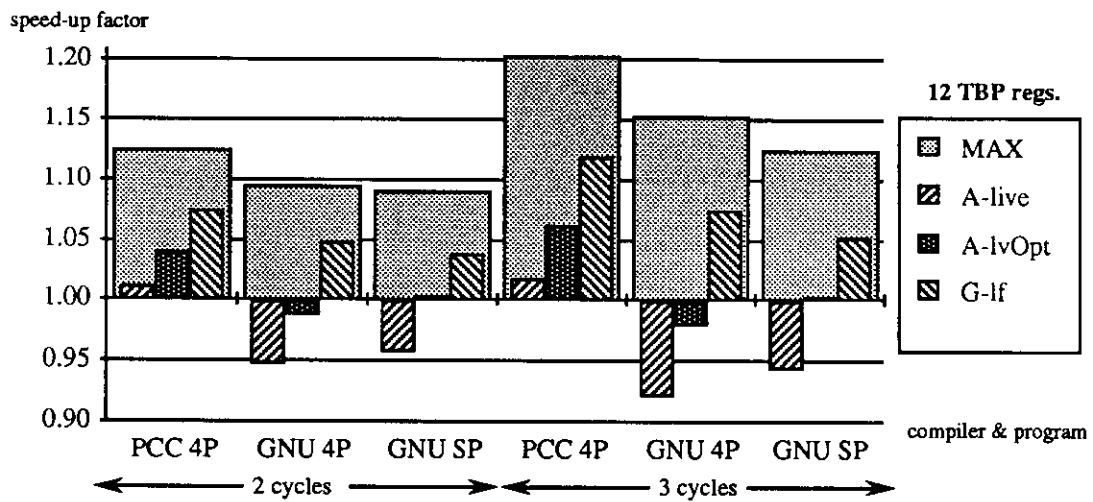


Figure 4.30: Speed-Up for Intra-Procedural Optimizations w.r.t. Policy B-lf

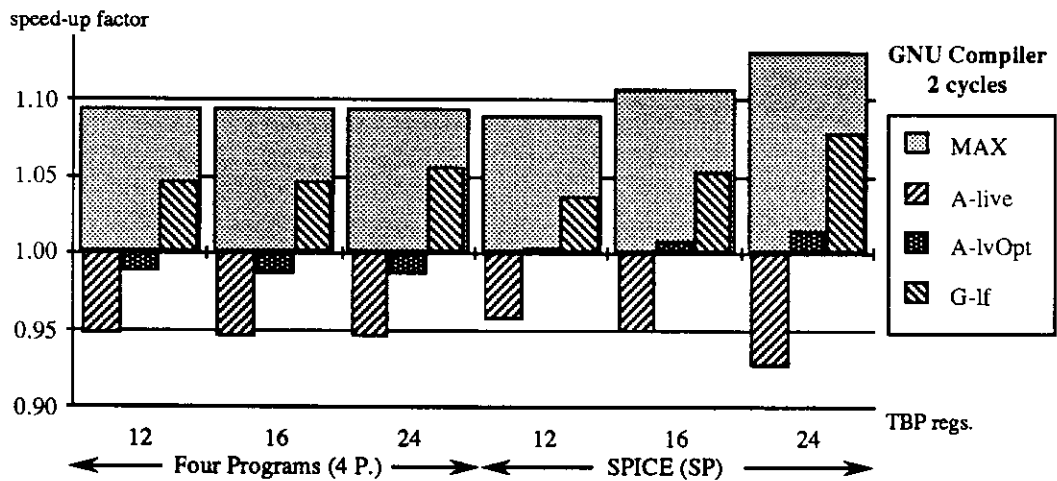


Figure 4.31: GNU Speed-Up for 12, 16, and 24 TBP registers w.r.t. Policy B-lf

Chapter 5

Inter-Procedural Optimizations

Chapter 5 shows how the remaining overall data memory traffic after performing *intra-procedural* compiler optimizations (see Chapter 4) is reduced even further with *inter-procedural* compiler optimizations. As in the previous chapter, our goal is to compare the RSR traffic reduction offered by these optimizations without architectural support (i.e., for the static policies) and with architectural support (i.e., the dynamic Policy G).

Figure 5.1 shows the remaining data memory traffic after performing intra-procedural optimizations for the average of the four programs (ASM, NROFF, SORT, and VPCC; on the top) and for SPICE¹ (on the bottom). The RSR policies used to compute the overall data memory traffic are the static Policy B-lf (on the left) and the dynamic Policy G-lf (on the right). Policy B-lf has been selected rather than either Policy A-live or A-lvOpt because it is the static policy that generates the least traffic for the average of the four programs and for SPICE with the GNU C Compiler (although not for every program and compiler; see Subsection 4.1.5 and Section 4.2). Also, the GNU C Compiler is the only compiler used in this chapter to evaluate the inter-procedural optimizations since it is the one that generates the least overall data memory traffic and that assigns double variables to registers (see Section 4.5). As we can see in the figure, scalar traffic still accounts from 51% to 61% of the overall data memory traffic.² These are the percentages of data memory traffic that we expect to reduce with our *inter-procedural optimizer*.

While an intra-procedural optimizer parses and generates code per function, the inter-procedural optimizer has a *global* view of the whole program (see Subsection 2.1.3.2). In this case, the compiler can assign some global scalar variables to registers (as discussed in Section 5.2) and perform some better register assignment policies for the return-address register (as discussed in Section 5.3) and for the intra-procedural allocated local scalar variables (as discussed in Section 5.4), so that the overall data memory traffic is reduced. In order to obtain this:

1. We determine the number of registers required for global scalar variables and eval-

¹See Section 4.3 for a discussion of why we have separated this program (SPICE) from the other four.

²Figure 5.1 is a copy of Figure 4.28. See Section 4.5 for a detailed discussion of the figure.

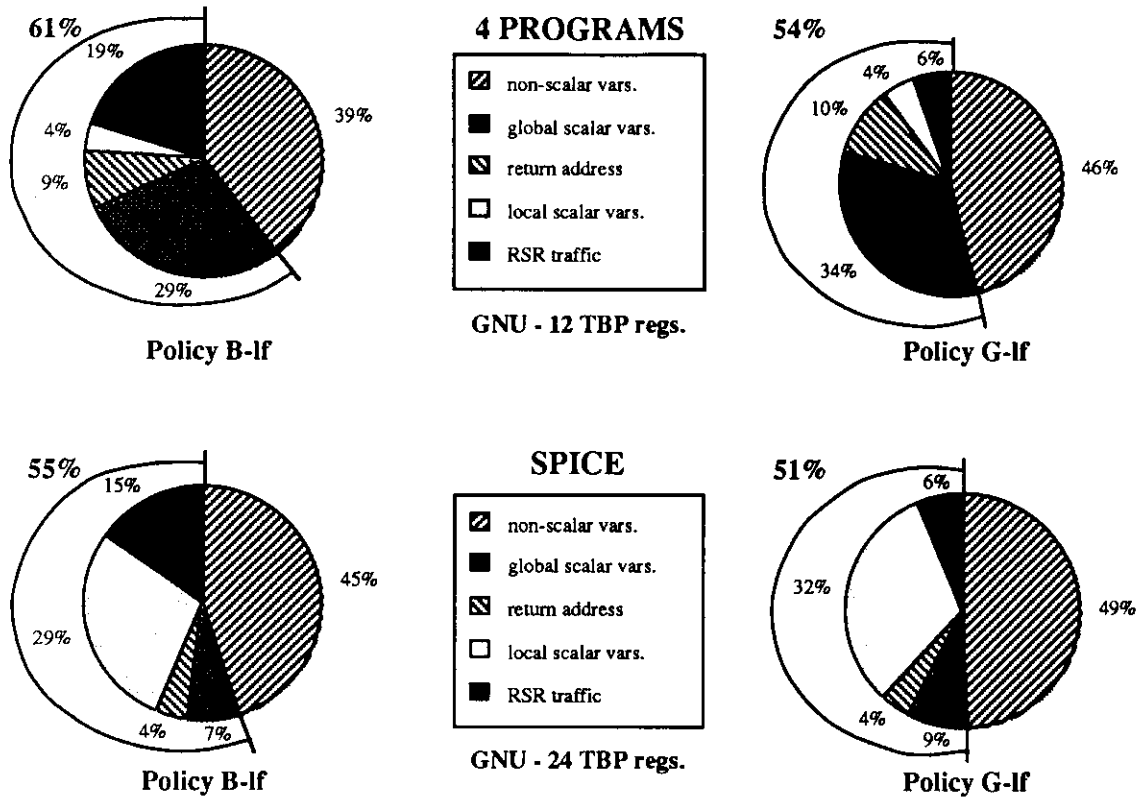


Figure 5.1: Scalar Data Memory Traffic To Be Eliminated

uate the data memory traffic reduction obtained. We show that most of the global scalar traffic is produced by a few variables and, therefore, the global scalar traffic is significantly reduced if these variables are assigned to registers rather than to memory. For instance, 8 registers can cut the global scalar traffic by 49% for the average of the four programs and by 44% for SPICE. More registers might be used to reduce the global scalar traffic further, as we will see in Section 5.2. However, since our goal is to have a 32-register file to avoid that its access time has a negative effect on the processor cycle time (see Subsection 2.1.1), we only mention the traffic reduction for a small number of registers.

Moreover, we also discuss how to select the global variables to be allocated to registers. We show that static information as used for intra-procedural allocators (see Section A.1) is not enough and that a *dynamic execution profile* of the program is necessary to obtain a more efficient register allocation. The use of dynamic information to perform some compiler optimizations has already been mentioned [Fish84, Elli86] and Wall [Wall86] has already used it for its inter-procedural register allocator (see Subsection 2.1.4).

2. We determine the number of registers required to keep the return address (RA) in a register, rather than storing it into memory when a function call is made, and we evaluate the data memory traffic reduction obtained. As we mentioned in

Subsection 4.1.3, this optimization is possible when the `call` and `return` instructions keep the return address in a register rather than saving/restoring it to/from memory. As for global scalar variables, we show that with a few registers (4) the traffic caused by the saving/restoring of the return address is cut in half for the average of the four programs and by 90% for SPICE, when a dynamic execution profile of the program is available.

3. We present new global register assignment policies to eliminate unnecessary register saving/restoring for the static Policies A-live, A-lvOpt, and B-lf (named A^g-live, A^g-lvOpt, and B^g-lf, respectively), and to obtain a better *disjoint register assignment* for the dynamic Policy G-lf (named G^g-lf). We show that, by using a dynamic execution profile of the program, each policy reduces its RSR traffic significantly. Moreover, while for an intra-procedural optimizer there is no reason for having more than 12 TBP registers for non-numeric applications (see Figures 4.26 and 4.27), this is not the case for the inter-procedural optimizer because more registers can be used to reduce the RSR traffic.

For the average of the four programs, we show that when a 32-register file is partitioned with 16 TBP registers, 8 registers for globals, and 2 registers for the return address, the inter-procedural optimizer eliminates 21% of the overall data memory traffic when both architectural and compiler support are provided (i.e., the dynamic Policy G as presented in Section 3.3) and 22% when only compiler support is available. Moreover, when both architectural and compiler support are provided, the overall data memory traffic generated is 86% of the one produced when only compiler support is available.

For SPICE, however, the data memory traffic is the same with and without architectural support, as we will discuss in Subsection 5.5.2. In this case, there is no benefit for providing the dynamic Policy G. In spite of this, we will show that the combination of both architectural and compiler support (i.e., Policy G^g-lf) provides the best approach to reduce the overall data memory traffic for most cases. Moreover, Policy G^g-lf simplifies the compilation process and reduces the data memory traffic independently of the characteristics of the programs (e.g., the percentage of recursive functions in the program).

No optimization is proposed for the non-scalar traffic because these have already been performed by the intra-procedural optimizer (common subexpression elimination, loop-invariant removal, etc.). We expect that since part of the data memory traffic has been eliminated, fewer conflicts will occur in the cache memory. Thus, the performance of the overall system will increase. However, we have left this as an open topic for discussion (see Section 6.2). Moreover, we have not looked neither for alternative register allocation schemes (e.g., as coloring) to reduce the local scalar traffic for SPICE (see also Section 6.2).

The outline for this chapter is the following: Section 5.1 presents our approach in how the inter-procedural optimizations should be implemented in the compilation process. Sections 5.2, 5.3, and 5.4 discuss the optimizations to be performed to reduce the global scalar traffic, the return-address traffic, and the register saving and restoring traffic,

respectively. Section 5.5 shows how the general-purpose register set should be partitioned to obtain the maximum overall traffic reduction. This section also presents the speed-up factor that we might obtain for a RISC-like machine. Readers who are not interested in the implementation details of the optimizer can go directly to Section 5.5. Finally, Section 5.6 summarizes the most important results obtained in this chapter.

5.1 The Inter-Procedural Optimizer

In this section we present how the inter-procedural optimizer should be incorporated in the compilation process and what information should be made available to it. Since C is a modular programming language and modules are compiled independently (see Figure 2.3), the following steps are taken to perform inter-procedural optimization:

1. The program is compiled with intra-procedural optimization. During compilation the following information is added to a common data base³:
 - Functions defined per module ($F = \{f_1, f_2, \dots, f_p\}$).
 - Variables allocated per function ($V^i = \{v_1^i, v_2^i, \dots, v_m^i\}, \forall f_i \in F$). We assume that all registers are of the same type and that there is a unique general-purpose register set with n TBP registers ($\{r_1, r_2, \dots, r_n\}$) for local scalar variables.
 - For each call instruction in f_i we keep the following:
 - The function that is called ($i \rightarrow j$ means that f_i calls f_j).
 - For Policy A^g-live, live-variable information at each call ($L_{i \rightarrow j}$).
 - For Policy A^g-lvOpt, the registers assigned by the intra-procedural allocator to be saved and restored at each call⁴ ($S_{i \rightarrow j}$ and $R_{i \rightarrow j}$).

Notice that, in a given function f_i , multiple calls to the same function f_j generate different $L_{i \rightarrow j}$, $S_{i \rightarrow j}$, and $R_{i \rightarrow j}$ sets even though this is not reflected in our notation.⁵

- Functions which are called indirectly through a pointer. When a call to a function is made through a pointer and no data-flow information is available to determine which set of functions might be called, we have to assume the worst case: any function might be called [Weih80, Coop86]. In this case, it is not possible to apply the static inter-procedural optimizations because no disjoint register assignment can be found (since all the functions might

³This information could also be collected by a high-level-language-parsing editor as the one proposed by [Coop86].

⁴We also assume that the intra-procedural assignment is performed sequentially. That is, if $V^i = \{v_1^i, v_2^i, \dots, v_m^i\}$ is the set of variables selected for allocation in function f_i and $\{r_1, r_2, \dots, r_n\}$ is the set of TBP registers, then variable v_1^i is assigned to register r_1 , v_2^i to r_2 , and so on. The reason for this assignment is given in Subsection 5.4.2.

⁵A subindex may be added to denote a different call for each $i \rightarrow j$; however, we do not do it to keep our notation simple.

be called). To limit the scope of indirect calls to a (expected small) set of functions, data-flow analysis must be performed to provide the functions which are called through a pointer.

- Global scalar variables defined (to perform global scalar variable allocation as discussed in Section 5.2).

Once all the user modules have been compiled, the source modules for the library functions, which are required by the program, are also compiled. The process of detecting which source modules need to be compiled for a given program and where these are stored should be done automatically (i.e., invisible to the user). Thus, a library for the sources should be kept in a similar way than the library for the objects [UNI81]. For each source module the data base is also updated with the corresponding information.

2. The executable program is run and profile information is collected and added to the data base. This can be performed as described for BKGGEN in Section 1.2. The profile information includes:
 - The frequency of execution for each call instruction ($Q_{i \rightarrow j}$). This is to estimate the RSR traffic generated by this call.
 - The frequency of usage for each global variable.

Section 5.2 shows why static information is not enough to perform inter-procedural optimizations and a dynamic execution profile is required.

3. Once the above information is available, the inter-procedural optimizer generates the program's call graph and performs register allocation and assignment for global scalar variables as described in Section 5.2, register assignment for registers with the return address as described in Section 5.3, and register assignment for TBP registers as described in Section 5.4. For Policies A^s-live, A^s-lvOpt, and B^s-lf, the registers which are saved and restored unnecessarily are detected (as discussed in Subsections 5.4.1, 5.4.2, and 5.4.3) and marked for optimization. For Policy G^s-lf the optimizer only needs to perform the disjoint register assignment since the unnecessary register saving/restoring is automatically removed by the dynamic behavior of Policy G (see Section 3.3).

The information on register assignment and unnecessary register saving/restoring computed by the inter-procedural optimizer is kept in the data base for the assembler as we can see in Figure 5.2. If we compare this figure with Figure 2.3, the main difference with the standard compilation process is that all modules must be compiled before being assembled.

4. Finally, the assembly modules are translated to object code in such a way that:
 - Register assignment is changed as defined by the inter-procedural optimizer.
 - Instructions that perform unnecessary register saving/restoring are removed.

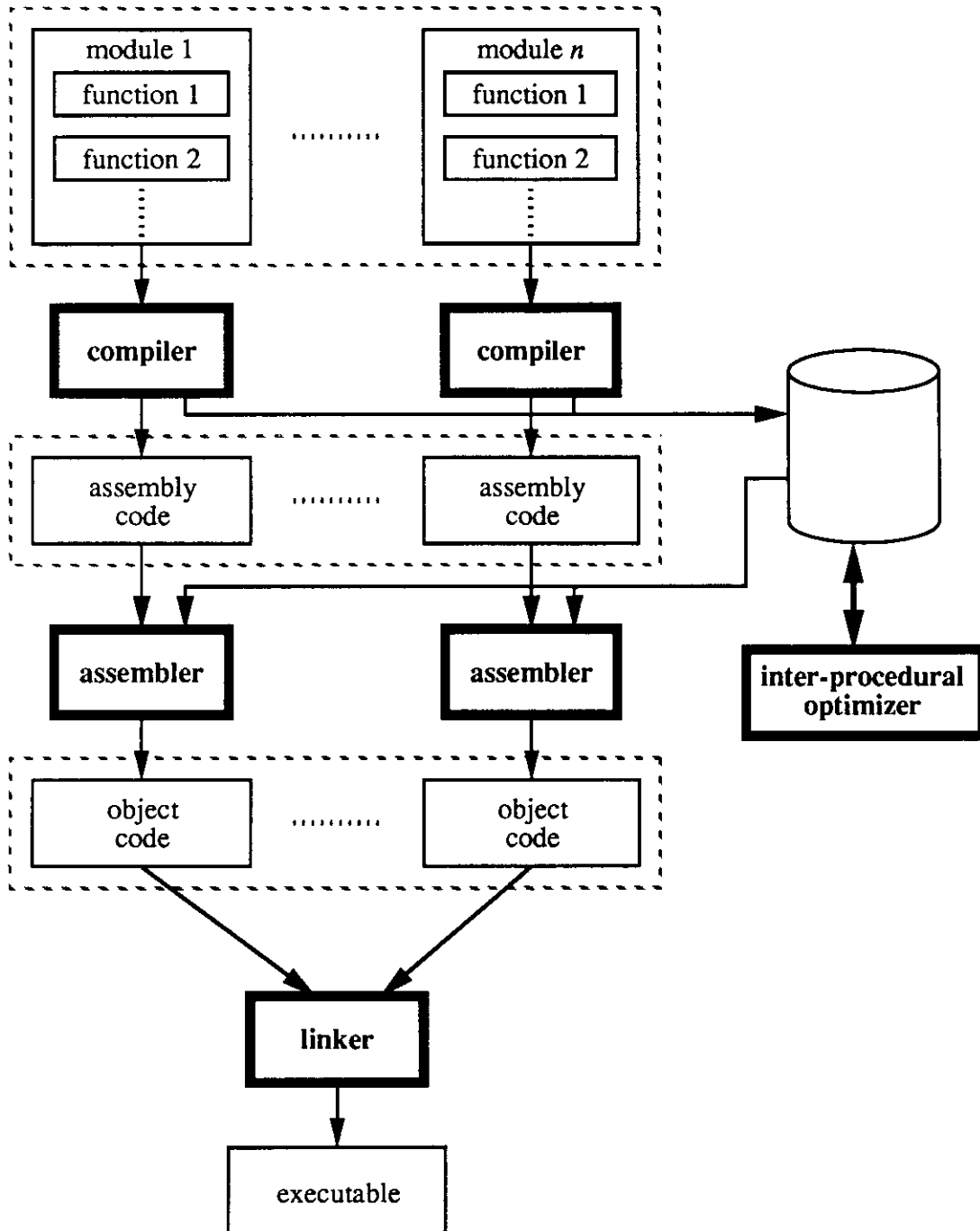


Figure 5.2: The Inter-Procedural Optimizer in the Compilation Process

Notice that the source modules do not need to be recompiled for inter-procedural optimization.

Although the inter-procedural optimization makes the compilation process more complex, it can be justified because of the reduction in the data memory traffic obtained.

5.2 Register Allocation for Global Scalar Variables

In this section we discuss the data memory traffic reduction obtained when an inter-procedural optimizer allocates some global scalar variables with no alias (see Subsection 2.1.3.1) to registers. Variables are assigned for the whole program so that no loading/storing traffic is generated. We show that:

1. A few registers are enough to obtain a significant reduction in the data memory traffic caused by the global scalar variables.
2. When a dynamic execution profile is used to select the variables to allocate, a larger reduction is obtained than when static information is used.

Notice that since the allocated variables are going to be assigned to registers during the whole program, the critical decision is the selection of which variables should be allocated. Once this decision has been made, the assignment of these variables to registers is straight forward. For this reason, in this section we use *register allocation* and *register assignment* in an interchangeable way.

The global scalar variables assigned to registers have been selected according to the following criteria:

Static: By static linear frequencies. The selected variables are the ones which have been more frequently (statically) referred in the program. No weight is given to the variables accessed inside loops. This is the policy used by the GNU and ACK intra-procedural allocators (see Section A.1).

Profile: A dynamic execution profile has been obtained using three different inputs to each program (as indicated in Table 5.1). Also, an average execution profile has been computed from these three execution profiles. The advantage of using an average profile rather than any of individual profiles is discussed afterwards. The average profile has been calculated so that each execution profile is equally weighted. These are labeled *prof. 1*, *prof. 2*, *prof. 3*, and *prof. avg.*, respectively.

Optimal: By dynamic execution frequencies. The execution frequencies correspond to the specific execution of the program for the input data given in Section 1.3. Since the more frequently used global scalar variables are assigned to registers, this is the optimal criterion for their selection. This criterion is similar to the Belady's optimal

program	prof. 1	prof. 2	prof. 3
VPCC	<i>trees.c</i> —a module from the Portable C Compiler (1700 source lines)	<i>n1.c</i> —a module from the <i>nroff</i> word processor (1000 source lines)	<i>sort.c</i> —the <i>sort</i> program (900 source lines)
ASM	<i>trees.s</i> —the assembly code for <i>trees.c</i>	<i>n1.s</i> —the assembly code for <i>n1.c</i>	<i>sort.s</i> —the assembly code for <i>sort.c</i>
SORT	sort a 90-entry password file by user ID	sort 2250 4-field records with the options <i>+1 -2</i>	sort 600 6-field records with the options <i>+3 -t -r</i>
NROFF	format the article [Hugu87]	format the manual page for <i>csh</i>	format a set of problems with equations
SPICE	<i>bipole, digsr, toronto</i> —three circuits published in [John87] to use SPICE as a floating-point benchmark		

Table 5.1: Input Data for Profiling the Measured Programs

page replacement algorithm [Bela66] (because the “future” program behavior is known in advance) and it is used to evaluate the performance of the above criteria.

Table 5.2 shows the global scalar traffic remaining for each program when 1, 2, 4, 8, 16, or 32 global scalar variables are assigned to the corresponding number of registers. The traffic is normalized with respect to the global scalar traffic generated when no global scalar variables (*gsv*) are assigned to registers. From the table we conclude the following:

1. When profile information is used to allocate global scalars (for any of the three sets of input data discussed above), the traffic generated is always less than the traffic produced when static information is used (see also Figures 5.3 and 5.4). Therefore, the selection of which global scalar variables should be assigned to registers cannot be based on static linear priority, as it is done for intra-procedural register allocation (see Section A.1), because static information does not provide reliable information about which global scalars are the most frequently used during program execution.
2. Since the program is expected to run under different input data, it is difficult to find a single input data to be representative for all program executions. For this reason, we selected an average of the three profiles obtained as the criterion to allocate global scalar variables. We expect that different runs activate alternative program paths so that, on the average, the register allocation performed benefits different programs runs, not a single type. As we can see in the table and in Figures 5.3 and 5.4, the selection performed with the average profile is very close to the optimal for almost every program and any number of registers. The exceptions are NROFF

no. regs.	register allocation	ASM	NROFF	SORT	VPCC	4 P.	SPICE
0	(gsv traffic)	1 (6.02)	1 (7.25)	1 (1.18)	1 (1.54)	1 (4.73)	1 (3.11)
1	static	0.87	1.00	1.00	0.63	0.96	1.00
	prof. 1	0.85	0.99	0.84	0.63	0.95	0.89
	prof. 2	0.85	0.89	0.84	0.63	0.86	0.89
	prof. 3	0.85	0.89	0.84	0.63	0.86	0.89
	prof. avg.	0.85	0.89	0.84	0.63	0.86	0.89
	optimal	0.85	0.89	0.67	0.63	0.86	0.89
2	static	0.82	0.98	0.84	0.63	0.94	0.98
	prof. 1	0.73	0.97	0.76	0.54	0.92	0.83
	prof. 2	0.73	0.82	0.68	0.54	0.79	0.82
	prof. 3	0.73	0.82	0.68	0.54	0.79	0.82
	prof. avg.	0.73	0.81	0.68	0.54	0.78	0.82
	optimal	0.73	0.81	0.51	0.54	0.77	0.82
4	static	0.77	0.92	0.84	0.62	0.89	0.94
	prof. 1	0.59	0.85	0.27	0.41	0.77	0.72
	prof. 2	0.61	0.69	0.27	0.42	0.64	0.70
	prof. 3	0.59	0.69	0.27	0.41	0.64	0.70
	prof. avg.	0.59	0.69	0.27	0.41	0.64	0.70
	optimal	0.59	0.69	0.18	0.41	0.64	0.70
8	static	0.53	0.86	0.41	0.47	0.79	0.94
	prof. 1	0.43	0.77	0.05	0.30	0.68	0.56
	prof. 2	0.43	0.60	0.06	0.30	0.54	0.56
	prof. 3	0.41	0.51	0.04	0.33	0.47	0.56
	prof. avg.	0.43	0.56	0.06	0.30	0.51	0.56
	optimal	0.41	0.51	0.04	0.30	0.46	0.56
16	static	0.44	0.71	0.04	0.43	0.64	0.93
	prof. 1	0.21	0.57	0.01	0.17	0.50	0.31
	prof. 2	0.20	0.42	0.00	0.18	0.37	0.30
	prof. 3	0.23	0.31	0.00	0.19	0.28	0.30
	prof. avg.	0.21	0.42	0.00	0.18	0.37	0.30
	optimal	0.20	0.31	0.00	0.17	0.28	0.30
32	static	0.31	0.48	0.00	0.29	0.43	0.93
	prof. 1	0.05	0.28	0.00	0.07	0.24	0.03
	prof. 2	0.04	0.21	0.00	0.08	0.18	0.02
	prof. 3	0.05	0.13	0.00	0.09	0.12	0.03
	prof. avg.	0.05	0.17	0.00	0.07	0.15	0.02
	optimal	0.04	0.13	0.00	0.06	0.11	0.02

Table 5.2: Global Scalar Traffic Reduction

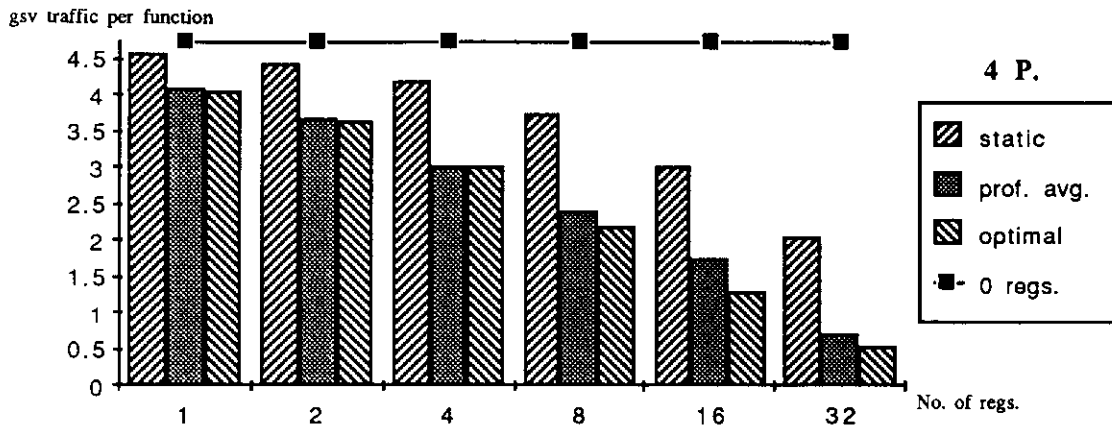


Figure 5.3: Global Scalar Traffic for the Four Programs

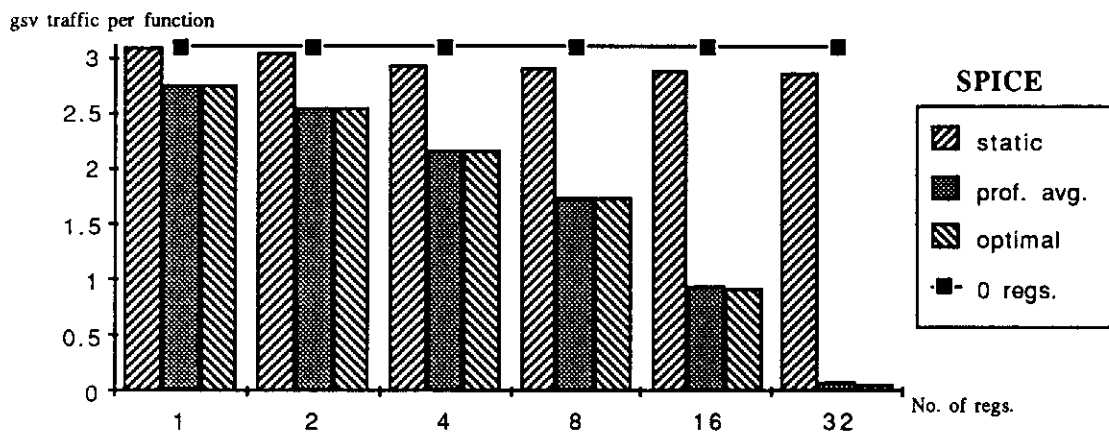


Figure 5.4: Global Scalar Traffic for SPICE

with 8 or more registers and SORT with 4 or less registers.

- Even though the number of global scalars is usually large (39 for SORT, 119 for ASM, 138 for VPCC, 237 for NROFF, and 307 for SPICE), there are only a few globals which generate most of the traffic. For instance, 50% of the global scalar traffic is generated by 2 variables for SORT and VPCC, 6 for ASM, 8 for NROFF, and 10 for SPICE. Therefore, a few registers are sufficient to significantly reduce the global scalar traffic. As we can see in the table, more registers might be used to reduce the global scalar traffic further. However, since our goal is to have a small register file, we have only mentioned the traffic reduction for a small number of registers for global variables.

Finally, one more consideration on the number of registers to be used for global scalar variable allocation. Since each program has a different data memory traffic distribution,

it is convenient not to have a fixed partition of the general-purpose register set (i.e., a fixed number of registers for global scalar variables). In this case, the inter-procedural optimizer can distribute the general-purpose registers available between the registers required for global scalars and the ones required by the optimizations discussed in the next sections so that the least data memory traffic is generated. For instance, 44% of the data memory traffic generated by NROFF corresponds to global scalars and 28% to local scalars. On the other hand, for VPCC these percentages are 5% and 51%, respectively. Thus, the optimizer could use more registers for globals for NROFF and fewer for VPCC. Let us postpone this discussion until we have commented on the other two inter-procedural optimizations and come back to it afterwards (in Section 5.5).

In conclusion, a dynamic execution profile has to be used to obtain a good allocation for global scalars. In this case, a few registers are sufficient to significantly reduce the global scalar traffic. For instance, 8 registers can cut the global scalar traffic by 49% for the average of the four programs and by 44% for SPICE. On the other hand, if only static information is used, the traffic reduction will be 21% and 6%, respectively. More registers can be used to reduce the traffic even further. However, we have to consider that the total number of general-purpose registers available is limited and it might be more convenient to use them to reduce the return-address traffic or the register saving/restoring traffic as we will discuss in Section 5.5 (once the inter-procedural optimizations to reduce the return-address traffic and the RSR traffic will have been presented in Sections 5.3 and 5.4).

5.3 Return-Address Inter-Procedural Optimization

In Subsection 4.1.3 an intra-procedural optimization to reduce the traffic caused by the saving/restoring of the return address (RA) was mentioned. To perform this optimization, call and return instructions are required to keep the return address in a register (called the *link* register) rather than storing it in memory by the call instruction itself. In this case, non-leaf functions save the link register to memory at function entry and restore it before return, but leaf functions do not have to perform this saving/restoring since the link register is not needed by the leaf function (see Figure 4.4).

An inter-procedural optimizer can have multiple link registers available so that the RA saving/restoring is eliminated not only for leaf functions, but also for non-leaf ones. In this section we first discuss how the RA traffic elimination is performed by an inter-procedural optimizer; second, we mention the effect of recursion and indirect calls on the call graph; third, we evaluate the data memory traffic reduction obtained by different numbers of link registers; and, finally, we compare this optimization with the alternative approach of having architectural support (i.e., multiple PCs) to reduce the RA traffic.

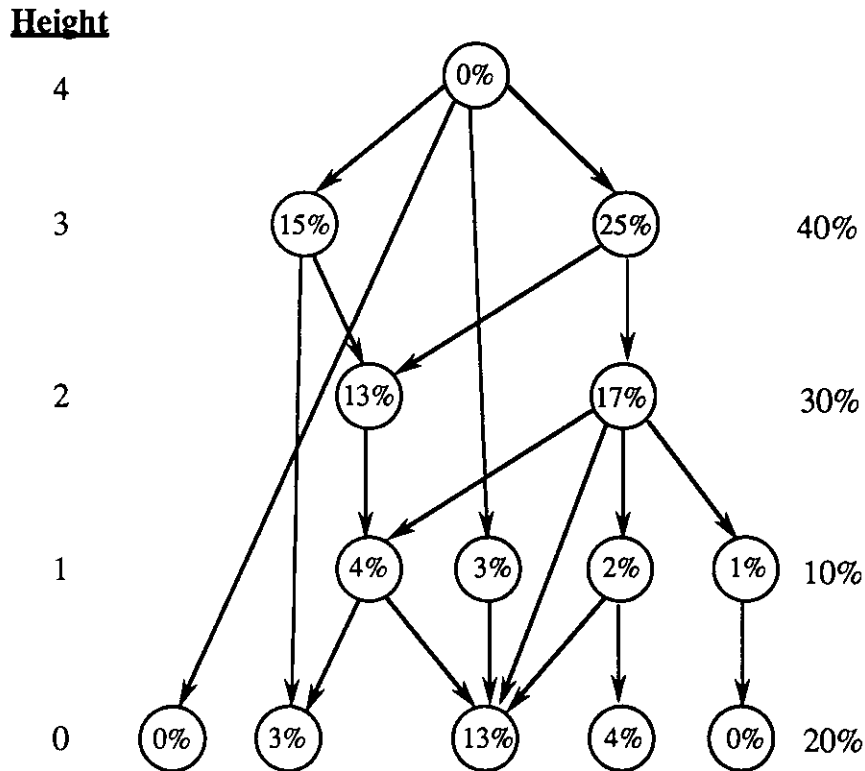


Figure 5.5: Function Execution Frequency in a Call Graph

An Example of RA Traffic Elimination

Figure 5.5 shows the call graph for a program,⁶ where the nodes represent the functions and the directed arcs, the calls. The percentages inside each node correspond to the function execution frequencies. Functions in the call graph have been grouped according to their *height*, that is, the number of descendents (or callees) in the longest path from this function to a leaf. Notice that all leaf functions have zero height. For this call graph, if only one link register is available, 20% of the RA traffic is eliminated since this is the percentage of leaf functions executed. This is the traffic reduction obtained by an intra-procedural optimizer as we discussed in Subsection 4.1.3.

When more link registers are available, an inter-procedural optimizer can use them for the non-leaf functions. In this case, the RA saving/restoring traffic can be eliminated for those functions for which their link register is not used by any of the function descendents. Notice the following:

- All the callers of a given function must use the same link register.

⁶Notice that the call graph is acyclic; that is, there are no recursive functions. Let us assume for this moment that this is the case; afterwards, we will discuss what happens with recursive functions and indirect calls.

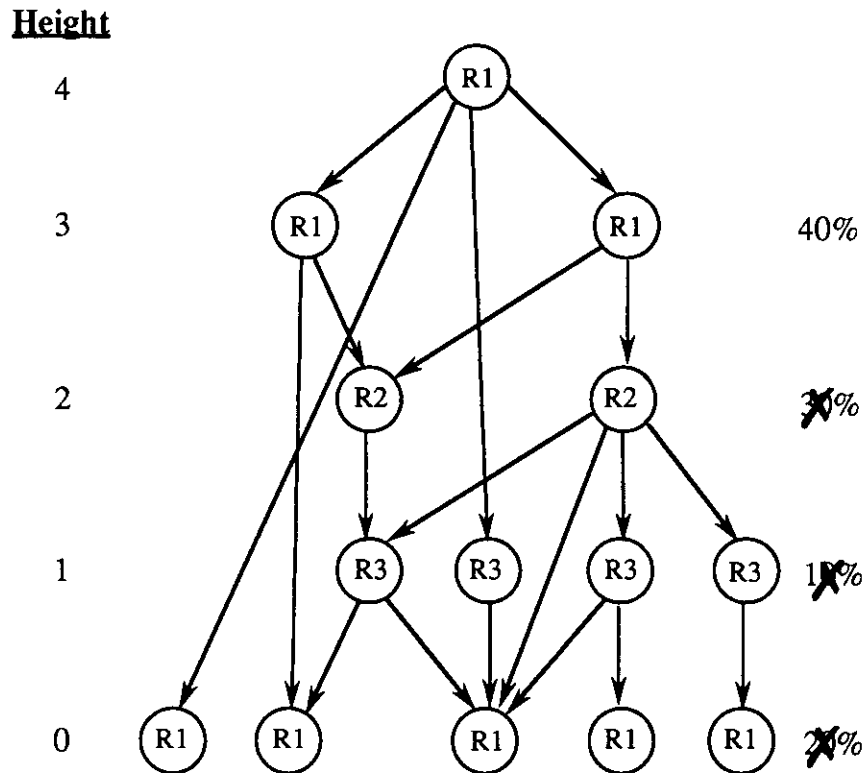


Figure 5.6: Register Assignment with Three-Link Registers

- Functions at the same height can share the same link register because they are not simultaneously active.

For instance, Figure 5.6 shows the assignment of three link registers to the functions at the bottom of the call graph. In this example, 60% ($= 30\% + 10\% + 20\%$) of the RA traffic is eliminated. The functions at height 3 and up could use any of the three link registers available. Since for these functions the link register has always to be saved/restored, only one link register is used for all of them (*R1* in this example).

Not all the link registers need to be assigned necessarily to the functions at the bottom of the call graph. That is, the RA memory traffic can be eliminated for all the functions at any given height whenever their descendents do not use the same link register. By obtaining a dynamic execution profile (as discussed in Section 5.2), it is possible to select the group of functions, at a given height, which are the most frequently used for the profiled input data and assign to them the additional link registers. For instance, with the register assignment shown in Figure 5.7, the RA traffic is reduced by 90% ($= 40\% + 30\% + 20\%$) in this example. Notice that in this case, functions at height 1 must use register *R1* as their link register.

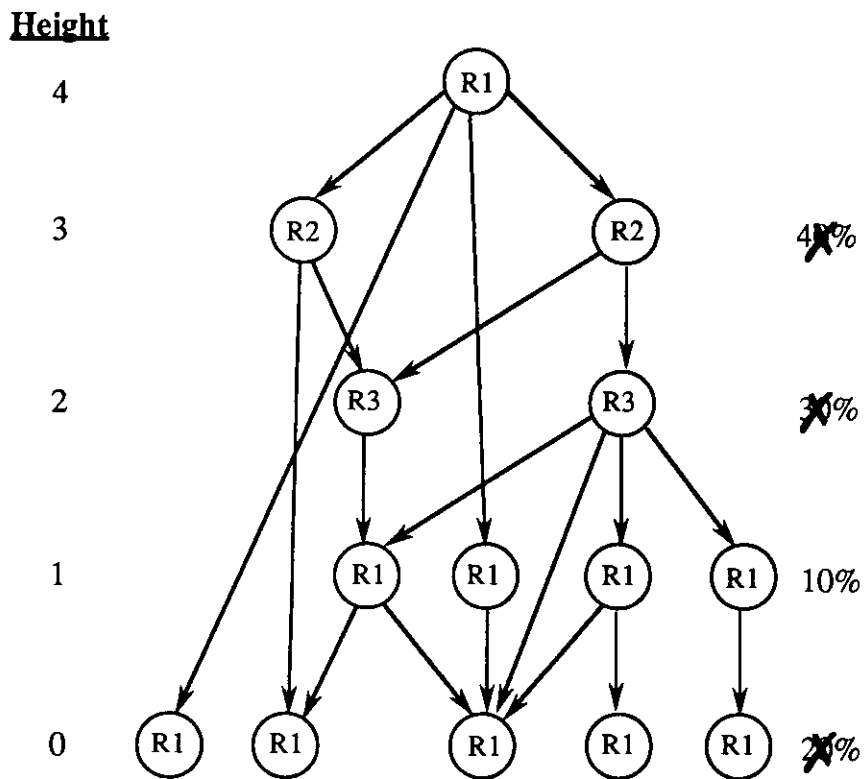


Figure 5.7: A Better Register Assignment with Three-Link Registers

program	% ftns.
ASM	2.8%
NROFF	49.6%
SORT	1.1%
VPCC	34.5%
4 P.	34.6%
SPICE	0.1%

Table 5.3: Percentage of Functions in a Cycle

Recursion and Indirect Calls

In the above example the call graph was acyclic. This is not the case when the program has *recursive* functions. For recursive functions, no RA traffic can be eliminated for the functions inside the cycle caused by the recursion, because the same link register is used by the function (or functions) which is (are) called recursively. Thus, the link register has always to be saved/restored. Table 5.3 shows the percentage of functions inside a cycle for the measured programs. As we can see, for some programs (SPICE, SORT, and ASM) the percentage of functions in a cycle is small and they do not have any impact in the RA traffic reduction which might be obtained. However, this is not the case for NROFF and VPCC. For NROFF, even if an infinite pool of link registers were provided, the RA traffic could be cut only by half and for VPCC, by one third. To reduce the RSR traffic for this type of programs, architectural support should be provided, as we will discuss at the end of this section.

If the program has *indirect* calls, the optimizer needs to perform *data-flow analysis* to determine the functions which can be reached from this indirect call. All these functions must use the same link register independently of their height because they will be called from the same call instruction. Since the number of indirect calls for the measured programs is small, this has not been taken into account for the measurements discussed next.

RA Memory Traffic Reduction

Two different register allocation approaches have been evaluated to reduce the RA traffic. These are similar to the ones used for global scalar variables (see Section 5.2):

Prof. Avg.: This corresponds to the average profile also used for the allocation of global scalars.

Optimal: This corresponds to the optimal allocation.

A static allocation has not been evaluated (as we did for the global scalar variables) because the most frequent called functions are not at the bottom of the call graph:

- For the four programs, Figure 5.8 shows the distribution of the calls to functions in the call graph. The call graph of each program has been divided into four regions such that each region includes one fourth of the maximum height. Region I includes the *main* function and Region IV includes the leaf functions with height zero. As we can see, only SORT has 79% of its executed functions in the last region. VPCC and NROFF have more than half of their executed functions in the third region while ASM has them in the second.
- For SPICE (not shown in the figure), 89% of the executed functions have height two or less (out of a maximum height of 37).

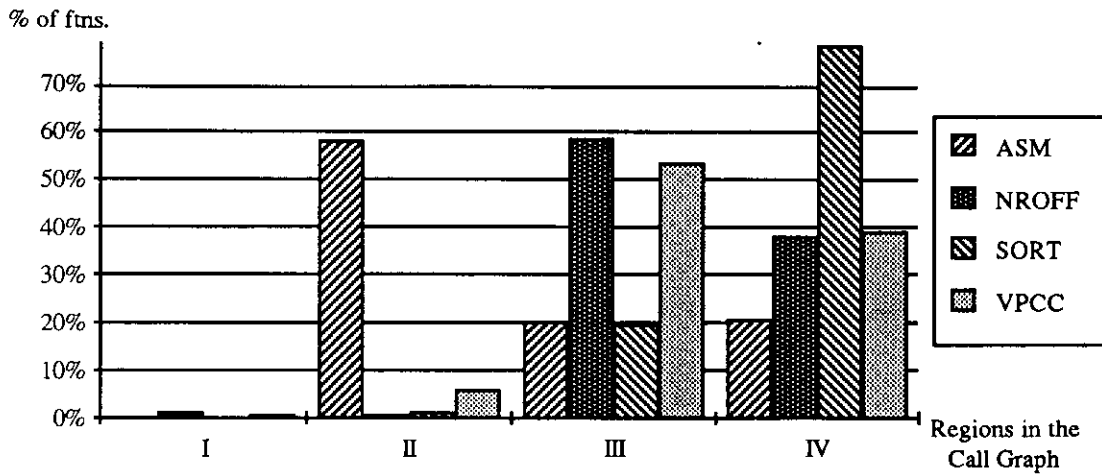


Figure 5.8: Distribution of Functions by Height

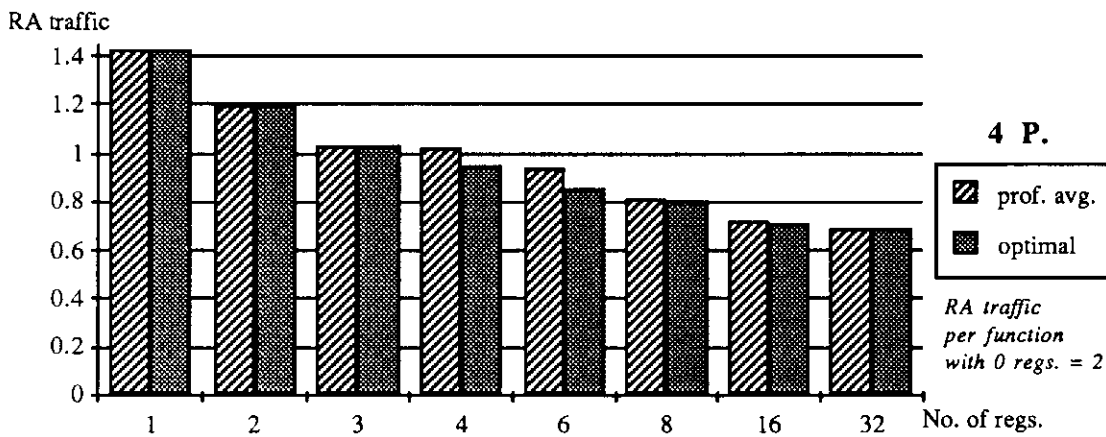


Figure 5.9: RA Traffic Reduction for the Four Programs

Since a selection based on static information will benefit only programs whose functions are concentrated at the bottom of the call graph (i.e., SORT and SPICE), we have only used dynamic profile information because a better selection is obtained for programs whose traffic is evenly distributed throughout the call graph as well as for those whose traffic is concentrated at the bottom.

Table 5.4 shows the RA traffic reduction obtained by each program for the two selection mechanisms mentioned above and for different number of registers. From the table we conclude the following:

1. In general, the register allocation based on the average profile is close to the optimal. On the average for the four programs, the RA traffic generated by the average profile allocation is from 100% to 110% of the optimal traffic (see also Figure 5.9).

no. regs.	register allocation	ASM	NROFF	SORT	VPCC	4 P.	SPICE
0	(RA traffic)	1 (2.0)	1 (2.0)	1 (2.0)	1 (2.0)	1 (2.0)	1 (2.0)
1	prof. avg.	0.81	0.85	0.22	0.82	0.71	0.56
	optimal	0.81	0.85	0.22	0.82	0.71	0.56
2	prof. avg.	0.75	0.73	0.22	0.65	0.60	0.44
	optimal	0.39	0.71	0.03	0.65	0.60	0.24
3	prof. avg.	0.72	0.59	0.03	0.62	0.52	0.43
	optimal	0.27	0.59	0.01	0.55	0.52	0.11
4	prof. avg.	0.61	0.57	0.01	0.58	0.51	0.10
	optimal	0.21	0.57	0.01	0.52	0.47	0.08
5	prof. avg.	0.19	0.55	0.01	0.48	0.49	0.08
	optimal	0.16	0.55	0.01	0.48	0.45	0.06
6	prof. avg.	0.16	0.55	0.01	0.47	0.47	0.05
	optimal	0.12	0.53	0.01	0.45	0.43	0.03
7	prof. avg.	0.16	0.54	0.01	0.47	0.45	0.02
	optimal	0.09	0.53	0.01	0.42	0.41	0.02
8	prof. avg.	0.14	0.54	0.01	0.45	0.41	0.02
	optimal	0.06	0.52	0.01	0.40	0.40	0.01
10	prof. avg.	0.08	0.53	0.01	0.45	0.40	0.01
	optimal	0.03	0.51	0.01	0.37	0.38	0.00
12	prof. avg.	0.04	0.52	0.01	0.42	0.38	0.00
	optimal	0.02	0.50	0.01	0.36	0.37	0.00
14	prof. avg.	0.04	0.51	0.01	0.42	0.37	0.00
	optimal	0.02	0.50	0.01	0.35	0.36	0.00
16	prof. avg.	0.04	0.51	0.01	0.41	0.36	0.00
	optimal	0.02	0.50	0.01	0.35	0.36	0.00
20	prof. avg.	0.02	0.50	0.01	0.39	0.35	0.00
	optimal	0.02	0.50	0.01	0.35	0.35	0.00
24	prof. avg.	0.02	0.50	0.01	0.36	0.35	0.00
	optimal	0.02	0.50	0.01	0.35	0.35	0.00
28	prof. avg.	0.02	0.50	0.01	0.35	0.35	0.00
	optimal	0.02	0.50	0.01	0.35	0.35	0.00
32	prof. avg.	0.02	0.50	0.01	0.35	0.35	0.00
	optimal	0.02	0.50	0.01	0.35	0.35	0.00

Table 5.4: RA Traffic Reduction with Inter-Procedural Optimization

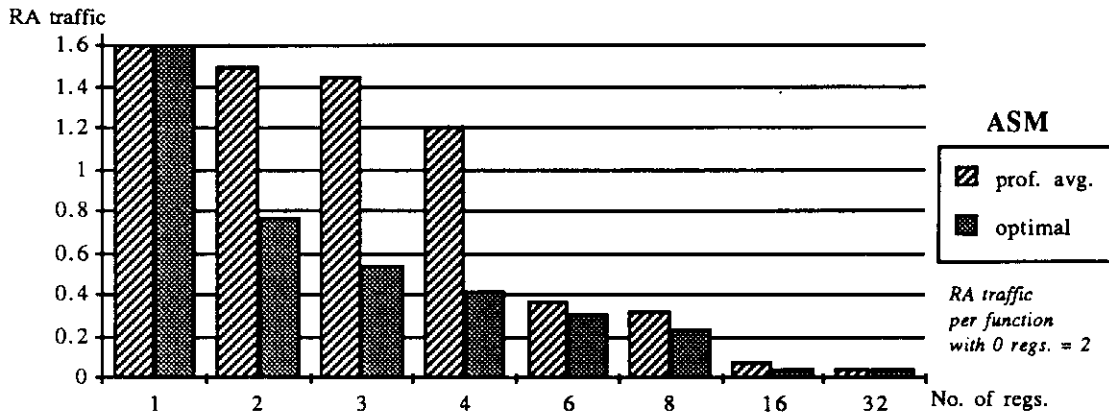


Figure 5.10: RA Traffic Reduction for ASM

2. ASM is the exception to the previous rule, with a significant difference when a small number of registers are available (between 2 and 4) as we can see in Figure 5.10. The 2, 3, and 4 most frequent executed variables do not coincide with the most frequent variables used in the execution profile. However, this is not the case when at least the 6 most frequent variables are considered.

3. The number of registers required to eliminate most of the RA traffic is usually small for non-recursive programs. For instance, with four registers 90% of the RA traffic is eliminated for SPICE, 99% for SORT, and about 60% for the other programs.

Notice that for NROFF and VPCC, a significant increase in the register set size does not correspond with a significant reduction in the RA traffic. For instance, doubling the number of registers from four to eight, the RA traffic eliminated increases from 43% to 46% for NROFF (out of a 50% maximum reduction) and from 48% to 55% for VPCC (out of 65%).

Compiler versus Architectural Support

The RA traffic reduction provided by the optimizer depends on the characteristics of the program. These program's characteristics are:

1. The percentage of functions in cycles. As we said above, no RA traffic can be eliminated for these functions. For instance, half of the executed functions in NROFF are in a cycle. This implies that in the optimal case, only half of the RA traffic can be eliminated.
2. The distribution of the functions by height. If the most frequently executed functions are concentrated in a small set of heights for any possible program execution, then a few registers are enough to reduce the RA traffic significantly. Otherwise, a

no. regs.	NROFF	SORT	VPCC	3 P.
0	1 (2.0)	1 (2.0)	1 (2.0)	1 (2.0)
4	0.36	0.00	0.24	0.30
8	0.04	0.00	0.06	0.04
16	0.00	0.00	0.00	0.00

Table 5.5: RA Traffic Reduction with Multiple PCs

larger register set (between 8 and 16 registers) is necessary to obtain a significant traffic reduction (90% or more). For instance, this is the case for ASM as we can see in Figure 5.10.

To overcome these constraints a register file with multiple PCs can be provided. The multiple PCs are handled as a multiple-window register file with a register window of size one (see Subsection 2.1.1). Thus, no memory traffic is generated while there are no overflows or underflows.

Table 5.5 shows the RA traffic reduction with multiple PCs. The traffic has been computed from our previous measurements [Hugu85a, Table 3.5]. As we mentioned in Section 1.3, these only included NROFF, SORT, and VPCC. For this reason, Table 5.5 only shows these programs and not ASM and SPICE. Since we expect similar results for these two programs, we have not repeated the measurements.

If we compare Table 5.5 with Table 5.4, the RA traffic is much smaller for any register-set configuration when architectural support is provided. However, data memory traffic is not the only parameter to be considered in order to compare both alternatives. The cost of processing an overflow (or underflow) condition has also to be taken into account. The cost is defined by these parameters:

1. Processor interrupts. An overflow condition stops the current processor execution flow and control is passed to a function handler to process the exception. On the other hand, when the optimizer has not been able to eliminate a RA saving/restoring instruction, a memory transfer occurs, but the processor is not interrupted.
2. Stack for overflows/underflows. Multiple PCs require a separate stack to store them as it was discussed for multiple-window architectures (see Subsection 2.1.1 and Figure 2.1). On the other hand, when compiler support is provided, the activation record of the function is used to save/restore the RA when it is necessary. Thus, the programming environment is more complex for multiple PCs.
3. Dedicated resources. The register file provided for multiple PCs can only be used to store the return addresses.⁷ The compiler cannot use these resources to reduce

⁷This is the case with a standard multiple-window implementation. An alternative approach where the multiple PCs are part of the general-purpose register file would be possible to implement. The

other traffic when they are not needed to reduce the RA traffic of certain programs (like SPICE and SORT).

Since the cost of processing each overflow (or underflow) condition is higher than a single memory transfer, the number of registers provided should be large enough to minimize the number of overflows/underflows. In this case, for a larger register file (8 or 16 registers), the performance of both approaches is similar for programs without a significant number of recursive functions. Therefore, the compiler can make a better use of a general-purpose register file depending on the program's characteristics (with the exception of recursivity).

In conclusion, the tradeoff between selecting architectural support and compiler support is that for the former, the RA traffic is almost eliminated if at least an 8-register file is provided. On the other hand, the compiler makes a better use of the general-purpose registers available and, for programs without recursion, the compiler reduces significantly the RA traffic with a few registers. For instance, with four registers 90% of the RA traffic is eliminated for SPICE, 99% for SORT, and about 60% for the other programs.

5.4 RSR Inter-Procedural Optimizations

To reduce the RSR traffic even further, four inter-procedural optimizations are proposed. These optimizations perform the register assignment for the variables selected by an intra-procedural register allocator. Since at this phase the program call graph is known, registers can be assigned in such a way that the RSR traffic is reduced. For instance, for Policy A-live, the optimizer can eliminate, in a specific call, the saving/restoring instructions for the registers that are not used by any of the functions that might be called from this point. Although the goal (to reduce the RSR traffic) and the method (disjoint register assignment) is the same for each optimization, the algorithms themselves are different for each policy: A-live (renamed A^g -live), A-lvOpt (renamed A^g -lvOpt), B-lf (renamed B^g -lf), and G-lf (renamed G^g -lf). The reason for selecting both Policies A-live and A-lvOpt as candidates for inter-procedural optimization is that Policy A-lvOpt can only be partially optimized as we will explain in Subsection 5.4.2. Thus, we have to compare both optimized policies to verify whether Policy A^g -lvOpt generates less RSR traffic than Policy A^g -live, as it is the case for Policy A-lvOpt with respect to A-live.

Three current inter-procedural register allocators [Wall86, Stee87, Chow88] have been reviewed in Subsection 2.1.3.5. Our approach is different than the one used by Wall; he allocates as many local and global scalar variables as possible to registers in such a way that no register saving/restoring is generated. When no more registers are available, the variables are allocated to memory. Consequently, it might be that the traffic generated

compiler could reserve a few general-purpose registers to be used as multiple PCs. The first and the last register to be used by the architecture should be indicated in specialized registers initialized when the program execution starts. In this case, the drawback of having dedicated resources would be eliminated.

by the non-allocated variables is greater than the traffic that would be generated if some registers were saved/restored to free registers for these variables (see Section 6.2). Here we still assume that register allocation is performed per function and we want to optimize register assignment to reduce the RSR traffic. This is also the approach taken by Steenkiste and by Chow. Their approach is based on Policy A-live. We discuss the differences between our approaches in Subsection 5.4.1. No comparison is made with Wall's register allocator because for small register sets he evaluates his allocator with speed-up factors rather than memory traffic reduction.

5.4.1 Global Policy A-live

The Global Policy A-live (**A~~g~~-live**) eliminates the register saving/restoring instructions performed at a given call when the registers being saved/restored are not used by any of the functions that might be called from this point. Thus, the optimizer needs to find a disjoint register assignment for the functions in the same path in the call graph to avoid as much traffic as possible. The problem of finding an optimal register assignment to generate the minimum RSR traffic is NP-complete [Chai81]. Here we present a heuristic solution to this problem. First, we discuss the approach used to solve the problem; our approach is presented together with an example to show how the registers are assigned and how the RSR traffic is eliminated. Second, we introduce an algorithm to solve the problem when the program does not have recursive functions. Third, we explain how to handle recursive functions in a program. Fourth, we compare our inter-procedural register assignment approach with Steenkiste's and Chow's inter-procedural register allocators. Finally, we show the RSR traffic reduction obtained with our approach.

Our Approach and an Example of RSR Traffic Elimination

We start assigning variables to registers for the leaf functions. Since leaf functions are never going to be active simultaneously, they can share the same set of registers without having to save/restore them [Wall86, Stee87, Chow88]. Next step is to select a function at height 1 (i.e., a function such that all its descendents are leaf functions) and to find a register assignment for the pre-allocated variables of this function that minimizes the RSR traffic, taking into account both the live variables in each call and the registers already assigned to the functions being called. The RSR traffic considered is based on the dynamic information obtained with the average profile also used in Sections 5.2 and 5.3. The register assignment which minimizes the RSR traffic is determined as explained below. Once all functions at height 1 have their registers assigned, we move to the functions at height 2 and so on. Let us illustrate the RSR traffic elimination with the example given in Figure 5.11.

Figure 5.11 shows four functions which are part of a program. Function f_1 calls the functions f_2 , f_3 , and f_4 with dynamic frequencies 10, 5, and 30, respectively. We assume that we have in total 5 TBP registers available ($\{r_1, r_2, r_3, r_4, r_5\}$). Figure 5.12 shows the registers already assigned for the functions f_2 , f_3 , and f_4 . Let us discuss now how

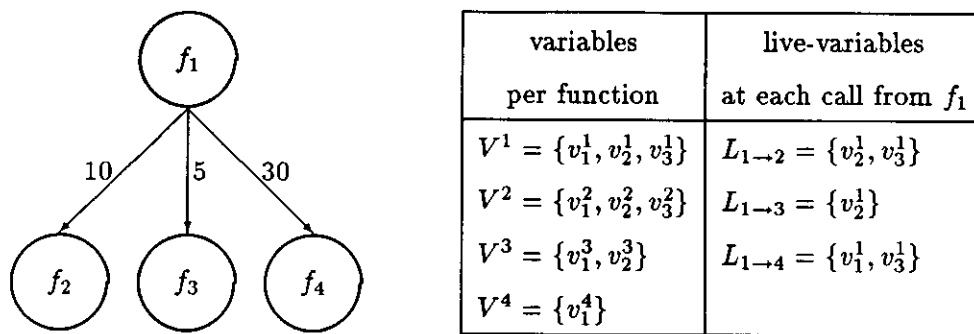


Figure 5.11: An Example of Register Assignment for Policy A⁸-live

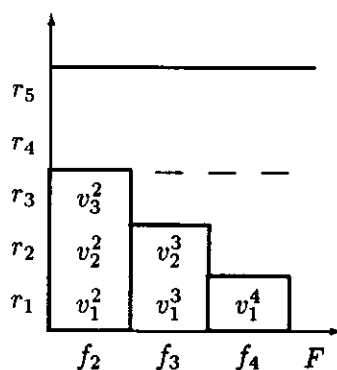


Figure 5.12: Registers Assigned to Functions f_2 , f_3 , and f_4

to assign registers for the variables of function f_1 . Since each variable can be assigned to only one register and the cost of this assignment can be determined (as we will see below), this is a well-understood linear programming problem [Hill67, Section 6.4].

The cost of assigning a variable to a register is given as the sum of RSR traffic generated at all calls on which this variable is alive when this register is already assigned to the function called. For instance, variable v_2^1 is alive when the functions f_2 and f_3 are called. If v_2^1 is assigned to either r_1 or r_2 , the register will have to be saved/restored in both calls because both registers are used by f_2 and f_3 ; thus, the cost⁸ will be 15 (= 10 savings for f_2 + 5 savings for f_3). However, if it is assigned to r_3 , the register will only have to be saved/restored on the call to f_2 ; thus, the cost will be 10. Let c_{ji}^1 be the cost of assigning variable v_j^1 to register r_i . For this example, the cost matrix C^1 is:

$$C^1 = \begin{pmatrix} r_1 & r_2 & r_3 & r_4 & r_5 \\ \begin{pmatrix} 30 & 0 & 0 & 0 & 0 \\ 15 & 15 & 10 & 0 & 0 \\ 40 & 10 & 10 & 0 & 0 \end{pmatrix} & \begin{matrix} v_1^1 \\ v_2^1 \\ v_3^1 \end{matrix} \end{pmatrix}$$

⁸Since the saving traffic is the same as the restoring traffic, only the saving is considered to compute the cost.

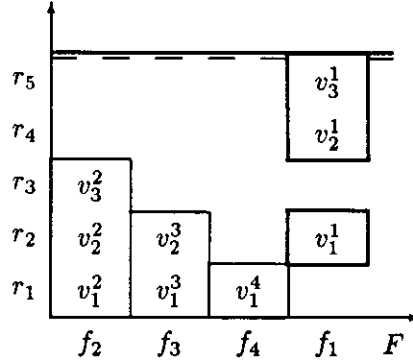


Figure 5.13: Variable-to-Register Assignment for Function f_1

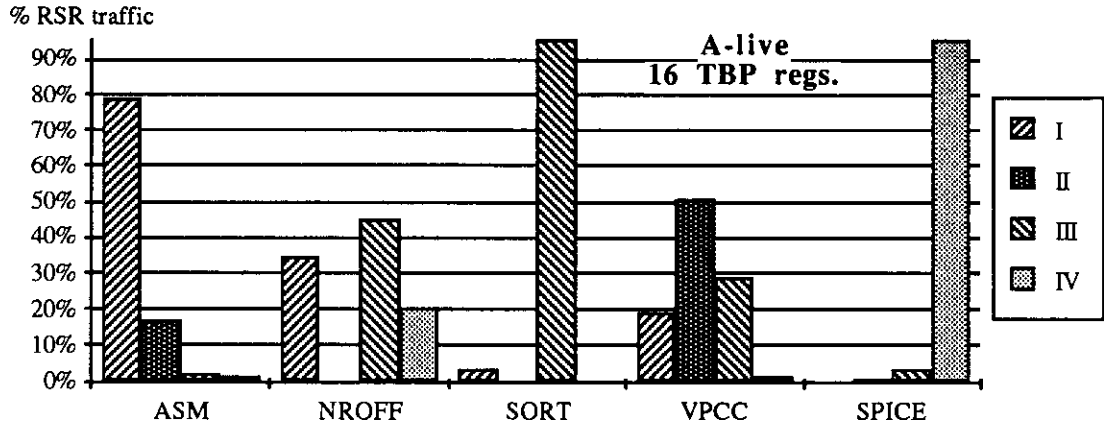


Figure 5.14: Distribution of the RSR Traffic Caused by Policy A-live

To find a solution to the assignment problem the procedure described in [Hill67, Section 6.4] is applied. The optimal solution obtained for this example is the one shown in Figure 5.13. In this particular case, the RSR traffic generated by this assignment will be zero. Notice that this optimal solution might not be the optimal for the whole call graph, although it is for the partial set of functions being considered. Once registers have been assigned, the four functions can be considered as only one leaf function that uses 5 registers. Thus, we can assign registers to the functions that call f_1 using the same approach.

The above described procedure has one main drawback. Since the cost of saving/restoring registers which are not used for any of the functions being called is zero, these registers are the best candidates to be selected for assignment. However, this is not a good policy because the RSR traffic eliminated corresponds to functions which are at the bottom of the call graph, which not necessarily are the ones that generate more RSR traffic. For instance, Figure 5.14 shows the distribution of RSR traffic caused by

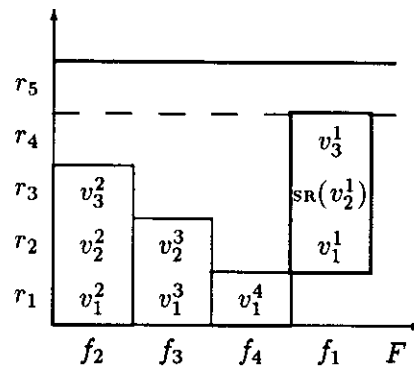


Figure 5.15: Variable-to-Register Assignment for Function f_1 when $K = 11$

Policy A-live when 16 TBP registers are available for the five programs.⁹ As we can see in the figure, SPICE is the only program which has 96% of the RSR traffic at the bottom of the call graph. SORT and ASM have their RSR traffic concentrated in a single region (96% in region III and 79% in region I, respectively), while NROFF and VPCC have their RSR traffic distributed among the regions. Thus, it is convenient not to exhaust all free registers for the functions at the bottom of the call graph.

Keeping Free Registers for Functions up in the Call Graph

To reserve some registers for the functions up in the call graph, registers which are not used by any of the functions being called get a cost K , instead of 0, to preserve some of them for functions up in the call graph. For instance, Figure 5.15 shows another variable-to-register assignment for $K = 11$. Here variable v_1^2 gets assigned to register r_3 so that it has to be saved/restored 10 times when f_2 is called. If more than 10 calls are required to reach f_1 , then there is still a register available to obtain a larger register saving/restoring reduction. In general, functions in the bottom of the call graph with low execution frequency can share the same registers as their descendents.

The value K is not defined as a unique constant for each function, but as an n -element vector (with n being the number of TBP registers). For any given function in the call graph, $\sum_{j=0}^i K[j]$ indicates the maximum number of saving operations¹⁰ to be eliminated if i free registers are left for one of the function's ancestors. The algorithm given below shows exactly how the values of K are computed. Also, an explanation of why the savings for only one of the function's ancestors are considered is given after the algorithm.

⁹The RSR traffic shown in the figure corresponds to the possible traffic which can be eliminated. That is, it does not include the RSR traffic caused by functions in a cycle because their RSR traffic cannot be eliminated as we will discuss afterwards.

¹⁰Remember that only the saving traffic is considered to compute the cost since the restoring traffic is the same.

A One-Pass Algorithm to Perform Register Assignment

We now present an algorithm that describes our first approach to perform register assignment for Policy A^g-live. This algorithm assumes that the call graph is acyclic, i.e., the program does not have recursive functions. In this first approach, the call graph is only traversed once. Afterwards, we will discuss how to handle recursive functions and present another approach in which the call graph is traversed twice.

Algorithm 1 Register Assignment for Policy A^g-live.

Inputs: The call graph G , the set of defined functions (F), the sets of selected local scalar variables for allocation per function (V^i), the live variables at each call ($L_{i \rightarrow j}$), and the dynamic profiled frequency of each call ($Q_{i \rightarrow j}$).

Outputs: The set of registers assigned per function (M^i), the set of registers to be saved/restored per call ($SR_{i \rightarrow j}$), and a matrix A that indicates the register assignment per function:

$$a_{ij} = \begin{cases} v_k^j & \text{if } v_k^j \in V^j \text{ is assigned to register } r_i \\ \emptyset & \text{otherwise} \end{cases}$$

Locals: The algorithm uses a matrix C to compute the assignment costs (as described in Figure 5.16), a set U^j per function f_j to indicate the registers used by this function and all its descendents in the call graph, and three n -element vectors ($Knew$, $Kold$, and X) to estimate the maximum number of savings to be eliminated. Moreover, the following definitions are required:

$$\begin{aligned} \text{REG_USED}(r_i, U^j) &= \begin{cases} 1 & \text{if } r_i \in U^j \\ 0 & \text{otherwise} \end{cases} \\ \text{LIVE_VAR}(v_k^j, L_{j \rightarrow t}) &= \begin{cases} 1 & \text{if } v_k^j \in L_{j \rightarrow t} \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

Method: Apply algorithm given in Figure 5.16. The initial call is performed with f_j being the *main* function and $Kold = \{0, 0, \dots, 0\}$. Notice that the algorithm does not consider the assignment of some local scalar variables to TBD registers for leaf functions. For intra-procedural allocation, this is not necessary because a leaf function always receives a clean set of registers (since the caller does not know whether the callee is a leaf function). However, for inter-procedural allocation, we can assign local scalar variables to TBD registers for leaf functions to reduce the number of registers required by the leaf functions. To keep the algorithm simple we assume that this assignment has already been performed and the variables already assigned have been removed from the V^j set. \square

```

procedure gblAlive ( $f_j, Kold$ )
  if  $f_j$  has already been VISITED then return fi
   $U^j \leftarrow M^j \leftarrow \emptyset; m \leftarrow |V^j|$ 
  for each  $f_t \in F$  such that  $j \rightarrow t$  do  $SR_{j \rightarrow t} \leftarrow \emptyset$  od
  if  $f_j$  is NOT a leaf function then
    (* compute savings to be eliminated for free registers left to callers *)
    for each  $v_s^j \in V^j$  do  $X[s] \leftarrow \sum_{\forall t, j \rightarrow t} \{Q_{j \rightarrow t} \text{LIVE\_VAR}(v_s^j, L_{j \rightarrow t})\}$  od
    sort  $X$  such that  $X[s_1] \geq X[s_2] \geq \dots \geq X[s_n]$ 
     $p \leftarrow q \leftarrow 1$ 
    for  $i = 1, 2, \dots, n$  do
      if  $X[s_p] > Kold[q]$  then  $Knew[i] \leftarrow X[s_p]; p \leftarrow p + 1$ 
      else  $Knew[i] \leftarrow Kold[q]; q \leftarrow q + 1$  fi
    od
    (* visit each function in reverse depth-first search order *)
    for each  $f_t \in F$  such that  $j \rightarrow t$  do
      gblAlive ( $f_t, Knew$ )
       $U^j \leftarrow U^j \cup U^t$ 
    od
  fi
  (* perform register assignment according to cost *)
  if  $f_j$  is a leaf function or  $U^j = \emptyset$  then
     $a_{1,j} \leftarrow v_1^j, a_{2,j} \leftarrow v_2^j, \dots, a_{m,j} \leftarrow v_m^j$ 
     $M^j \leftarrow \{r_1, r_2, \dots, r_m\}$ 
  else  $p \leftarrow 1$ 
    for each  $v_k^j \in V^j$  do
      for  $i = n, n-1, \dots, 1$  do
        if  $r_i \in U^j$  then
           $c_{ki} \leftarrow \sum_{\forall t, j \rightarrow t} \{Q_{j \rightarrow t} \text{REG\_USED}(r_i, U^t) \text{LIVE\_VAR}(v_k^j, L_{j \rightarrow t})\}$ 
        else
           $c_{ki} \leftarrow Kold[p]; p \leftarrow p + 1$ 
        fi
      od
    od
    od
    Perform register assignment based on [Hill67]
     $a_{i_1,j} \leftarrow v_1^j, a_{i_2,j} \leftarrow v_2^j, \dots, a_{i_m,j} \leftarrow v_m^j$ 
     $M^j \leftarrow \{r_{i_1}, r_{i_2}, \dots, r_{i_m}\}$ 
    for each  $f_t \in F$  such that  $j \rightarrow t$  do
      for each  $v_k^j \in V^j$  do
        if  $v_k^j \in L_{j \rightarrow t}$  and  $r_{i_k} \in U^t$  then  $SR_{j \rightarrow t} \leftarrow SR_{j \rightarrow t} \cup \{r_{i_k}\}$  fi
      od
    od
  fi
   $U^j \leftarrow U^j \cup M^j$ 
  mark  $f_j$  as VISITED
end gblAlive.

```

Figure 5.16: Register Assignment for Policy A⁵-live

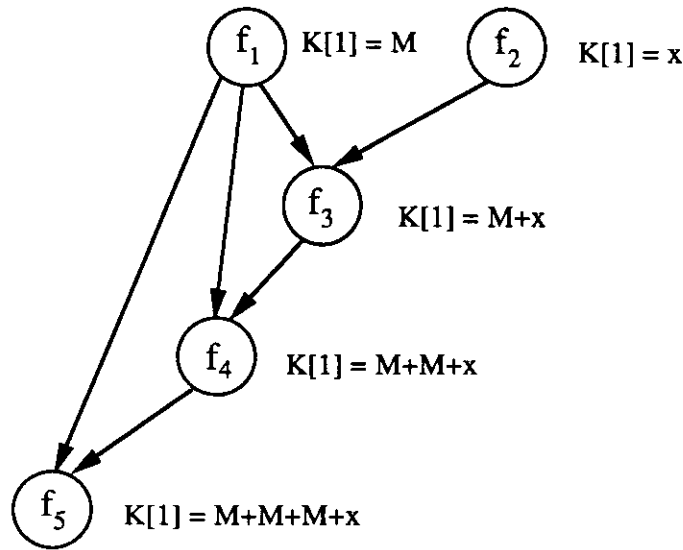


Figure 5.17: Wrong Computation for $K[i]$ when Values Are Added

As we said in the definition of K above, $\sum_{j=0}^i K[j]$ indicates the saving operations to be eliminated when i free registers are left for *one* of the function's ancestors (see Algorithm 1). However, the value of $\sum_{j=0}^i K[j]$ does not correspond to the *total number of savings* to be eliminated when i registers are left free for *all* the function's antecedents, because the functions at the same height of the one for which $\sum_{j=0}^i K[j]$ savings can be eliminated (if i registers are free), can also reduce their RSR traffic by using the same i free registers.

An alternative computation of K to indicate the savings to be eliminated by all the ancestors at a given height cannot be performed. That is, the values of K for different functions cannot be added. If they were, the estimate of the savings to be obtained would be incorrect because callers to functions at different heights would have their savings accounted multiple times. This is shown in Figure 5.17. Let us assume that the maximum amount of traffic to be eliminated for f_1 , when there is a free register, is M and for f_2 , is x . If f_3 leaves one register free for its callers, the total number of savings operations to be eliminated are $M + x$ (assuming that $M + x$ is larger than the local traffic generated by f_3). For f_4 , $M + M + x$. For f_5 , $M + M + M + x$. That is the M savings in f_1 are accounted multiple times and this is not correct because if function f_5 leaves one register free for its callers, only $M + x$ savings will be eliminated, not $3M + x$. For this reason, the K values are computed as indicated in Algorithm 1.

Recursive Functions

Let us discuss now how to handle recursive calls. Let $f_{j_1}, f_{j_2}, \dots, f_{j_q}$ be functions in the call graph that form part of a cycle (i.e., $f_{j_1} \rightarrow f_{j_2} \rightarrow \dots \rightarrow f_{j_q} \rightarrow f_{j_1}$). No RSR traffic can be eliminated for the registers alive in this path because the registers might be used by one of the function's descendents, the function itself. However, the RSR traffic can

be eliminated for the registers which are dead in the call (or calls) to functions in the cycle, but alive to functions outside the cycle. Thus, variables for functions in a cycle are divided into two groups:

1. Variables which are *alive in any call* to a function in the cycle. Since the registers to which these variables are assigned have always to be saved/restored, they are assigned to the registers already used by the function's descendents in order to keep as many free registers as possible.
2. Variables which are *dead in every call* to functions in the cycle. These variables are assigned in the same way as the ones for a non-recursive function. That is, their assignment cost matrix is computed and an optimal register assignment is performed as described in Algorithm 1.

Therefore, all functions in a cycle have to be marked so that they can be detected by Algorithm 1, to perform the register assignment described. Moreover, the calls which cause a cycle in the call graph have also to be marked. These calls have to be skipped to avoid that the recursive procedure *gblAlive* enters in an infinite loop while visiting each descendent in reverse depth-first search order. Notice that we said *marked* and not *removed* from call graph because they are needed to specify the *SR* set.

A Comparison with Other Inter-Procedural Register Allocators

Let us compare now our approach with Steenkiste's [Stee87] and Chow's [Chow88] inter-procedural allocators. Steenkiste only considers the maximum number of registers already assigned to the function descendents. Disjoint registers are assigned to the functions with higher height, while registers are available. This approach works well when most of the traffic is concentrated at the bottom of the call graph, as it is the case for LISP, the language for which this inter-procedural allocator was designed. LISP performs, on the average, 80% of the calls at the bottom of the call graph. Steenkiste reports that his allocator eliminates 70% of the RSR traffic.

As we have seen in Figures 5.8 and 5.14, C programs do not usually have a similar locality either for the percentage of functions called or the percentage of traffic generated by Policy A-live. For this reason, Steenkiste's allocator would not perform as well as for LISP. A comparison of the RSR traffic generated by his disjoint register allocator is given below.

Chow's inter-procedural allocator only considers the functions defined in the module being parsed. That is, the allocator has only a partial view of the call graph. For the functions being considered, Chow's allocator performs live-variable analysis and tries to assign live variables in a call to registers not used by the descendents and dead variables to registers used. In this case, more free registers are left for the callers. Moreover, Chow mentions a priority criterion to decide when to take a register or when to leave it for the callers. However, not enough information is provided in his paper to present a precise comparison of his priority criterion with our approach, as we will see below.

no. TBP regs.	program	Policy A-live	Policy A ^g -live					
			disjoint	One Pass			Two Passes	
				K = 0	by def.	priority	no tags	with tags
12	ASM	1.0 (10.69)	0.83	0.83	0.76	0.78	0.68	0.68
	NROFF	1.0 (2.76)	0.79	0.79	0.79	0.79	0.79	0.79
	SORT	1.0 (14.33)	0.19	0.17	0.17	0.17	0.17	0.17
	VPCC	1.0 (5.85)	0.88	0.82	0.66	0.66	0.68	0.66
	4 P.	1.0 (6.09)	0.53	0.51	0.47	0.47	0.46	0.46
	SPICE	1.0 (9.78)	0.41	0.26	0.25	0.25	0.25	0.25
16	ASM	1.0 (13.07)	0.86	0.85	0.66	0.66	0.49	0.49
	NROFF	1.0 (2.76)	0.77	0.77	0.77	0.77	0.74	0.74
	SORT	1.0 (14.33)	0.19	0.11	0.11	0.11	0.11	0.11
	VPCC	1.0 (5.85)	0.88	0.73	0.63	0.63	0.64	0.60
	4 P.	1.0 (6.19)	0.53	0.46	0.42	0.42	0.40	0.40
	SPICE	1.0 (11.81)	0.35	0.21	0.22	0.22	0.23	0.23
24	ASM	1.0 (13.07)	0.86	0.71	0.36	0.36	0.18	0.08
	NROFF	1.0 (2.76)	0.77	0.73	0.72	0.72	0.67	0.66
	SORT	1.0 (14.33)	0.15	0.10	0.10	0.10	0.10	0.10
	VPCC	1.0 (5.85)	0.87	0.59	0.63	0.63	0.64	0.58
	4 P.	1.0 (6.19)	0.51	0.41	0.38	0.38	0.35	0.33
	SPICE	1.0 (15.79)	0.24	0.11	0.14	0.14	0.13	0.13
32	ASM	1.0 (13.07)	0.85	0.64	0.10	0.08	0.04	0.02
	NROFF	1.0 (2.76)	0.68	0.66	0.66	0.66	0.66	0.66
	SORT	1.0 (14.33)	0.15	0.10	0.10	0.10	0.10	0.10
	VPCC	1.0 (5.85)	0.82	0.55	0.58	0.58	0.63	0.55
	4 P.	1.0 (6.19)	0.48	0.38	0.33	0.33	0.34	0.32
	SPICE	1.0 (20.89)	0.22	0.11	0.12	0.12	0.13	0.12

Table 5.6: RSR Traffic Reduction for Policy A^g-live

Evaluation of the RSR Traffic Reduction

Table 5.6 shows the RSR traffic generated by the approaches discussed before (and for three additional approaches that will be introduced below) normalized with respect to Policy A-live. The approach labeled “disjoint” corresponds to Steenkiste’s interprocedural allocator. As expected from our previous discussion, only SPICE and SORT reduce significantly their RSR traffic (59%–78% for SPICE and 81%–85% for SORT). These programs have most of their RSR traffic produced by the functions at the bottom of the call graph (see Figure 5.14¹¹). For the other programs, less RSR traffic is eliminated since no free registers are available for the functions (up in the call graph) which need

¹¹ Although Figure 5.14 shows that SORT has 96% of its RSR traffic in Region III, we have to consider that SORT has a maximum height of 9. Thus, functions in Region III can be considered to be at the bottom of the call graph.

them.

The column labeled " $K = 0$ " corresponds to the approach mentioned in our first example. Live-variable information is taken into the account to assign registers (as in Chow's allocator) and free registers are taken as soon as needed because they do not have any cost associated with them (since K is zero). Again, SPICE and SORT are the programs which eliminate most of their RSR traffic, but not the others for the reasons given previously.

The following column, labeled "by def." (by definition), corresponds to the RSR traffic generated by Algorithm 1. The name "by definition" comes from the order taken to visit the descendants: they are visited according to the order they have been defined by the programmer, i.e., they appear in the source code. As we can see in the table, programs, like ASM, which have most of their RSR traffic up in the call graph eliminate more RSR traffic than with the previous approaches. For instance, when 16 TBP registers are available, the RSR traffic eliminated is 34% with "by def." versus 15% with " $K = 0$."

If Chow's allocator had a global view of the entire call graph (rather than a partial one to the module being parsed) and no priority was considered for leaving free registers to the ancestors, then its performance would be equivalent to our " $K = 0$ " approach. With Chow's undisclosed priority criterion, its performance would be between the " $K = 0$ " column and the "by def." column.

Surprisingly, for some programs, the RSR traffic eliminated has not increased much even with a larger set (NROFF) or has become even worse (VPCC and SPICE). This is caused by the fact that registers are assigned according to the savings to be eliminated (i.e., K) as given by a specific path. Remember that the functions are visited in reverse depth-first search order. It could be that when a different path is taken, the values for K will be greater and, therefore, more free registers will be left for the callers. To overcome this limitation, three new approaches are presented next.

The first one is still performing a one-pass algorithm through the call graph. However, rather than selecting the next function to visit by order of definition (as indicated in Algorithm 1), the most (profiled) frequently used path is selected first. We expected that the values of K would be greater and that more free registers would be left for the callers. This is not the case as we can see in the column labeled "priority." The RSR traffic generated is the same as the one produced by Algorithm 1, except for ASM. Thus, the selection by definition gives us the same RSR traffic reduction than the selection by priority.

For the other two approaches, we perform two passes through the call graph. In the first pass, the values of K are generated for each function so that K indicates the maximum number of savings to be eliminated from any possible path to this function. In the second pass, register assignment is performed as before.

The differences between these two new approaches (labeled "no tags" and "with tags") are explained with the case example given in Figure 5.18. Let us assume that f_i has a live variable which generates a significant amount of traffic (M). The savings

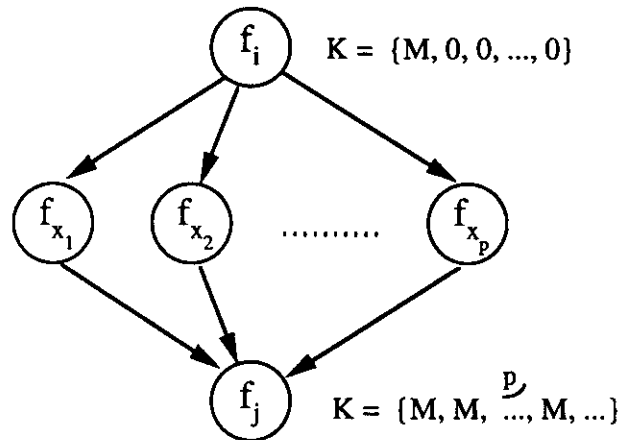


Figure 5.18: K Values with Two Passes

associated to these variables are passed to each function $f_{x_1}, f_{x_2}, \dots, f_{x_p}$. Since we assume that M is large, each of these functions will request f_j to leave a free register. Thus, the K vector for f_j will have p requests of M savings each, i.e., a request to leave p free registers for the callers. If p registers are left for the callers, f_i will be able to use only one of them. This is the reason why VPCC generates more RSR traffic with “no tags” than with “by def.”

To prevent this situation to happen, each value of K is tagged with a unique identifier so that duplicated requests for a given function are eliminated. The RSR traffic generated with this approach is shown in the last column of Table 5.6 labeled “with tags.” As we can see in the table, this is the approach which generates the least RSR traffic for all programs, except for SPICE. As we have already mentioned, SPICE has most of its RSR traffic at the bottom of its call graph (see Figure 5.14) and, for this reason, the “ $K = 0$ ” approach is performing slightly better than “with tags.” Based on the profile information, the SPICE behavior can be detected by the optimizer and, in this case, it can perform a “ $K = 0$ ” optimization for programs like SPICE and a “with tags” optimization for the other programs.

Notice that for some programs the traffic reduction obtained by the different approaches is identical (e.g., for SORT) or almost equivalent for any register-set configuration (e.g., for SPICE) or almost equivalent for certain register-set configurations (e.g., for NROFF and VPCC). However, for the four programs,

- The traffic never becomes worse when two passes with tags are performed than when only one pass is performed.
- For some programs, like ASM, which have the traffic more evenly distributed throughout the call graph, the traffic reduction becomes quite significant. For instance, the RSR traffic generated when registers are assigned by “with tags” for ASM is 89% (for 12 TBP registers), 74% (for 16), 22% (for 24), and 20% (for 32) with respect when they are assigned “by def.”

Therefore, the RSR traffic generated by the two approaches, “ $K = 0$ ” for SPICE and “with tags” for the other four programs, is the one that will be used henceforth to compare the RSR traffic produced by Policy A^g-live.

Summary

In conclusion, Policy A^g-live has between 46% (for 12 TBP registers) and 32% (for 32) of the RSR traffic generated by Policy A-live for the average of the four programs and between 26% and 11% for SPICE. The RSR traffic reduction is more significant for programs which have more of their RSR traffic produced at the bottom of the call graph like SPICE and SORT.

If we compare Policy A^g-live with Steenkiste’s inter-procedural register allocator, our inter-procedural register assignment algorithm has between 65% and 87% of the RSR traffic generated by Steenkiste’s allocator for the average of the four programs and between 54% and 61% for SPICE.

If Chow’s allocator had a complete view of the call graph rather than a partial one (only to the functions of the module being compiled as we discussed above), then its performance would be between the columns “ $K = 0$ ” and “by def.” In this case, Policy A^g-live would also generate less RSR traffic than Chow’s allocator for ASM, NROFF, SORT, and VPCC, and the same RSR traffic for SPICE.

5.4.2 Global Policy A-lvOpt

The goal of the Global Policy A-lvOpt (**A^g-lvOpt**) is the same one as Policy A^g-live: to eliminate the register saving/restoring instructions performed at a given call when the registers being saved/restored are not used by any of the functions that might be called from this point. However, we cannot use the same algorithm as Policy A^g-live to perform this, because we cannot eliminate the saving/restoring instructions for variables in registers whose saving/restoring traffic has already been optimized by Policy A-lvOpt. Only registers that have not been assigned by any of the descendent functions are candidates for this optimization. An explanation for this is given below. Since the RSR traffic can only be partially optimized, we have selected both Policies A^g-live and A^g-lvOpt for inter-procedural optimization so that we can verify whether Policy A^g-lvOpt generates less RSR traffic than Policy A^g-live as Policy A-lvOpt does for A-live.

In this section we discuss first why we cannot eliminate completely the RSR traffic for variables in registers whose RSR traffic has already been partially optimized per function. Afterwards, we present a one-pass algorithm to perform register assignment for Policy A^g-lvOpt. Finally, we discuss the RSR traffic reduction obtained by this algorithm and by a two-pass algorithm equivalent to the one discussed for Policy A^g-live.

Cases for which Global Policy A-lvOpt Cannot Be Optimized

The saving/restoring of a register which is not used by the function that it is being called and its descendents cannot always be eliminated (as in Policy A^g-live) because:

1. A saving instruction could have already been eliminated by Policy A-lvOpt. This situation is shown in Figure 5.19.a. Register r_3 is saved before the call to f_x . Since register r_3 is only read from this call to the call to f_y , Policy A-lvOpt has eliminated the register saving before this second call. Although f_x and its descendents do not make use of the register r_3 , the saving/restoring of the register cannot be eliminated since the already optimized (eliminated) saving relies that the register was previously saved.
2. A restoring instruction could have already been eliminated by Policy A-lvOpt. This situation is shown in Figure 5.19.b. Since the register r_3 is not used from the call to f_x to the call to f_y , the restoring instruction after the call to f_x and the saving instruction before the call to f_y have been eliminated by Policy A-lvOpt. Although f_x and its descendents do not make use of the register r_3 , the saving of the register cannot be eliminated since the already optimized (eliminated) restoring relies that the register was previously saved.

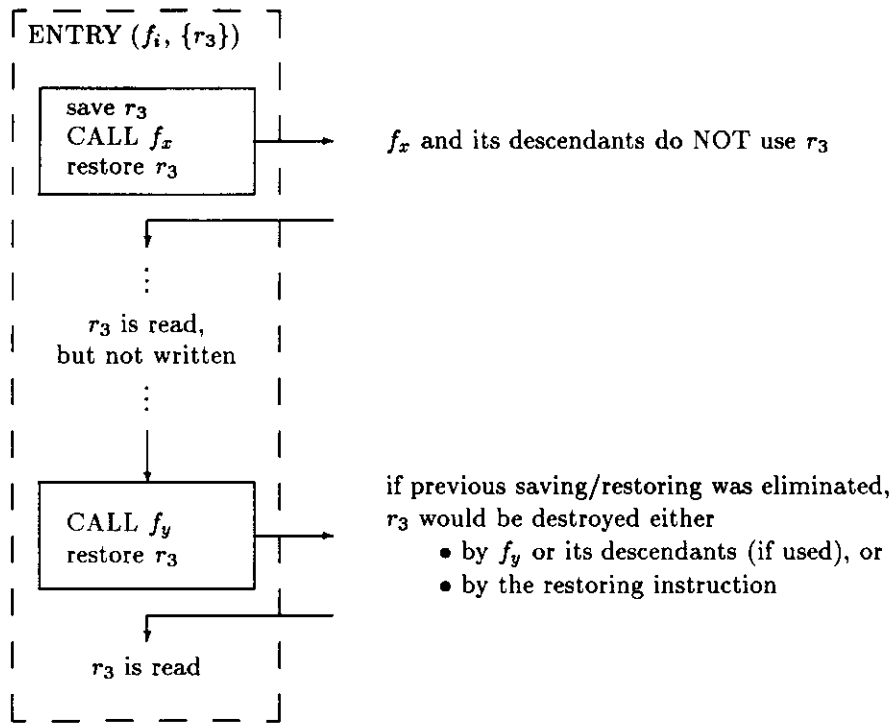
The above problems could be solved if control-flow information were available to the inter-procedural optimizer. In this case, the optimizer could follow all possible paths to check whether the elimination of a RSR instruction would have some negative effects on already optimized RSR instructions. However, one of our assumptions is that no control-flow information is available to the inter-procedural optimizer, except for the call graph (see step number 1 and Figure 5.2 in Section 5.1). The alternative of providing control-flow information to the inter-procedural optimizer is not attractive because of the extra computation time required to perform a second control-flow analysis and the extra storage space required in the data base to keep this information. Therefore, we prefer to perform Policy A^g-lvOpt without control-flow information.

A One-Pass Algorithm to Perform Register Assignment

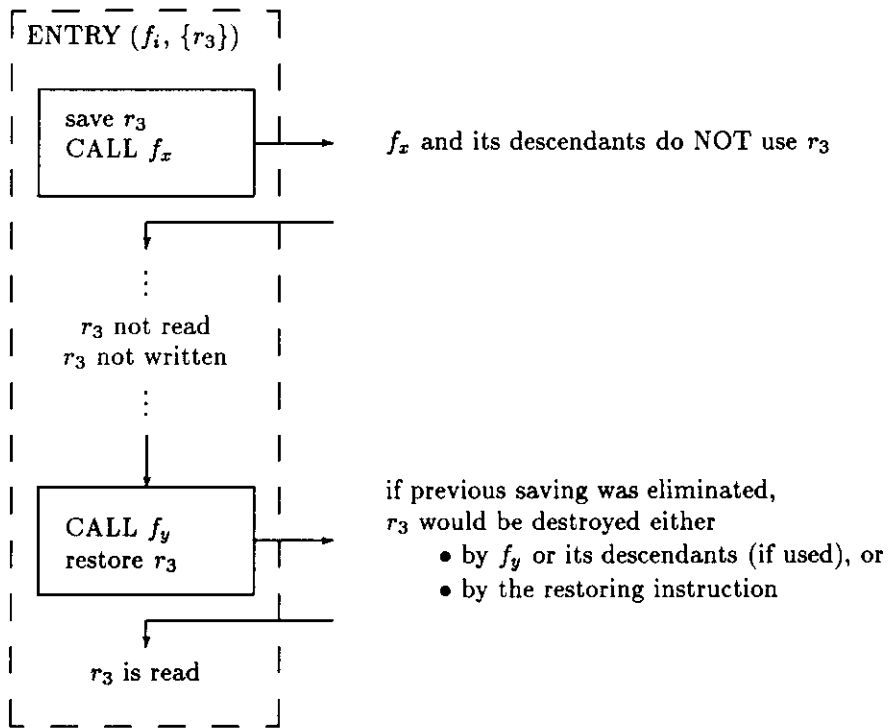
The only case in which it is always possible to eliminate the RSR traffic occurs when the variable is allocated to a register which is not used by any of the descendents of the function (see Figure 5.20). Algorithm 2 presents how register assignment is performed. Recursive functions are handled for Policy A^g-lvOpt as we discussed for Policy A^g-live (see Subsection 5.4.1). However, the algorithm does not show how register assignment is performed for recursive functions to keep it simple.

Algorithm 2 Register Assignment for Policy A^g-lvOpt.

Inputs: The call graph G , the set of defined functions (F), the sets of selected local scalar variables for allocation per function (V^i), the dynamic profiled frequency of each



(a) Eliminated Saving by Policy A-lvOpt



(b) Eliminated Restoring by Policy A-lvOpt

Figure 5.19: Cases for which Policy A^g-lvOpt Cannot Be Optimized

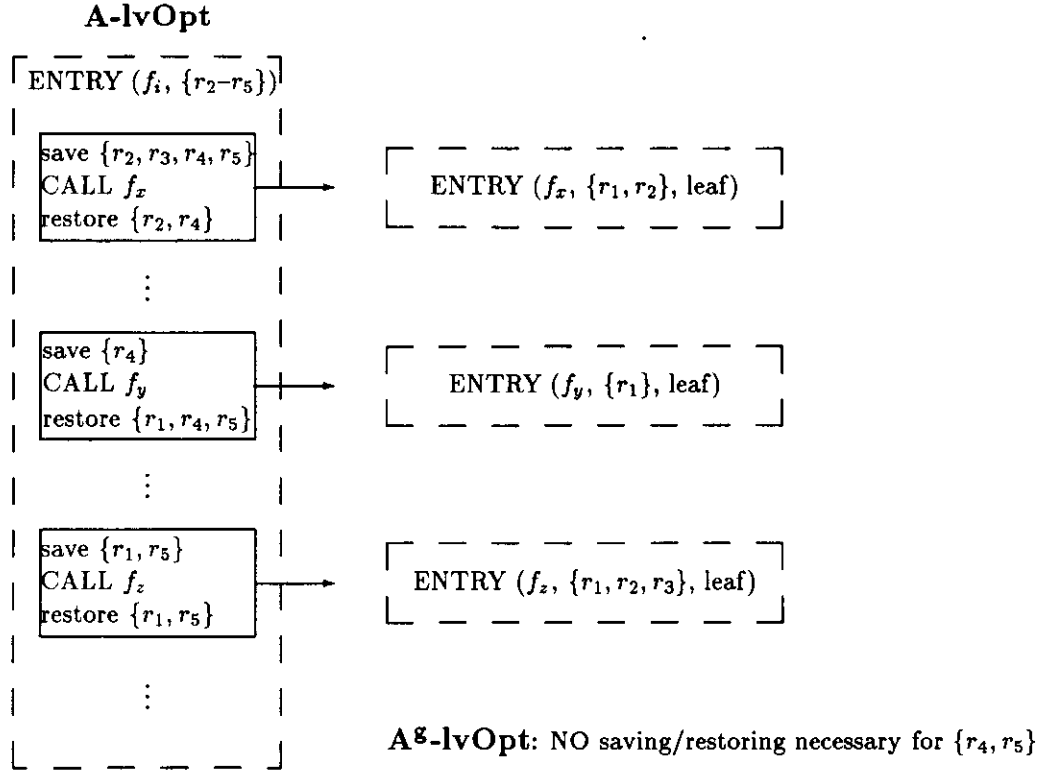


Figure 5.20: Policy A^g-lvOpt versus Policy A-lvOpt

call ($Q_{i \rightarrow j}$), and the registers to be saved ($S_{i \rightarrow j}$) and restored ($R_{i \rightarrow j}$) per each call. Notice that because of the way the compiler has assigned registers (see step 1 at the beginning of Section 5.4), there is a direct correspondence between variables and registers (i.e., v_k^j has been assigned to register r_k). For this reason, we can use variable subindices as register subindices for the intra-procedural assignment made by the compiler.

Outputs: The set of registers assigned per function (M^i), the set of registers to be saved and/or restored per call ($S_{i \rightarrow j}$ and $R_{i \rightarrow j}$) and the matrix A that indicates the register assignment per function (as defined in Algorithm 1).

Locals: This algorithm uses the matrix C , the U^j sets, and the three n -element vectors $Kold$, $Knew$, and X as given in Algorithm 1. It also requires the following definition:

$$\text{MEM_REFS}(r_k, S_{j \rightarrow t}, R_{j \rightarrow t}) = \begin{cases} 2 & \text{if } r_k \in S_{j \rightarrow t} \cap R_{j \rightarrow t} \\ 1 & \text{if } r_k \in S_{j \rightarrow t} \cup R_{j \rightarrow t} \wedge r_k \notin S_{j \rightarrow t} \cap R_{j \rightarrow t} \\ 0 & \text{if } r_k \notin S_{j \rightarrow t} \cup R_{j \rightarrow t} \end{cases}$$

Method: Apply algorithm given in Figure 5.21. As for Algorithm 1, we assume that local scalar variables assigned to TBD registers for leaf functions have already

```

procedure gblAlv_Opt ( $f_j, Kold$ )
  if  $f_j$  has already been VISITED then return fi
   $U^j \leftarrow M^j \leftarrow \emptyset; m \leftarrow |V^j|$ 
  if  $f_j$  is NOT a leaf function then
    (* compute savings to be eliminated for free registers left to callers *)
    for each  $v_s^j \in V^j$  do  $X[s] \leftarrow \sum_{\forall t, j \rightarrow t} \{Q_{j \rightarrow t} \text{MEM\_REFS}(r_s, S_{j \rightarrow t}, R_{j \rightarrow t})\}$  od
    sort  $X$  such that  $X[s_1] \geq X[s_2] \geq \dots \geq X[s_n]$ 
     $p \leftarrow q \leftarrow 1$ 
    for  $i = 1, 2, \dots, n$  do
      if  $X[s_p] > Kold[q]$  then  $Knew[i] \leftarrow X[s_p]; p \leftarrow p + 1$ 
      else  $Knew[i] \leftarrow Kold[q]; q \leftarrow q + 1$  fi
    od
    (* visit each function in reverse depth-first search order *)
    for each  $f_t \in F$  such that  $j \rightarrow t$  do
      gblAlv_Opt ( $f_t, Knew$ )
       $U^j \leftarrow U^j \cup U^t$ 
    od
  fi
  (* perform register assignment according to cost *)
  if  $f_j$  is a leaf function or  $U^j = \emptyset$  then
     $a_{1,j} \leftarrow v_1^j, a_{2,j} \leftarrow v_2^j, \dots, a_{m,j} \leftarrow v_m^j$ 
     $M^j \leftarrow \{r_1, r_2, \dots, r_m\}$ 
  else  $p \leftarrow 1$ 
    for each  $v_k^j \in V^j$  do
      for  $i = n, n-1, \dots, 1$  do
        if  $r_i \in U^j$  then
           $c_{ki} \leftarrow \sum_{\forall t, j \rightarrow t} \{Q_{j \rightarrow t} \text{MEM\_REFS}(r_k, S_{j \rightarrow t}, R_{j \rightarrow t})\}$ 
        else
           $c_{ki} \leftarrow Kold[p]; p \leftarrow p + 1$ 
        fi
      od
    od
    Perform register assignment based on [Hill67]
     $a_{i_1,j} \leftarrow v_1^j, a_{i_2,j} \leftarrow v_2^j, \dots, a_{i_m,j} \leftarrow v_m^j$ 
     $M^j \leftarrow \{r_{i_1}, r_{i_2}, \dots, r_{i_m}\}$ 
    for each  $f_t \in F$  such that  $j \rightarrow t$  do
       $Y \leftarrow S_{j \rightarrow t}; Z \leftarrow R_{j \rightarrow t}; S_{j \rightarrow t} \leftarrow R_{j \rightarrow t} \leftarrow \emptyset$ 
      for each  $v_k^j \in V^j$  do
        if  $r_{i_k} \in U^j$  then
          if  $r_k \in Y$  then  $S_{j \rightarrow t} \leftarrow S_{j \rightarrow t} \cup \{r_{i_k}\}$  fi
          if  $r_k \in Z$  then  $R_{j \rightarrow t} \leftarrow R_{j \rightarrow t} \cup \{r_{i_k}\}$  fi
        fi
      od
    od
  fi
   $U^j \leftarrow U^j \cup M^j$ 
  mark  $f_j$  as VISITED
end gblAlv_Opt.

```

Figure 5.21: Register Assignment for Policy A^g -lvOpt

no. regs.	program	Policy		Policy A ⁸ -lvOpt	
		A-lvOpt		One Pass	Two Passes
12	ASM	1.0	(5.79)	0.98	0.93
	NROFF	1.0	(1.69)	0.80	0.79
	SORT	1.0	(8.57)	0.36	0.36
	VPCC	1.0	(4.51)	0.88	0.90
	4 P.	1.0	(3.85)	0.63	0.64
	SPICE	1.0	(5.48)	0.58	0.63
16	ASM	1.0	(6.32)	0.95	0.79
	NROFF	1.0	(1.69)	0.78	0.74
	SORT	1.0	(8.57)	0.28	0.28
	VPCC	1.0	(4.51)	0.85	0.83
	4 P.	1.0	(3.88)	0.59	0.56
	SPICE	1.0	(6.25)	0.63	0.67
24	ASM	1.0	(6.32)	0.82	0.63
	NROFF	1.0	(1.69)	0.77	0.68
	SORT	1.0	(8.57)	0.26	0.26
	VPCC	1.0	(4.51)	0.82	0.76
	4 P.	1.0	(3.88)	0.56	0.51
	SPICE	1.0	(7.53)	0.52	0.56
32	ASM	1.0	(6.32)	0.70	0.30
	NROFF	1.0	(1.69)	0.68	0.68
	SORT	1.0	(8.57)	0.26	0.26
	VPCC	1.0	(4.51)	0.73	0.69
	4 P.	1.0	(3.88)	0.51	0.47
	SPICE	1.0	(8.56)	0.42	0.46

Table 5.7: RSR Traffic Reduction for Policy A⁸-lvOpt

been removed from V^j and that the initial call to $gblAlv_Opt$ is performed with f_j being the *main* function and $Kold = \{0, 0, \dots, 0\}$. \square

Evaluation of the RSR Traffic Reduction

Table 5.7 shows the RSR traffic generated by the previous one-pass algorithm normalized with respect to the RSR traffic produced by Policy A-lvOpt. When the call graph is traversed only once, Policy A⁸-lvOpt has between 63% (for 12 TBP registers) and 51% (for 32) of the RSR traffic produced by Policy A-lvOpt for the average of the four programs and between 58% and 42% for SPICE.

Table 5.7 also shows the RSR traffic generated when two passes are performed through the call graph. As we mentioned in the previous section, during the first pass,

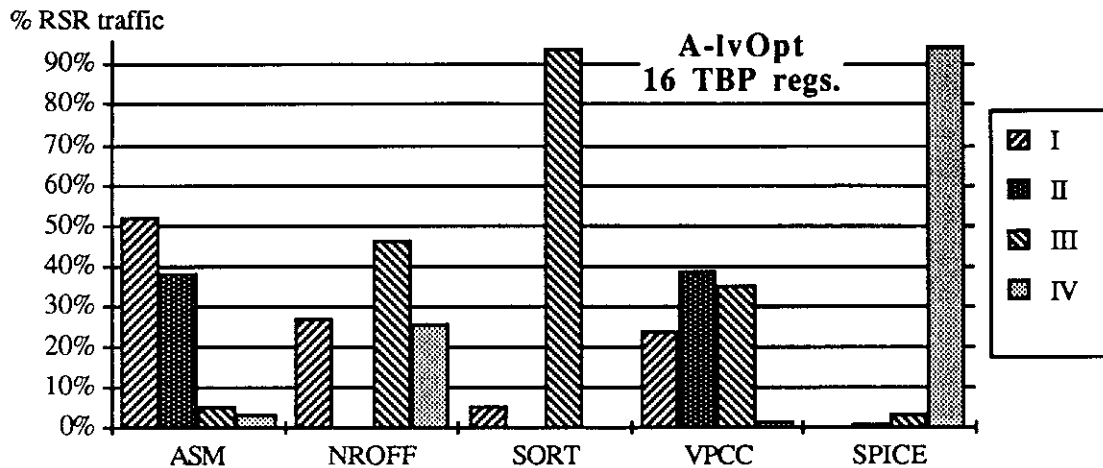


Figure 5.22: Distribution of the RSR Traffic Caused by Policy A-lvOpt

the values of K are generated per function taking into account all possible paths to this function and they are tagged with a unique identifier so that duplicated requests can be eliminated. The values of K are computed as indicated in Algorithm 2. During the second pass, registers are also assigned as shown in the algorithm; however, the values of K correspond to the maximum savings to be eliminated for any possible path, not only to the first path taken by the programmer's order of definition of the calls. When the call graph is traversed twice, Policy A^s-lvOpt has between 64% (for 12 TBP registers) and 47% (for 32) of the RSR traffic produced by Policy A-lvOpt for the average of the four programs and between 63% and 46% for SPICE.

If we compare the RSR traffic generated by Policy A^s-lvOpt with one and two passes, we obtain similar conclusions to the ones mentioned for Policy A^s-live:

- For programs which have most of its RSR traffic concentrated at the bottom of the call graph, one pass generates less or the same RSR traffic than two passes. This is the case for SORT and SPICE. Figure 5.22 shows the distribution of the RSR traffic generated by Policy A-lvOpt for these two programs.
- On the other hand, for programs which have their traffic more distributed through the call graph, two passes generates less RSR traffic than a single one. The difference between RSR traffic produced by the two approaches becomes more significant for programs which have most of their RSR traffic generated up in the call graph like ASM (see also Figure 5.22). For instance, when two passes are performed for ASM, the RSR traffic generated is 96% (for 12 TBP registers), 83% (16), 77% (24), and 43% (32) of the traffic produced when only one pass is performed.

Since the optimizer can detect when most of the RSR traffic is generated at the bottom of the call graph, it can select the best approach to perform register assignment. Thus, only one pass is performed for SPICE and two for the four programs. The traffic generated

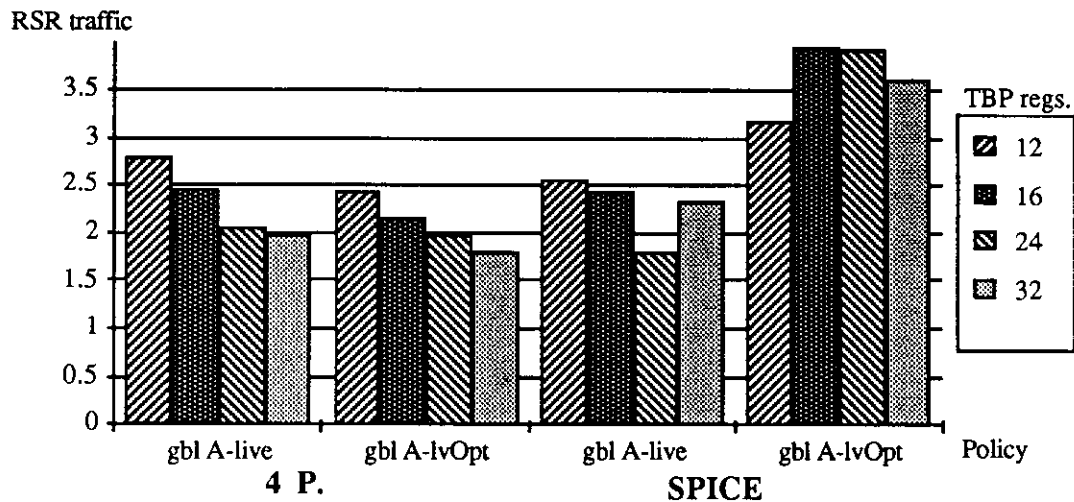


Figure 5.23: RSR Traffic Generated by Policies A^g-live and A^g-lvOpt

by these two approaches is the one that will be used henceforth to compare the RSR traffic produced by Policy A^g-lvOpt.

Let us compare now Policies A^g-live and A^g-lvOpt. While Policy A-lvOpt always generates less RSR traffic than Policy A-live, this is not the case for their respective inter-procedural optimizations. Policy A^g-live generates less RSR traffic than A^g-lvOpt for SORT and SPICE (for any number of TBP registers), for ASM (for 24 and 32 TBP registers), and VPCC (for 12, 16, and 24 TBP registers). For the other programs and register-set sizes, the opposite is true. Consequently, when inter-procedural optimization is performed, the selection of which intra-procedural policy to use is not as obvious as it is when only intra-procedural optimizations are performed. Although for the average of the four programs (see Figure 5.23¹²) the RSR traffic is smaller for Policy A^g-lvOpt than for Policy A^g-live (between 9% and 12%), the behavior of both policies for each individual program does not help us to decide the best policy to select. The selection could be based on one of following criteria:

1. To reduce the compiler complexity. In this case, Policy A-live should be selected since it is easier to implement than Policy A-lvOpt.
2. To generate more efficient code for the programs compiled only with intra-procedural optimizations; that is, for the programs that the programmer does not want to obtain a profile execution and to perform inter-procedural optimizations. In this case, Policy A-lvOpt should be selected.

Thus, if only one policy has to be implemented, the decision is left open for the compiler writer who knows the characteristics and goals of the compiler because our measurements

¹²Since Excel (the program used to generate the graphs) does not let us generate superindices, the names of the Global Policies A^g-live and A^g-lvOpt have been changed to **gbl A-live** and **gbl A-lvOpt**.

have shown that it is impossible to select a unique static policy which performs well for any program and register set configuration. However, the best solution is to let the inter-procedural optimizer decide which policy to select based on the profile information that it has available.

One additional advantage of inter-procedural optimization versus intra-procedural is that the RSR traffic decreases for larger register sets for every program, except SPICE (see Figure 5.23). As we mentioned in the previous chapter, this is not the case for the intra-procedural static policies. When inter-procedural optimization is performed, the RSR traffic decreases with larger register sets and, therefore, the overall data memory traffic is also reduced as we will discuss in Section 5.5.

Summary

In conclusion, Policy A^g-lvOpt has between 64% (for 12 TBP registers) and 47% (for 32) of the RSR traffic produced by Policy A-lvOpt for the average of the four programs and between 58% and 42% for SPICE. When only intra-procedural optimizations are performed, our measurements show that it is always more efficient to use Policy A-lvOpt than Policy A-live because less RSR traffic is generated. However, our measurements do not show which of their respective global policies should be used because for some programs, Policy A^g-live produces less RSR traffic than Policy A^g-lvOpt and for others, the opposite is true.

5.4.3 Global Policy B-lf

The Global B Policy with Leaf Functions (**B^g-lf**) eliminates the register saving/restoring instructions performed at function entry and return when the registers being saved/restored are not used by any of the functions that might call this function. Thus, we again have to find an optimal disjoint register assignment for the functions in the same path in the call graph to avoid as much RSR traffic as possible. In this section we first discuss the problem using a small example; second, we present an algorithm to perform register assignment; and, finally, we show the RSR traffic reduction obtained with inter-procedural optimization.

As before, our approach is based on the fact that functions which are never active simultaneously can share the same set of registers. Since registers are saved at the callee, to perform a disjoint register assignment, the inter-procedural optimizer needs to know which registers have already been assigned to the callers. Thus, we start assigning variables to the root function (i.e., the *main* function for C programs). Afterwards, we select a function for which its ancestors have their registers already assigned and assign registers to it.

Since the register assignment is performed from the root function to the leaf functions (rather than from the leaf to the root as Policies A^g-live and A^g-lvOpt), the call graph

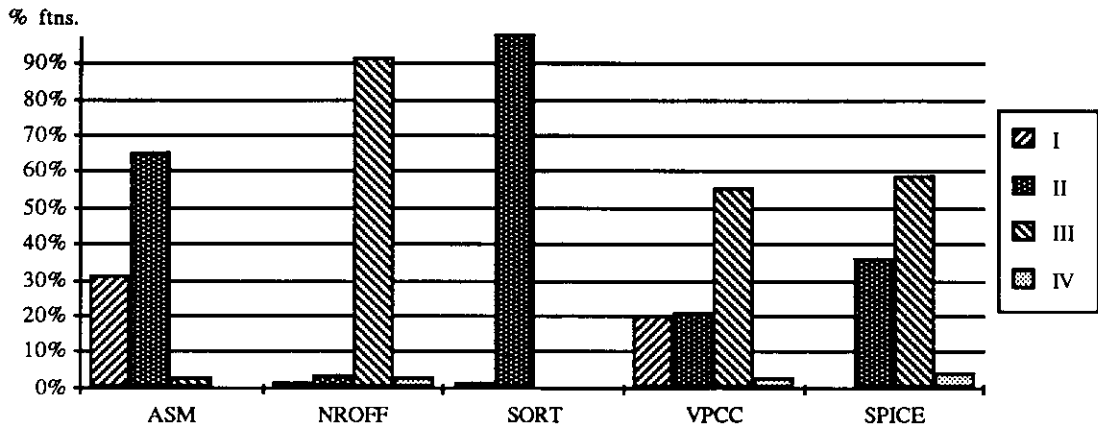


Figure 5.24: Distribution of Functions by Depth

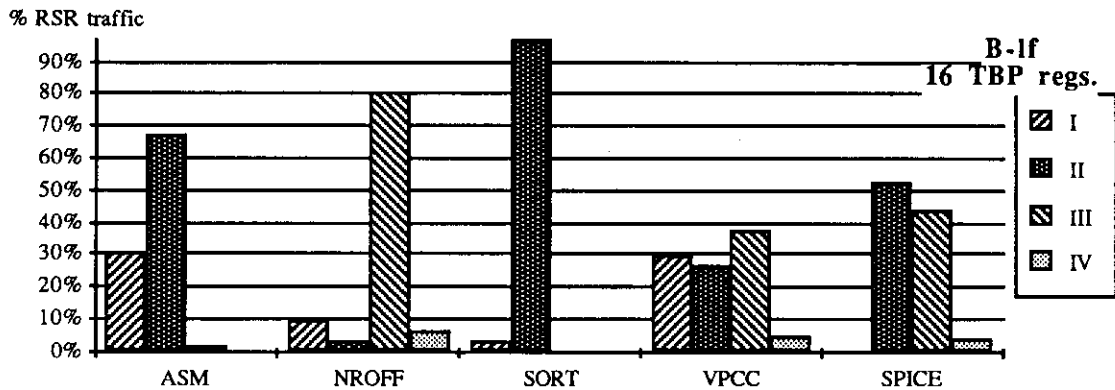


Figure 5.25: Distribution of the RSR Traffic Caused by Policy B-lf

is ordered differently. Functions in the call graph are grouped according to their *depth*. The depth of a function indicates the number of ancestors (or callers) in the longest path from this function to the root function. The root function is the only function which has zero depth. Notice that all leaf functions are not grouped in a single depth level (as they are for Policies A^s-live and A^s-lvOpt), but distributed throughout the call graph according to their depth.

Figures 5.24 and 5.25 shows the distribution by depth of the executed functions and of the RSR traffic caused by Policy B-lf. Each region corresponds to one fourth of the maximum depth of the call graph. For NROFF and SORT most of their executed functions are concentrated in one region, as well as their RSR traffic generated by Policy B-lf. For ASM, VPCC, and SPICE both the execution frequency and the RSR traffic are more evenly distributed throughout the call graph. As we can see in Figure 5.25,

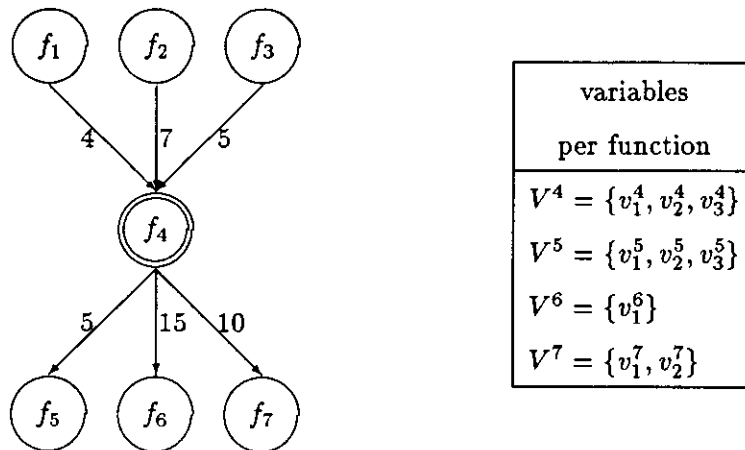


Figure 5.26: An Example of Register Assignment for Policy B^s-lf

only ASM and VPCC have 30% of their RSR traffic¹³ in Region I. Most of RSR traffic is concentrated in Regions II and III of the call graph. Thus, a disjoint register assignment as the one discussed for Policy A^s-live would only eliminate the RSR traffic at the top of the call graph which is not the most significant one. For this reason, this approach has not been considered for Policy B^s-lf and a more efficient algorithm has been used as we will discuss below.

An Example of RSR Traffic Elimination

Let us illustrate the problems of performing an assignment by the example given in Figure 5.26. Registers for the ancestors of f_4 have already been assigned. Now, we want to assign registers for the variables in f_4 . We have three possibilities:

1. Assign registers from the subset of registers not used by its callers. Let us refer to these registers as *free* registers since the RSR generated is zero. In this case, register saving/restoring is eliminated since the registers have not been used previously.
2. Assign registers from the subset of registers already used by its callers. In this case, no register saving and restoring is eliminated because the assigned registers will have to be saved and restored each time the function is called independently of who the caller has been. However, no free registers are consumed.
3. An intermediate approach on which we select registers from both subsets, which combines the advantages of the above approaches: the RSR traffic for some registers is eliminated without exhausting the number of free registers available for the callers.

¹³As for Figures 5.14 and 5.22, this RSR traffic only corresponds to the traffic which can be eliminated by the inter-procedural static policies. RSR traffic generated by functions in a cycle is not accounted in these figures because it cannot be eliminated.

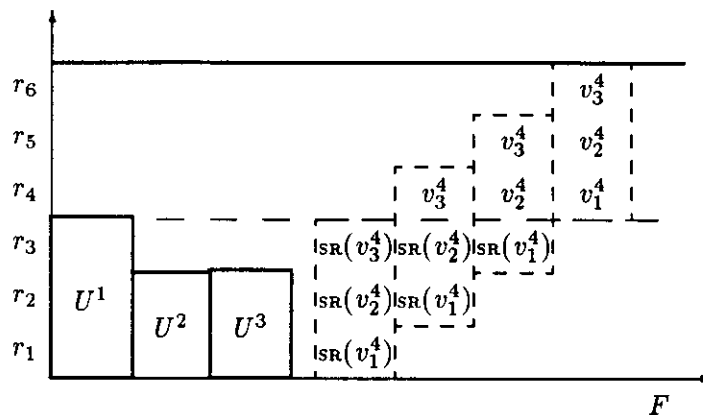


Figure 5.27: Alternative Register Assignments for Function f_4

These alternatives are reflected in Figure 5.27. The U^j sets now indicate the registers which are defined in function f_j and all its callers. The most suitable assignment depends on several factors:

- The number of free registers.
- The number of times that f_4 is executed.
- The number of registers required by f_4 .
- The number of functions executed after f_4 .
- The number of registers required by the descendents of f_4 .

For instance, if f_4 is executed only a few times, then the second alternative will be more attractive because more free registers will be left for the f_4 descendents and, therefore, more RSR traffic will be eliminated. On the other hand, if f_4 is a heavy-used function, then the first alternative will be, because no register saving/restoring traffic is generated. Thus, we have to compare the RSR traffic that can be eliminated when one of the registers not used by the function's ancestors is taken with the RSR traffic that could be eliminated if the register is left for one of the function's descendents.

If we just consider the *immediate* descendents, the register savings that could be eliminated can be easily computed. For instance, in our example, if only one free register is left to be shared by the functions f_5 , f_6 and f_7 , $1 \times 5 + 1 \times 15 + 1 \times 10 = 30$ savings could be eliminated; if 2 registers are left, $2 \times 5 + 1 \times 15 + 2 \times 10 = 45$ savings could be eliminated; and if 3 registers are left, $3 \times 5 + 1 \times 15 + 2 \times 10 = 50$ savings could be eliminated. Since f_4 is executed 16 times and requires 3 registers, when the 3 free registers are taken, $3 \times 16 = 48$ savings are eliminated; when only 2 are taken and 1 is left for the functions' descendents, $30 + 16 \times 2 = 62$ savings are eliminated; when only 1 is taken and 2 are left, $45 + 1 \times 16 = 61$ savings are eliminated; finally, when no register is taken, 50 savings are eliminated. Thus, the optimal solution for this specific case is the one shown in Figure 5.28.

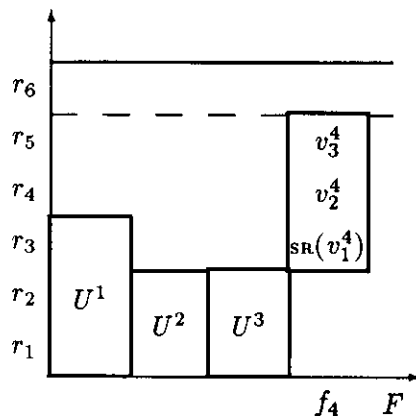


Figure 5.28: Optimal Variable-to-Register Assignment for Function f_4

Although it can be advantageous for f_4 to use 2 out of the 3 free registers, it prevents that these free registers be used by one of functions descendents. Thus, to preserve some free registers for the function's descendents we not only consider the savings to each immediate descendent, but also the savings for all descendents. This is important for the functions which are at the top of the call graph because as we can see in Figure 5.25, most of the RSR traffic generated by Policy B-1f is concentrated in Regions II and III of the call graph.

A Two-Pass Algorithm to Perform Register Assignment

We now present an algorithm that describes our approach to perform register assignment. We assume that the call graph does not have any cycles. If it does, then they are eliminated in the way explained in Subsection 5.4.1.

In the first pass through the call graph, the two-pass inter-procedural optimizer uses the frequencies obtained by the average profile to compute the savings generated for all functions at a given depth. Once the RSR traffic generated by all functions at a given depth is known, the *optimal* number of free register to be left for the function's descendents to generate the least RSR traffic is determined as it is indicated in the algorithm (in a similar manner to the example given above). This number could be the optimal for the execution frequencies obtained, but it might not be for the current program execution. However, this is usually the case as we will show below. In the second pass, registers are assigned and RSR instructions are eliminated according to the registers already assigned to the ancestors and the number of registers to be left for the descendents.

Algorithm 3 Register Assignment for Policy B^g-1f.

Inputs: The call graph G , the set of defined functions (F), the sets of selected variables for allocation per function (V^j), the profiled frequency of each executed function

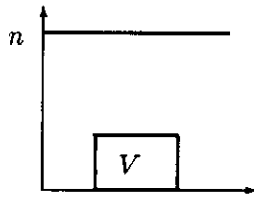
```

for each  $f_j \in F$  do for  $i = 1, 2, \dots, |V^j|$  do
     $X^{\text{DEPTH}(f_j)}[i] \leftarrow X^{\text{DEPTH}(f_j)}[i] + Q^j \text{VAR\_DEF}(v_i^j, V^j)$  od od
for  $l = 1, 2, \dots, \text{MAXDEPTH}$  do
     $Y^l[0] \leftarrow 0$ 
    for  $i = 1, 2, \dots, n$  do  $Y^l[i] \leftarrow Y^l[i-1] + X^l[i]$  od
od
 $W^1 \leftarrow |V^1 \equiv \text{main}|$ 
for  $l = 2, \dots, \text{MAXDEPTH}$  do
     $q \leftarrow n - W^{l-1}; Z[0] \leftarrow 0$ 
    if  $q \neq 0$  then
        for  $i = 1, 2, \dots, q$  do  $Z[i] \leftarrow Y^l[i] + \max_{k=i+1, \dots, \text{MAXDEPTH}}(Y^k[q-i])$ 
        find  $s$  such that  $\forall i, Z[i] \leq Z[s]$ 
         $W^l \leftarrow W^{l-1} + s$ 
    else  $W^l \leftarrow n$  fi
od
for  $f_j = O[1], O[2], \dots, O[p]$  do
     $U^j \leftarrow \bigcup_{t, t \rightarrow j} U^t; q \leftarrow |U^j|; m \leftarrow |V^j|$ 
    if  $m = 0$  then  $M^j \leftarrow SR^j \leftarrow \emptyset$ 
    else if  $f_j$  is in a cycle or  $q = n$  then (* see Figure 5.30.(I) *)
         $a_{1,j} \leftarrow v_1^j, \dots, a_{m,j} \leftarrow v_m^j$ 
         $M^j \leftarrow SR^j \leftarrow \{r_1, r_2, \dots, r_m\}$ 
    else if  $f_j$  is a leaf function then
        if  $m \leq n - q$  then (* see Figure 5.30.(II) *)
             $a_{q+1,j} \leftarrow v_1^j, \dots, a_{q+m,j} \leftarrow v_m^j$ 
             $SR^j \leftarrow \emptyset; M^j \leftarrow \{r_{q+1}, \dots, r_{q+m}\}$ 
        else (* see Figure 5.30.(III) *)
             $a_{n,j} \leftarrow v_1^j, \dots, a_{n-m+1,j} \leftarrow v_m^j$ 
             $SR^j \leftarrow \{r_{n-m+1}, \dots, r_q\}; M^j \leftarrow \{r_{n-m+1}, \dots, r_n\}$  fi
        else  $s \leftarrow W^{\text{DEPTH}(f_j)}$ 
        if  $m \leq s$  and  $q < s$  then
            if  $m \leq s - q$  then (* see Figure 5.30.(IV) *)
                 $a_{q+1,j} \leftarrow v_1^j, \dots, a_{q+m,j} \leftarrow v_m^j$ 
                 $SR^j \leftarrow \emptyset; M^j \leftarrow \{r_{q+1}, \dots, r_{q+m}\}$ 
            else (* see Figure 5.30.(V) *)
                 $a_{s,j} \leftarrow v_1^j, \dots, a_{s-m+1,j} \leftarrow v_m^j$ 
                 $SR^j \leftarrow \{r_{s-m+1}, \dots, r_q\}; M^j \leftarrow \{r_{s-m+1}, \dots, r_s\}$  fi
            else (* see Figure 5.30.(VI) *)
                 $a_{1,j} \leftarrow v_1^j, \dots, a_{m,j} \leftarrow v_m^j$ 
                 $M^j \leftarrow \{r_1, \dots, r_m\}$ 
                if  $q < m$  then  $SR^j \leftarrow \{r_1, \dots, r_q\}$  else  $SR^j \leftarrow M^j$  fi
            fi
        fi
     $U^j \leftarrow U^j \cup M^j$ 
od

```

Figure 5.29: Register Assignment for Policy B^s-lf

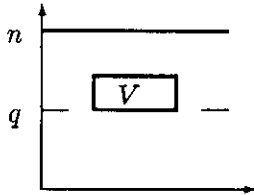
(I) if f is in a cycle or $q = n$ then



$$a_1 \leftarrow v_1, \dots, a_{-1} \leftarrow v$$

$$M \leftarrow SR \leftarrow \{r_1, \dots, r\}$$

(II) if f is a leaf function and $m \leq n - q$ then

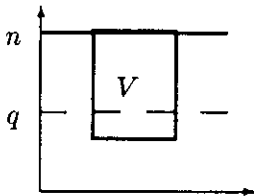


$$a_{+1} \leftarrow v_1, \dots, a_{+m} \leftarrow v$$

$$SR \leftarrow \emptyset$$

$$M \leftarrow \{r_{+1}, \dots, r_{+m}\}$$

(III) if f is a leaf function and $m > n - q$ then

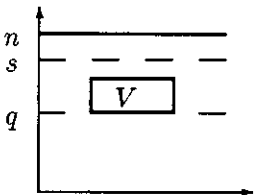


$$a_{-1} \leftarrow v_1, \dots, a_{-m+1} \leftarrow v$$

$$SR \leftarrow \{r_{-m+1}, \dots, r\}$$

$$M \leftarrow \{r_{-m+1}, \dots, r\}$$

(IV) if $m \leq s$ and $q < s$ and $m \leq s - q$ then

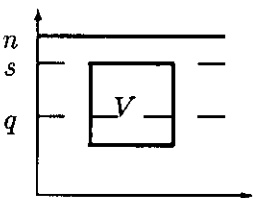


$$a_{+1} \leftarrow v_1, \dots, a_{+m} \leftarrow v$$

$$SR \leftarrow \emptyset$$

$$M \leftarrow \{r_{+1}, \dots, r_{+m}\}$$

(V) if $m \leq s$ and $q < s$ and $m > s - q$ then

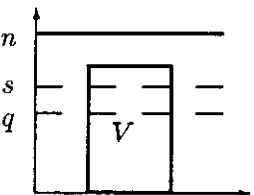


$$a_{-1} \leftarrow v_1, \dots, a_{-m+1} \leftarrow v$$

$$SR \leftarrow \{r_{-m+1}, \dots, r\}$$

$$M \leftarrow \{r_{-m+1}, \dots, r\}$$

(VI) if $m > s$ or $q \geq s$ then



$$a_1 \leftarrow v_1, \dots, a_{-1} \leftarrow v$$

$$M \leftarrow \{r_1, \dots, r\}$$

$$\text{if } q < m \text{ then } SR \leftarrow \{r_1, \dots, r\} \text{ else } SR \leftarrow M \text{ fi}$$

Figure 5.30: Alternative Register Assignments for Algorithm 3

($Q^j = \sum_{v_t, t \rightarrow j} Q_{t \rightarrow j}$), and a p -element array O with an acyclic numbering of the nodes such that, in increasing order, O always contains the caller before the callee (except for recursive functions). The recursive functions should have been taken into account when the O array is built.

Outputs: The set of registers assigned per functions (M^j), the set of registers to be saved and restored per function (SR^j), and the matrix A that indicates the register assignment per function as defined in Algorithm 1. Note that since all registers are saved at function entry, the precise variable-to-register assignment is not important. Each variable in V^j can be distributed arbitrarily among the registers given by M^j because each one of them generates the same amount of RSR traffic. We have just kept the definition of A to be consistent with the previous algorithms.

Locals: A set U^j per function to indicate the registers defined by the function f_j and all its ancestors in the call graph; the variable $MAXDEPTH$ which indicates the maximum depth of the call graph G ; $MAXDEPTH$ arrays $X^l[i]$, $i = 1, 2, \dots, n$ used to estimate the saving traffic caused by the variables v_i^j , $\forall j$ such that $DEPTH(f_j) = l$; $MAXDEPTH$ arrays $Y^l[i]$, $i = 0, 1, \dots, n$ used to estimate the saving traffic caused by the variables $v_1^j, v_2^j, \dots, v_i^j$, $\forall j$ such that $DEPTH(f_j) = l$; an n -element array $Z[i]$ used to estimate the optimal number of registers to leave free for the descendants; and a $MAXDEPTH$ -element array W^l used to indicate the maximum register number to be used for the functions at depth l . Notice that $W^l - W^{l-1}$ gives the number of free registers between the functions at depth l and the ones at depth $l - 1$.

Method: Apply algorithm given in Figure 5.29. As for Algorithm 1, we assume that local scalar variables assigned to TBD registers for leaf functions have already been removed from V^j . \square

For Policy B^g-lf it is not necessary to apply the cost-assignment algorithm as we did in Algorithms 1 and 2 for Policies A^g-live and A^g-lvOpt. Since all registers are saved at function entry if they are already used by the ancestors, the cost matrix would have identical values for all the assignments to registers already used by the ancestors.

Evaluation of the RSR Traffic Reduction

Table 5.8 shows the RSR traffic generated by the previous two-pass algorithm normalized with respect to the RSR traffic produced by Policy B-lf. The RSR traffic for Policy B^g-lf is given for two different values of the frequencies $Q_{j \rightarrow i}$ used to compute $X^l[i]$: the first column (“profile”) corresponds to the frequencies values obtained by the average profile program execution (as given in Section 5.2). The second one (“optimal”) corresponds to the real execution frequencies of the program being measured. The comparison between both columns let us conclude how good are the frequency estimations provided by the average profile.

From the table we conclude the following:

no. regs.	program	Policy B-lf	Policy B ^g -lf profile optimal		Policy A ^g -live	Policy A ^g -lvOpt
12	ASM	1.0 (5.94)	1.00	1.00	1.22	0.91
	NROFF	1.0 (1.74)	0.97	0.96	1.25	0.77
	SORT	1.0 (4.14)	0.91	0.72	0.59	0.75
	VPCC	1.0 (5.28)	0.94	0.91	0.73	0.77
	4 P.	1.0 (3.17)	0.94	0.88	0.88	0.77
	SPICE	1.0 (5.77)	1.00	1.00	0.44	0.55
16	ASM	1.0 (5.95)	0.69	0.69	1.08	0.84
	NROFF	1.0 (1.74)	0.82	0.83	1.18	0.72
	SORT	1.0 (4.14)	0.90	0.34	0.39	0.57
	VPCC	1.0 (5.28)	0.94	0.90	0.66	0.71
	4 P.	1.0 (3.17)	0.87	0.71	0.78	0.68
	SPICE	1.0 (5.86)	0.99	0.98	0.42	0.67
24	ASM	1.0 (5.95)	0.42	0.42	0.17	0.67
	NROFF	1.0 (1.74)	0.63	0.63	1.05	0.66
	SORT	1.0 (4.14)	0.12	0.04	0.36	0.53
	VPCC	1.0 (5.28)	0.86	0.83	0.64	0.65
	4 P.	1.0 (3.17)	0.57	0.53	0.65	0.62
	SPICE	1.0 (5.92)	0.97	0.96	0.31	0.66
32	ASM	1.0 (5.95)	0.27	0.27	0.04	0.32
	NROFF	1.0 (1.74)	0.62	0.62	1.04	0.66
	SORT	1.0 (4.14)	0.03	0.03	0.36	0.53
	VPCC	1.0 (5.28)	0.77	0.71	0.61	0.59
	4 P.	1.0 (3.17)	0.49	0.47	0.62	0.57
	SPICE	1.0 (5.95)	0.96	0.96	0.39	0.61

Table 5.8: RSR Traffic Reduction for Policy B^g-lf

1. The estimations provided by the average profile are usually representative of the program behavior. The exception to this rule is SORT because it is a small program and the RSR traffic is concentrated in one out of the two comparison functions, which is not the one executed by the profile input data.
2. For SPICE, the RSR traffic reduction is insignificant (up to 4% for 32 TBP registers; see also Figure 5.31), because SPICE has a heavy register usage and, therefore, functions at the top of the call graph use all the TBP registers available and do not leave any free registers for the rest of the functions.
3. For the four programs, when a small register set is available (12 or 16 TBP registers), the RSR traffic reduction is, on the average, less than 13%. The reason for this is the same as the one given above for SPICE. For larger register sets, the RSR traffic becomes more significant: for 24 TBP registers, 43% of the RSR traffic is eliminated and for 32, 51%. Also, notice that the RSR traffic becomes smaller

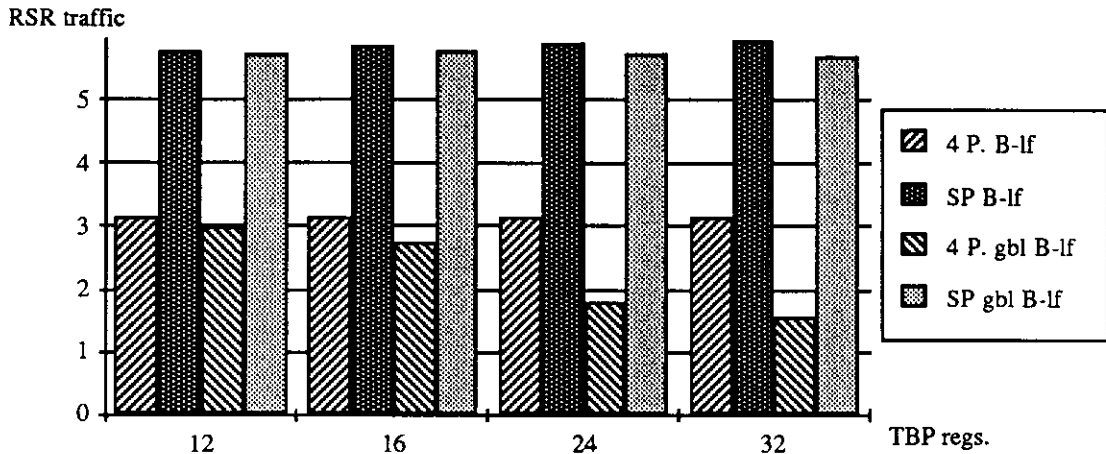


Figure 5.31: RSR Traffic for Policy B^s-lf

for larger register sets (see also Figure 5.31).

Global Policy B-lf versus Global Policies A-live and A-lvOpt

We now compare Policy B^s-lf with Policies A^s-live and A^s-lvOpt. The behavior of the global policies does not have the same pattern as their corresponding intra-procedural policies. The RSR traffic generated by Policies A^s-live and A^s-lvOpt normalized with respect to Policy B-lf is also shown in Table 5.8. If we compare these policies with Policy B^s-lf we conclude that:

- For SPICE and VPCC, Policy B^s-lf generates more RSR traffic than both Policies A^s-live and A^s-lvOpt. For SPICE, Policy B^s-lf has between 225% (for 6 TBP registers) and 318% (for 24) of the RSR traffic generated by Policy A^s-live. For VPCC, Policy B^s-lf has between 129% (for 6) and 141% (for 12).
- For NROFF and SORT as well as the average of the four programs, Policy B^s-lf generates the least RSR traffic when 24 or 32 TBP registers are available, but not for 12 or 16. For instance, on the average, Policy B^s-lf has 107%, 113%, 87%, and 78% of the traffic generated by Policy A^s-live when 12, 16, 24, and 32 TBP registers are available.
- For ASM, the opposite is true: Policy B^s-lf generates the least RSR traffic when 12 or 16 TBP registers are available, but not for 24 or 32. For instance, Policy B^s-lf has 82%, 64%, 241%, and 704% of the traffic generated by Policy A^s-live when 12, 16, 24, and 32 TBP registers are available.

Therefore, the performance of the static intra and inter-procedural optimizations depends notably on the program's characteristics. Our measurements have shown that it

is impossible to select a unique static policy that performs well for any program and register-set configuration. The optimizer should have the option of selecting the most appropriate policy based on the average profile information gathered by a few program executions. With this profile information, the optimizer performs the three static optimizations and selects the one which generates the least RSR traffic for the average of the profiled programs as the one to be used.

Summary

Our measurements have shown that Policy B^g-lf has 94%, 87%, 57%, and 49% of the RSR traffic generated by Policy B-lf for the average of the four programs when 12, 16, 24, and 32 TBP registers are available and up to 94% for SPICE. Compared with Policies A^g-live and A^g-lvOpt, our measurements have shown that it is impossible to select a unique static policy which performs well for any program in a given register-set configuration. Thus, the optimizer should have the option of selecting the most appropriate policy based on the average profile information.

5.4.4 Global Policy G-lf

The Global Policy G with Leaf Functions (G^g-lf) differs from the previous inter-procedural optimizations in that no RSR instruction has to be eliminated, because the unnecessary RSR traffic is already eliminated by the dynamic behavior of the policy (see Section 3.3). The goal of this optimization is to implement a *real* round robin register assignment for the functions in the same path in the call graph, like the one shown in Figure 5.32. The reason for this is that we would like that, whenever it is possible, the set of registers defined for a function be different from the set of registers defined for the functions that this one might call. In this case, the probability of having to save a register is reduced.

Program execution usually spans a small set of functions [Patt85a]. This is a direct consequence of program locality. Thus, if the set of functions being executed at a given time uses disjoint registers, then no register saving/restoring will be performed. This is equivalent to a multiple-window scheme because the register set is shared by several functions without any register saving/restoring until the register set overflows (i.e., a new function is called such that it uses a register already being used).

Our previous measurements [Hugu85a, Section 3.8] show that, on the average, 54% of the functions execute with a nesting depth of 2. That is, if all the caller-callee pairs have disjoint registers assigned, then at least 54% of the executed functions will not have to perform any register saving/restoring. This number increases up to 85% if the register set is large enough to always keep all variables allocated for 3-consecutively-called functions. We will show that when only the *immediate* ancestors are considered, the RSR traffic generated is less than when the function's immediate ancestors and their immediate ancestors are considered.

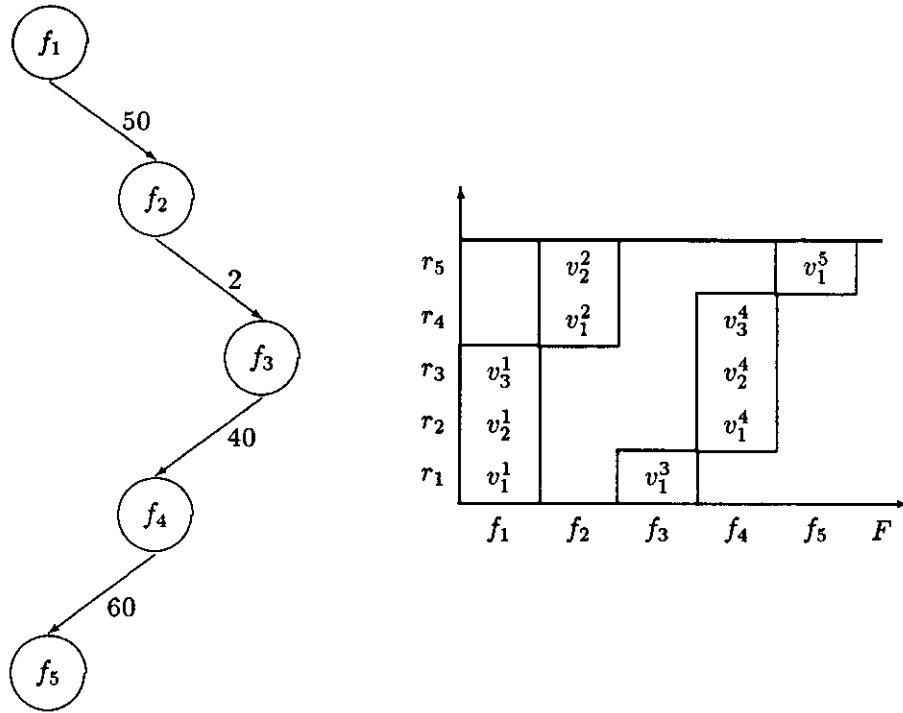


Figure 5.32: Disjoint Register Assignment for Policy $G^s\text{-lf}$

In this section we first introduce how we perform register assignment using a small example; second, we present a one-pass algorithm to perform register assignment; third, we show the RSR traffic reduction obtained with inter-procedural optimization; and, finally, we comment on the advantages of using Policy $G^s\text{-lf}$ versus the other static global policies.

An Example of Register Assignment

Figure 5.33 shows the call graph of a program with 6 functions ($F = \{f_1, f_2, \dots, f_6\}$). The variables selected for allocation by the intra-procedural allocator are also shown in the figure. The functions in the call graph are visited from the root (the *main* function for C programs) to the leaves in such a way that when we visit a function, we have already visited all its callers. This is the same order used by Policy $B^s\text{-lf}$ (see Algorithm 3). In this example, the visits are performed in the order given by the function subindexes.

Let us assume that there are 5 TBP registers ($\{r_1, r_2, \dots, r_5\}$). Registers are assigned as follows:

- The 3 variables of f_1 are assigned to registers r_1, r_2 , and r_3 .
- For f_2 , a disjoint assignment is possible because only two more registers are required (see Figure 5.34).

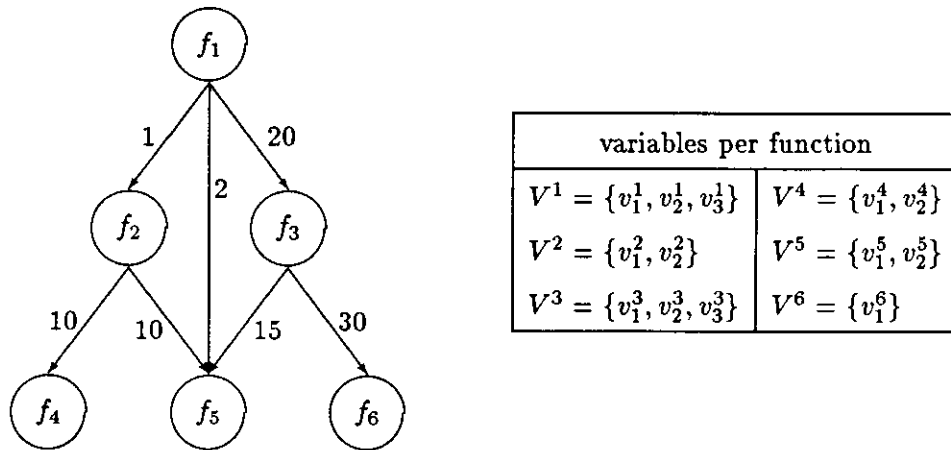


Figure 5.33: An Example of Register Assignment for Policy $G^{\text{e-lf}}$

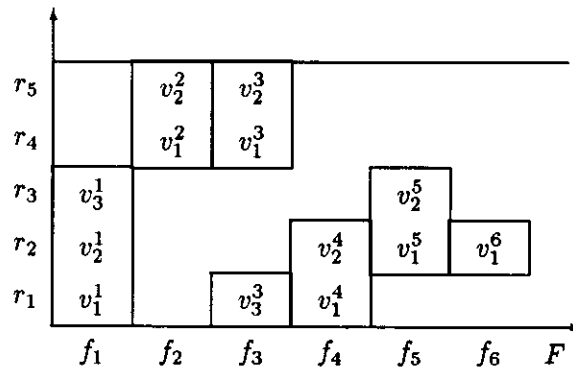


Figure 5.34: Variable-to-Register Assignment

- For f_3 , this is not possible because 3 registers are required. Thus, two variables are assigned to the two free registers r_4 and r_5 and the third variable to one of the registers already used by the caller of f_3 (f_1).
- For f_4 , two free registers are also available since we only consider the immediate ancestors.
- For f_5 , no free registers are available because all five registers are taken by its callers as we can see in Figure 5.34. In this case, we select the registers used in the lowest execution path. For our example, this corresponds to the call from f_1 and, therefore, variables v_1^5 and v_2^5 are assigned to registers r_2 and r_3 .
- For f_6 , there is no problem either because two registers are free.

A One-Pass Algorithm to Perform Register Assignment

Four different approaches have been evaluated to perform register assignment. These approaches are based on the following two factors:

1. The number of ancestors to consider, 1 or 2. In the former case, only the immediate ancestors are considered for obtaining a disjoint register assignment. In the latter, the immediate ancestors and their immediate ancestors are considered.
2. The functions in a cycle (if any). In one case, we ignore the functions which are in a cycle. The disjoint register assignment is performed only for the registers already assigned to the ancestors. In the other, if a function is inside a cycle, all the functions inside the cycle are considered for obtaining a disjoint register assignment in addition to the ancestors.

We now present only one algorithm for one of the approaches. This corresponds to the case of taking into account the functions in the same cycle and the immediate ancestors for obtaining a disjoint register assignment. This approach has been selected because it is the one that generates the least RSR traffic, as we will discuss below. The other algorithms could be easily deduced from this one.

Algorithm 4 Register Assignment for Policy G^s-lf.

Inputs: The call graph G , the set of defined functions (F), the sets of selected variables for allocation per function (V^j), the average profile frequency of each call ($Q_{t \rightarrow j}$) and of each function ($Q^j = \sum_{\forall t, t \rightarrow j} Q_{t \rightarrow j}$), and a p -element array O with an acyclic numbering of the nodes such that in increasing order, O always contains the caller before the callee (except for recursive functions). The recursive functions should have been taken into account when the O array is built.

Outputs: The set of registers defined per function (M^j) which might be saved if the registers have already been used in the exterior levels and the matrix A that indicates the register assignment per function. As for Policy B^s-lf, the matrix A is not really necessary because any variable in V^j can be assigned to any register in M^j .

Locals: A set U^j per function to indicate the registers defined by the immediate ancestors of function f_j and if f_j is in a cycle, the registers defined by the other functions in the cycle; the definition $\text{REG_USED}(r_i, M^j)$ as given in Algorithm 1, and an array $X = \{X[1], \dots, X[n]\}$ used to estimate the usage of each register by the immediate ancestors of the function being processed at a given time and by the functions in the same cycle (if any).

Method: Apply algorithm given in Figure 5.29. As for Algorithm 1, we assume that local scalar variables assigned to TBD registers for leaf functions have already been removed from V^j . \square


```

for  $f_j = O[1], O[2], \dots, O[p]$  do
   $U^j \leftarrow \bigcup_{t \rightarrow j} M^t$ 
  if  $f_j$  is inside a cycle then
    for each  $f_t \in F$  in the same cycle do  $U^j \leftarrow U^j \cup M^t$  od
  fi
   $q \leftarrow |U^j|$ 
   $m \leftarrow |V^j|$ 
  if  $m \leq n - q$  then
    (* at least  $m$  registers not used by functions at previous depth *)
    let  $v_1^j, v_2^j, \dots, v_m^j$  be assigned to  $r_{i_1}, r_{i_2}, \dots, r_{i_m}$  such that
       $i_1 = [1 + \max(s, \forall r_s \in U^j)] \bmod n$ 
      for  $k = 2, \dots, m$  do
        if  $r_{(1+i_{k-1}) \bmod n} \notin U^j$  then
           $i_k = (1 + i_{k-1}) \bmod n$ 
        else
           $i_k = (s + i_{k-1}) \bmod n$  such that
             $r_{(1+i_{k-1}) \bmod n}, \dots, r_{(s-1+i_{k-1}) \bmod n} \in U^j$  and
             $r_{(s+i_{k-1}) \bmod n} \notin U^j$ 
        fi
      od
       $a_{i_1, j} \leftarrow v_1^j, \dots, a_{i_m, j} \leftarrow v_m^j$ 
       $M^j \leftarrow \{r_{i_1}, r_{i_2}, \dots, r_{i_m}\}$ 
    else (* ( $n - q$ ) registers not used by functions at previous depth *)
      for  $i = 1, \dots, n$  do
         $X[i] = 0$ 
        for each  $f_t \in F$  such that  $t \rightarrow j$  do
          if  $Q_{t \rightarrow j} \neq 0$  then
             $X[i] \leftarrow X[i] + \text{REG\_USED}(r_i, M^t) Q_{t \rightarrow j}$ 
          else
             $X[i] \leftarrow X[i] + \text{REG\_USED}(r_i, M^t)$ 
          fi
        od
        if  $f_j$  is inside a cycle then
          for each  $f_t \in F$  in the same cycle do
            if  $Q_{t \rightarrow j} \neq 0$  then
               $X[i] \leftarrow X[i] + \text{REG\_USED}(r_i, M^t) Q^t$ 
            else
               $X[i] \leftarrow X[i] + \text{REG\_USED}(r_i, M^t)$ 
            fi
          od
        fi
      od
      sort  $X$  such that  $X[i_1] \leq X[i_2] \leq \dots \leq X[i_m] \leq \dots \leq X[i_n]$ 
      (* notice that  $X[i_1] = \dots = X[i_{n-q}] = 0$  *)
       $a_{i_1, j} \leftarrow v_1^j, \dots, a_{i_m, j} \leftarrow v_m^j$ 
       $M^j \leftarrow \{r_{i_1}, r_{i_2}, \dots, r_{i_m}\}$ 
    fi
  od
fi
od

```

Figure 5.35: Register Assignment for Policy $G^{\mathcal{K}}$ -lf

If we compare this algorithm with the algorithms for Policies A^g-live, A^g-lvOpt, and B^g-lf, we can observe its simplicity.

The return-address optimization discussed in Section 5.3 can be merged with the Policy G^g-lf optimization. A few TBP registers can be used as link registers so that the optimizer performs a disjoint assignment for the functions which are in the same execution path. When program execution spans a small set of functions, it is possible that no register saving/restoring has to be performed for these link registers. Moreover, the saving instructions for the link registers at the bottom of the call graph can be eliminated, as it has been indicated in Section 5.3. However, this feature has not been considered in the measurements presented next because we would like to compare only the RSR traffic generated by Policy G^g-lf with the one produced by the global static policies.

Evaluation of the RSR Traffic Reduction

Table 5.9 shows the RSR traffic generated by the four approaches mentioned above normalized with respect to Policy G-lf. From the table we conclude that it is better:

1. To obtain a disjoint register assignment with the immediate ancestors rather than with these and their ancestors. When two ancestors are considered, more RSR traffic is usually generated. The exceptions to the rule are ASM and VPCC when 24 or 32 TBP registers are available.
2. To consider the functions in a cycle to perform a disjoint register assignment. The programs which benefit the most are the ones which have a large percentage of recursive functions like NROFF and VPCC (see Table 5.3).

Thus, the RSR traffic generated by the approach which considers the immediate ancestors and the functions in cycles is taken as the RSR traffic produced by the “standard” Policy G^g-lf. Notice that this approach is the one given in Algorithm 4.

Policy G^g-lf has between 42% (for 16 and 32 TBP registers) and 72% (for 12) of the RSR traffic generated by Policy G-lf for the average of the four programs and between 72% (for 24) and 93% (for 16) for SPICE (see Figure 5.36). The only program whose RSR traffic is not reduced by the inter-procedural optimizer is NROFF. When 24 or 32 TBP registers are available, Policy G^g-lf has 140% or 143% of the RSR traffic generated by Policy G-lf. Since the absolute traffic generated was almost insignificant (0.04 registers per function for Policy G-lf versus 0.06 for Policy G^g-lf), we decided not to look into this anomaly.

Global Dynamic Policy versus Global Static Policies

Table 5.10 shows the RSR traffic generated by the global policies normalized with respect to Policy G^g-lf. The dynamic Policy G^g-lf generates the least RSR traffic. For the

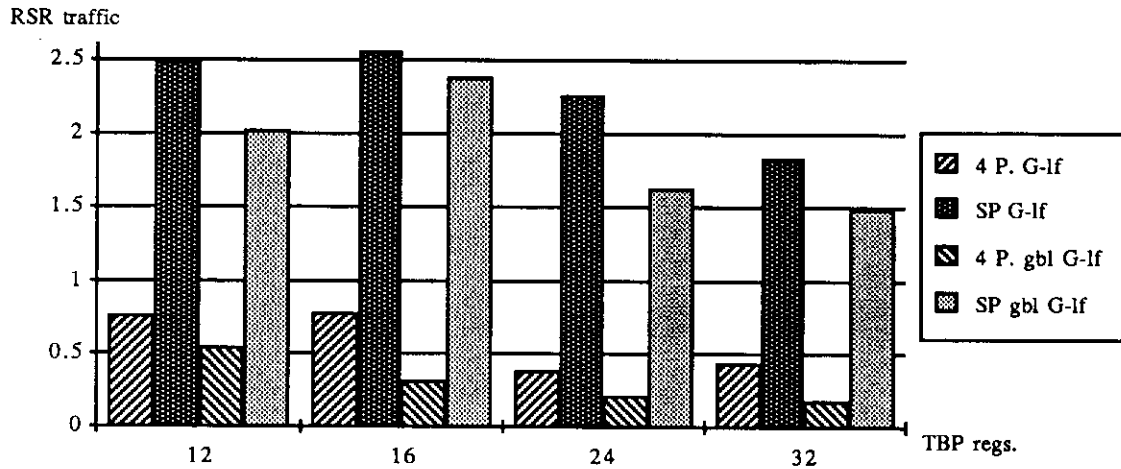


Figure 5.36: RSR Traffic for Policy G-lf

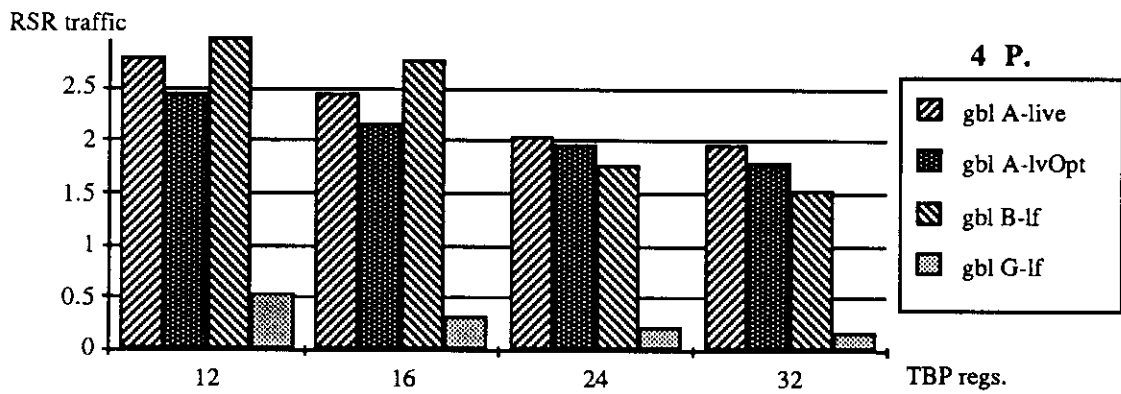


Figure 5.37: Inter-Procedural Optimized RSR Traffic for the Four Programs

no. regs.	program	Policy G-lf	Policy G ^s -lf			
			without ftns. in cycle		with ftns. in cycle	
			1 anc.	2 anc.	1 anc.	2 anc.
12	ASM	1.0 (2.31)	0.84	0.73	0.84	0.73
	NROFF	1.0 (0.13)	1.04	1.23	0.85	1.29
	SORT	1.0 (0.95)	0.73	0.95	0.73	0.95
	VPCC	1.0 (1.87)	0.68	0.99	0.67	0.92
	4 P.	1.0 (0.76)	0.74	0.97	0.72	0.94
	SPICE	1.0 (2.49)	0.78	0.83	0.81	0.92
16	ASM	1.0 (2.09)	0.55	0.72	0.55	0.72
	NROFF	1.0 (0.08)	1.56	1.58	0.96	1.59
	SORT	1.0 (1.16)	0.17	0.17	0.17	0.17
	VPCC	1.0 (1.89)	0.47	0.61	0.48	0.59
	4 P.	1.0 (0.78)	0.45	0.54	0.42	0.53
	SPICE	1.0 (2.55)	0.79	0.85	0.93	1.00
24	ASM	1.0 (1.98)	0.39	0.26	0.39	0.26
	NROFF	1.0 (0.04)	1.66	2.14	1.40	2.39
	SORT	1.0 (0.17)	0.50	0.50	0.50	0.50
	VPCC	1.0 (1.13)	0.60	0.59	0.58	0.58
	4 P.	1.0 (0.39)	0.61	0.60	0.58	0.61
	SPICE	1.0 (2.27)	0.77	0.82	0.72	0.85
32	ASM	1.0 (1.97)	0.17	0.15	0.17	0.15
	NROFF	1.0 (0.04)	1.77	1.72	1.43	2.05
	SORT	1.0 (0.08)	0.99	0.99	0.99	0.99
	VPCC	1.0 (1.48)	0.40	0.39	0.39	0.40
	4 P.	1.0 (0.44)	0.45	0.43	0.42	0.45
	SPICE	1.0 (1.84)	0.89	1.10	0.82	1.17

Table 5.9: RSR Traffic Reduction for Policy G^s-lf

average of the four programs (see Figure 5.37), the best static policy has between 445% (Policy A^s-lvOpt for 12 TBP registers) and 816% (Policy B^s-lf for 32) of the RSR traffic generated by Policy G^s-lf. For SPICE (see Figure 5.38), the best static policy is Policy A^s-live for any register set configuration. Policy A^s-live has between 103% (for 16) and 155% (for 32) of the RSR traffic generated by Policy G^s-lf. The difference on RSR traffic between the dynamic Policy G^s-lf and the static global policies is smaller than with the other four programs and their average because the static Policy A^s-live eliminates most of the RSR traffic for SPICE since most of this traffic is at the bottom of the call graph (96% as we mentioned in Subsection 5.4.1).

One additional advantage of Policy G^s-lf with respect to the global static policies is that the RSR traffic reduction obtained is not dependent of the program structure. Let us discuss why this happens for Policies A^s-live and A^s-lvOpt. Analogous reasons could

no. regs.	program	Policy G ⁸ -lf	Policy A ⁸ -live	Policy A ⁸ -lvOpt	Policy B ⁸ -lf
12	ASM	1.0 (1.95)	3.73	2.77	3.07
	NROFF	1.0 (0.11)	19.73	12.18	15.27
	SORT	1.0 (0.69)	3.54	4.52	5.45
	VPCC	1.0 (1.26)	3.04	3.21	3.93
	4 P.	1.0 (0.55)	5.09	4.45	5.44
	SPICE	1.0 (2.02)	1.27	1.57	2.86
16	ASM	1.0 (1.14)	5.66	4.39	3.60
	NROFF	1.0 (0.07)	29.29	17.86	20.43
	SORT	1.0 (0.20)	8.00	11.80	18.60
	VPCC	1.0 (0.90)	3.89	4.17	5.50
	4 P.	1.0 (0.32)	7.69	6.78	8.66
	SPICE	1.0 (2.38)	1.03	1.66	2.46
24	ASM	1.0 (0.78)	1.32	5.13	3.18
	NROFF	1.0 (0.06)	30.33	19.17	18.17
	SORT	1.0 (0.08)	18.38	27.38	6.50
	VPCC	1.0 (0.65)	5.20	5.29	7.02
	4 P.	1.0 (0.22)	9.36	9.00	8.14
	SPICE	1.0 (1.64)	1.10	2.39	3.61
32	ASM	1.0 (0.34)	0.68	5.59	4.76
	NROFF	1.0 (0.06)	30.17	19.17	17.83
	SORT	1.0 (0.08)	18.38	27.38	1.63
	VPCC	1.0 (0.58)	5.55	5.34	7.02
	4 P.	1.0 (0.19)	10.42	9.53	8.16
	SPICE	1.0 (1.50)	1.55	2.41	3.90

Table 5.10: RSR Traffic Reduction for the Global Policies

be given for Policy B⁸-lf.

The number of register saving/restoring instructions eliminated for a specific function f_j depends on the number of registers which are not used for the functions which might be called. If these functions have already all the registers assigned (because, for instance, one of them needs all of them¹⁴), then the optimizer will not be able to remove any RSR instruction for either f_j or any of its ancestors. Thus, if the RSR traffic is generated mainly by f_j and its ancestors, a small RSR traffic reduction will be obtained.

Moreover, if the program has a set of functions which are called recursively and they are frequently used (i.e., they generate a significant part of the RSR traffic), a small traffic reduction will again be obtained because the optimizer cannot eliminate all the

¹⁴Notice that with our scheme the number of variables to allocate is decided per function. An alternative approach would be to limit the maximum number of variables to be allocated per function and to start the compilation process again rather than to continue it with the assembler (see Figure 5.2).

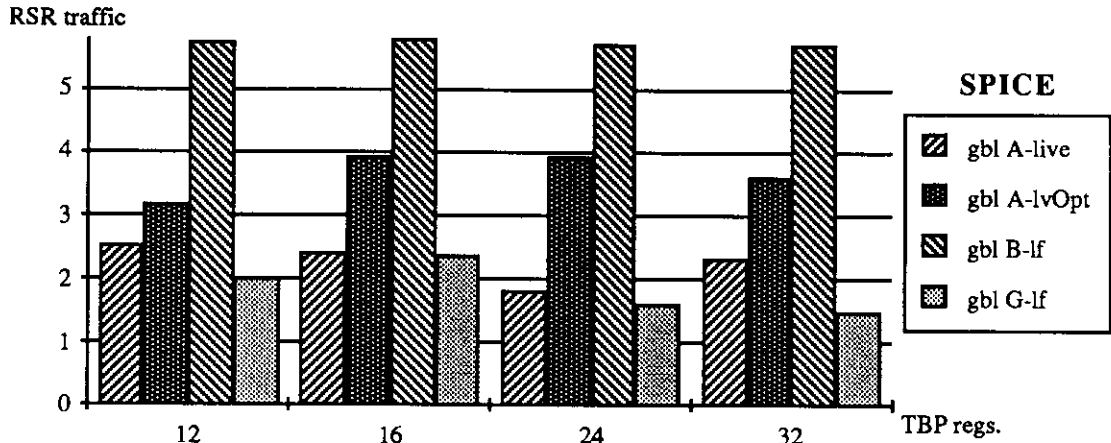


Figure 5.38: Inter-Procedural Optimized RSR Traffic for SPICE

RSR instructions for these functions.¹⁵ This is the case for NROFF and VPCC. For instance, when 16 TBP registers are available, 68% of the RSR traffic generated by Policy A-lvOpt for NROFF is produced by functions in cycles and, therefore, it cannot be eliminated; for VPCC, this percentage is 67%. Therefore, the RSR traffic reduction for the global static policies depends on the program structure.

Policy G^s-lf does not have any of these two limitations. While Policy B^s-lf eliminates the RSR traffic which occurs at the top of the call graph and Policies A^s-live and A^s-lvOpt, at the bottom, Policy G^s-lf eliminates the RSR traffic from any function in the call graph depending only on the dynamic register usage, but not on its location in the call graph. Moreover, Policy G^s-lf can even eliminate the RSR traffic generated by functions in a cycle. The only program's characteristic that influences the performance of Policy G^s-lf (as well as of Policies A^s-live, A^s-lvOpt, and B^s-lf) is the register usage required by the program. The more registers are required by the program, the less RSR traffic reduction is obtained. But this is a perfectly understandable factor.

In conclusion, Policy G^s-lf not only generates the least RSR traffic, but it also simplifies the compiler implementation and reduces the RSR traffic independently of where the functions that generate most of the RSR traffic are located in the call graph, and of the percentage of recursive functions in the program.

5.4.5 Summary

This section has presented and discussed four new algorithms to perform register assignment in such a way that the RSR traffic is reduced. These algorithms have to be applied to the whole program once the call graph is known. The complexity added to

¹⁵We said *all* because the optimizer can in fact eliminate the RSR traffic generated by the registers which are dead for all the calls to functions in the cycle.

the compilation process is justified by the RSR traffic reduction obtained.

Three of these algorithms are for the static policies. Policies A^g-live and A^g-lvOpt eliminate the register saving/restoring instructions performed at a specific call when the registers being saved/restored are not used by any of the functions that might be called from this point. Policy B^g-lf eliminates the register saving/restoring instructions performed at function entry and return when the registers being saved/restored are not used by any of the functions that might call this function. Thus, these policies find a disjoint register assignment such that the maximum number of register saving/restoring instructions can be eliminated.

These inter-procedural optimizations reduce the RSR traffic of their respective intra-procedural counterparts:

- Policy A^g-live has between 46% (for 12 TBP registers) and 32% (for 32) of the RSR traffic generated by Policy A-live for the average of the four programs and between 26% and 11% for SPICE (see Table 5.6 and Figure 5.23).
- Policy A^g-lvOpt has between 64% (for 12 TBP registers) and 47% (for 32) of the RSR traffic produced by Policy A-lvOpt for the average of the four programs and between 58% and 42% for SPICE (see Table 5.7 and Figure 5.23).
- Policy B^g-lf has 94%, 87%, 57%, and 49% of the RSR traffic generated by Policy B-lf for the average of the four programs when 12, 16, 24, and 32 TBP registers are available and up to 94% for SPICE (see Table 5.8 and Figure 5.31).

The static inter-procedural optimizations not only generate less RSR traffic than their respective intra-procedural optimizations, but also the RSR traffic produced decreases for larger register sets. In Chapter 4 we mentioned that when intra-procedural register allocation and assignment is performed, there is no need for having more than 12 TBP registers for non-numeric applications since the decrease in the local scalar traffic is balanced with the increase in the RSR traffic. This is not case when inter-procedural register assignment is performed as we can see comparing Figures 4.17 and 5.37. Therefore, a larger register set can be efficiently utilized when the compiler uses Policies A^g-live, A^g-lvOpt, or B^g-lf.

Our measurements have not shown which is the best static policy to be used for any program and for a given register-set configuration, because the RSR traffic reduction provided by these policies closely depends on the characteristics of the program—its maximum depth, where most of its RSR traffic is generated, the maximum number of registers allocated by the intra-procedural optimizer to the functions at the bottom of the call graph for Policies A^g-live and A^g-lvOpt or to the top for Policy B^g-lf, etc. Consequently, it is impossible to select a unique static policy which performs well for any program for a given register-set configuration. The optimizer should have all three static policies available so that the RSR traffic generated by each policy is computed, based on the profile information, and the most appropriate one is selected.

The fourth algorithm is for the dynamic Policy G. Policy G^s-lf assigns registers in a round-robin fashion for the functions in the same path, whenever this is possible. The optimizer, in this case, does not have to eliminate any register saving/restoring instruction because this is already being done by the dynamic behavior of the policy itself. Policy G^s-lf has between 42% (for 16 and 32 TBP registers) and 72% (for 12) of the RSR traffic generated by Policy G-lf for the average of the four programs and between 72% (for 24) and 93% (for 16) for SPICE (see Table 5.9 and Figure 5.36).

Policy G^s-lf generates less RSR traffic than any of the static inter-procedural optimizations (see Table 5.10 and Figures 5.37 and 5.38). The best static policy has between 445% and 816% of the RSR traffic generated by Policy G^s-lf for the average of the four programs, and between 103% and 155% for SPICE.

Moreover, Policy G^s-lf has three more advantages with respect to the static policies:

1. While Policy B^s-lf eliminates the RSR traffic which occurs at the top of the call graph and Policies A^s-live and A^s-lvOpt, at the bottom, Policy G^s-lf eliminates the RSR traffic from any function in the call graph depending only on the dynamic register usage, but not on its location in the call graph.
2. While the static policies cannot eliminate completely the RSR traffic generated by the recursive functions in a program, the dynamic Policy G^s-lf can eliminate it depending on the register usage in the cycle.
3. While all three static policies have to be implemented so that the optimizer can select the most convenient one to be used for a specific program, this is not necessary for the dynamic Policy G^s-lf because it always generates the least traffic. Moreover, Algorithm 4 for Policy G^s-lf is the easiest to implement.

In conclusion, Policy G^s-lf is our best approach to simplify compiler complexity and to eliminate the RSR traffic as much as possible, independently of the characteristics of the program.

5.5 General-Purpose Register Set Partition

In the previous sections three types of inter-procedural optimizations have been presented to reduce the data memory traffic: for global scalar traffic, for the return-address traffic, and for the RSR traffic. As we have seen in Section 4.5, their contribution towards the overall data memory traffic is not equally weighted (see Figure 4.28 or 5.1). For this reason, since the total number of general-purpose registers is fixed (and we would like to keep this number small), it is necessary to consider the best usage of these registers towards the overall data memory traffic reduction.

Seven different register-set configurations are evaluated. One configuration is for a 16-register file and one for a 64. For a 32-register file, five different partitions have

been evaluated, which corresponds to several register distributions among the different optimizations (e.g., more registers for global scalar variables and fewer for locals).

1. 12 to-be-preserved registers, 0 registers for global scalar variables, and 1 link register (denoted by $12\ TBP + 0\ GS + 1\ LR$).
2. 12 to-be-preserved registers, 12 registers for global scalar variables, and 2 link registers (denoted by $12\ TBP + 12\ GS + 2\ LR$).
3. 12 to-be-preserved registers, 8 registers for global scalar variables, and 6 link registers (denoted by $12\ TBP + 8\ GS + 6\ LR$).
4. 16 to-be-preserved registers, 8 registers for global scalar variables, and 2 link registers (denoted by $16\ TBP + 8\ GS + 2\ LR$).
5. 16 to-be-preserved registers, 6 registers for global scalar variables, and 4 link registers (denoted by $16\ TBP + 6\ GS + 4\ LR$).
6. 24 to-be-preserved registers, 2 registers for global scalar variables, and 1 link register (denoted by $24\ TBP + 2\ GS + 1\ LR$).
7. 32 to-be-preserved registers, 16 registers for global scalar variables, and 8 link registers (denoted by $32\ TBP + 16\ GS + 8\ LR$).

For the first configuration, the RSR traffic generated by the intra-procedural Policies B-lf and G-lf is used. Policy B-lf has been selected because it generates the least RSR traffic among the static policies (see Figures 4.14 and 4.21). This configuration is used as the comparison base for the inter-procedural optimizations. For the other configurations, the best static global policy is selected for each number of TBP registers. Thus,

- For the four programs, the RSR traffic generated by Policy A^s-lvOpt is selected for 12 and 16 TBP registers and by Policy B^s-lf for 24 and 32 (see Figure 5.37).
- For SPICE, the RSR traffic generated by Policy A^s-live is selected for any number of TBP registers (see Figure 5.38).

In this section we first compare the overall data memory traffic generated by the best static global policy and by the dynamic Policy G^s-lf for the seven register-set configurations given above. Afterwards, we evaluate the speed-up factor that we might obtain for a RISC-like machine using the model described in Section 4.6.

5.5.1 Overall Data Memory Traffic Reduction for the Four Programs

The data memory traffic generated by the best static global policy for the four programs is shown in Figure 5.39 and by Policy G^s-lf, in Figure 5.40. For a 32-register file, the configuration $16\ TBP + 8\ GS + 2\ LR$ produces the least traffic. When there is no

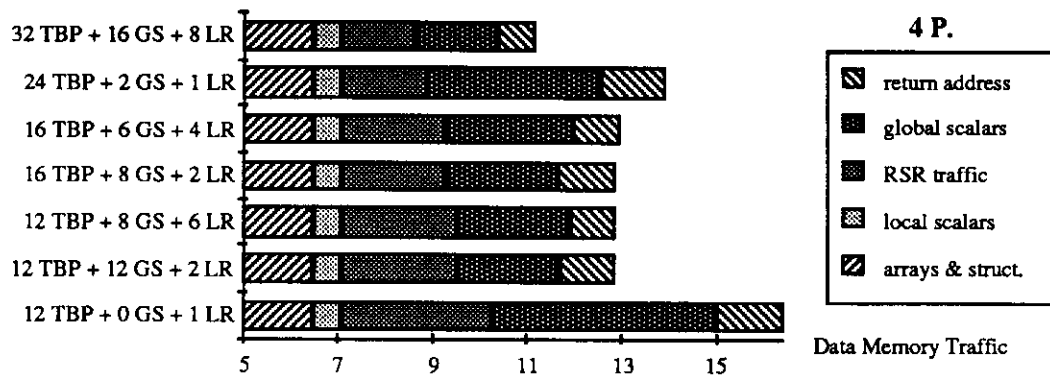


Figure 5.39: Data Memory Traffic for the Four Programs with the Best Static Global Policy

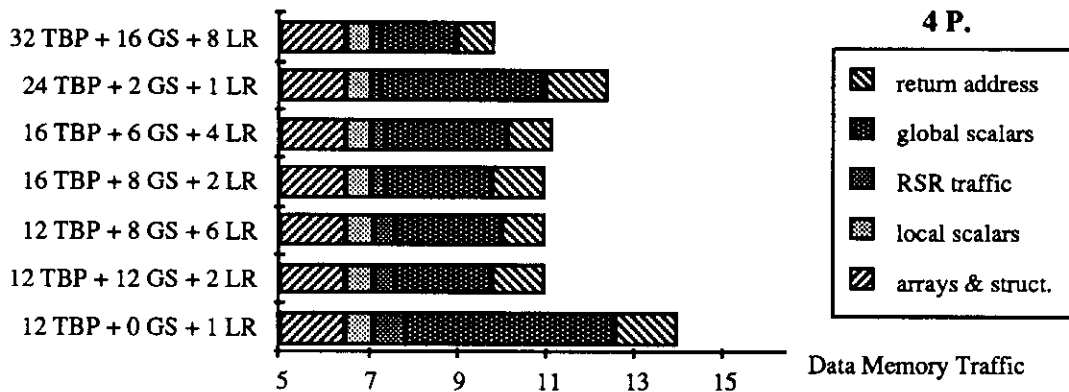


Figure 5.40: Data Memory Traffic for the Four Programs with Policy G^s-lf

architectural support, the inter-procedural optimizer generates 78% of the data memory traffic produced by the intra-procedural optimizer with Policy B-lf. With both architectural and compiler support, the data memory traffic generated is 67% with respect to the intra-procedural optimizer with Policy B-lf, 79% with respect to Policy G-lf, and 86% with respect to the inter-procedural optimizer with the best static policy.

Since the global scalar traffic is twice the traffic caused by the return address (see Figures 5.3 and 5.9), it is better to use registers for globals rather than for link registers. As we can see in Figures 5.39 and 5.40, the configurations *16 TBP + 6 GS + 4 LR* and *12 TBP + 8 GS + 6 LR* generate more traffic than the *16 TBP + 8 GS + 2 LR* and *12 TBP + 12 GS + 2 LR*, respectively.

The RSR traffic reduction obtained by increasing the number of TBP registers from 12 to 16 is equivalent to the global scalar traffic reduction obtained by increasing the number of registers for globals from 8 to 12 (see Figures 5.37 and 5.3). However, when the number of TBP registers is increased from 16 to 24 and, consequently, the number of registers for globals is reduced from 8 to 2, more data memory traffic is generated, 9% for the static policies and 13% for the dynamic Policy G^s-lf, because for the four programs

almost no new local scalars are assigned to registers even though the number of TBP registers is increased beyond 12 (see Figure A.1). The RSR traffic reduction when the number of TBP registers is increased from 16 to 24 (see Figure 5.37) is much smaller than the global scalar traffic reduction obtained with 8 registers rather than 2 (see Figure 5.3). Thus, for the four programs, it is better to limit the number of TBP registers to 16 (out of 32). This is not the case for SPICE, as we will discuss in Subsection 5.5.2.

If the inter-procedural optimizer only reduces the RSR traffic (i.e., no registers for global scalars and only 1 link register), the overall data memory traffic generated by Policy G^s-lf is 96% of the one produced by Policy G-lf and by the best static policy, 91% of Policy B-lf. Therefore, the interprocedural optimizer has also to consider global scalars and the traffic caused by the return address to obtain a more significant data memory traffic reduction (79% and 78% as we mentioned above).

For a 64-register file, when there is no architectural support, the inter-procedural optimizer generates 68% of the data memory traffic produced by the intra-procedural optimizer with Policy B-lf. With both architectural and compiler support, the data memory traffic generated is 60% with respect to the intra-procedural optimizer with Policy B-lf, 70% with respect to Policy G-lf, and 88% with respect to the inter-procedural optimizer with the best static policy.

When the register file size is doubled (from 32 to 64), the overall data memory traffic generated with the configuration $32\ TBP + 16\ GS + 8\ LR$ is 87% of the traffic produced with a configuration $16\ TBP + 8\ GS + 2\ LR$ when only compiler support is provided and 89% when both architectural and compiler support are provided. Thus, when inter-procedural optimizations are performed, there is still a significant data traffic reduction for larger register sets. This is not the case for an intra-procedural optimizer (see Figure 4.26). Although the selection of a 64-register file could be justified because of the data memory traffic reduction, there are some other considerations to be taken into account:

1. The number of bits required in the instruction format to address a register.
2. The number of bits required for the mask. For 32 TBP registers a 32-bit mask is required. If instructions are 32-bit wide, two instructions will be required for every saving/restoring instruction to load the 32-bit mask.
3. The access time to the register file. Since larger register files have a negative effect on the processor cycle time [Sher84, Ditz87b], it might not be convenient to have such a large register file (see Subsection 2.1.1).
4. The additional area required in the chip for a VLSI implementation. For the dynamic Policy G^s-lf, not only the area for the general-purpose registers has to be considered, but also the area required for the register pointers (see Section 3.3).
5. The performance of a cache memory. It might be that with a 32-register file most of the conflicts to the cache memory have been eliminated. Thus, the register file

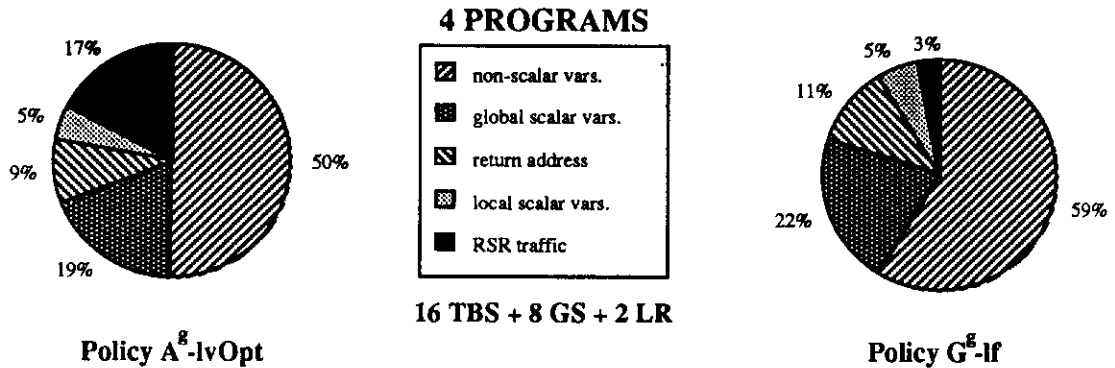


Figure 5.41: Remaining Data Memory Traffic for the Four Programs

increase from 32 to 64 might not have a significant influence on the performance of the overall system. However, this is an open topic for discussion (see Section 6.2).

Therefore, the data memory traffic is not the only factor that the designer has to consider to make his/her final choice.

In conclusion, the inter-procedural optimizer with the dynamic Policy G^S-lf is our best approach to reduce significantly the overall data memory traffic. For a 32-register file, the inter-procedural optimizer with Policy G^S-lf has 67% of the data memory traffic generated by an intra-procedural optimizer with Policy B-lf, 79% of the traffic generated by an intra-procedural optimizer with Policy G-lf, and 86% of the traffic produced by an inter-procedural optimizer with the best static policy. Figure 5.41 shows the remaining data memory traffic after performing inter-procedural optimizations (compare this figure with Figure 5.1).

5.5.2 Overall Data Memory Traffic Reduction for SPICE

In this section the overall data memory traffic reduction obtained for the seven register-set configurations mentioned above is evaluated for SPICE. The reason for separating completely the results for the four programs with the ones for SPICE is that when only compiler support is provided, the data memory traffic generated is the same as the one produced when both architectural and compiler support are. This is a direct consequence of the heavy register usage required by SPICE and caused by its floating-point variables (see Section 4.3) and the good performance of the static Policy A^S-live because the RSR traffic is concentrated at the bottom of the call graph (see Figure 5.14).

The best approach for SPICE is to have as many TBP registers as possible for a given register-set configuration. The data memory traffic generated by the best static global policy for SPICE is shown in Figure 5.42 and by Policy G^S-lf, in Figure 5.43. For a 32-register file, the best register-set partition is to have 24 TBP registers, 2 registers

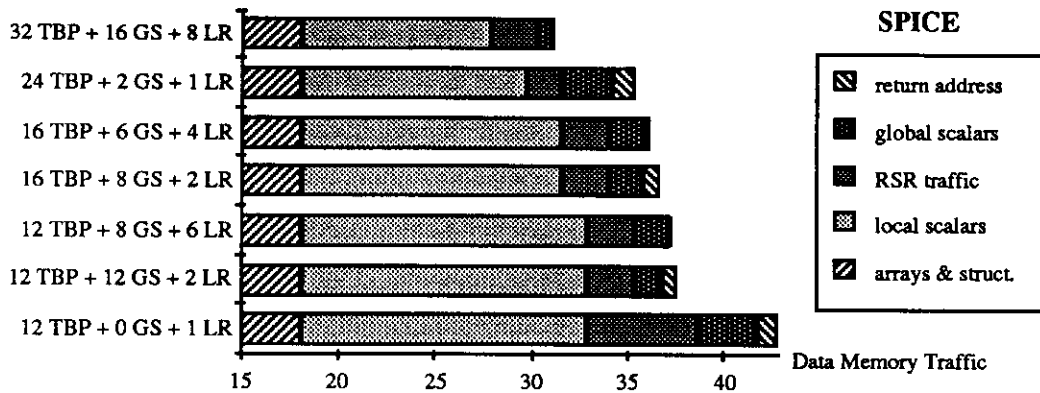


Figure 5.42: Data Memory Traffic for SPICE with the Best Static Global Policy

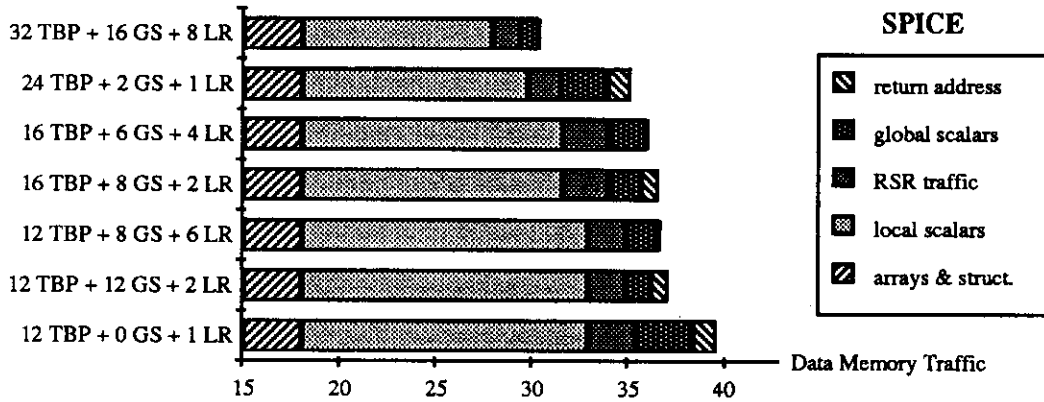


Figure 5.43: Data Memory Traffic for SPICE with Policy G^s-lf

for global scalars, and 1 link register.¹⁶ The data memory traffic generated by this configuration with the best static policy is 82% of the one generated by an intra-procedural optimizer with Policy B-lf and 12 TBP registers. When both architectural and compiler support are available, the inter-procedural optimizer generates 82% of the data traffic produced by the intra-procedural optimizer with Policy B-lf, and 89% by the optimizer with Policy G-lf. However, the data memory traffic generated is the same as the one produced when only compiler support is provided.

When the number of TBP registers is increased from 24 to 32 and, consequently, the general-purpose register file is doubled from 32 to 64 registers, the data memory traffic is smaller with architectural support than without, but only by 3%.

Although the data memory traffic generated with and without architectural support is the same for SPICE, we still believe that Policy G^s-lf is the best approach to reduce the overall data memory traffic. Although the inter-procedural optimizer with Policy

¹⁶We assume that 24 registers is the maximum number of TBP registers available in the processor, because the instructions to save and restore the TBP registers need to specify an opcode in addition to the 24-bit mask.

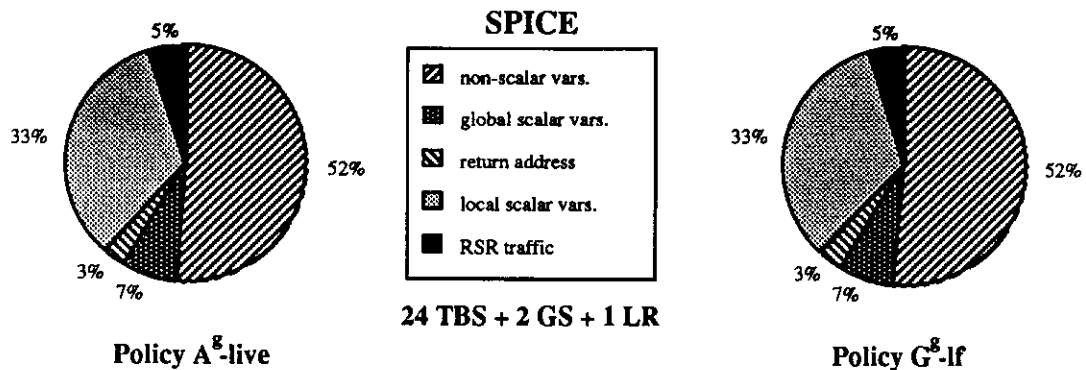


Figure 5.44: Remaining Data Memory Traffic for SPICE

A^s-live can eliminate most of the RSR traffic for a program like SPICE (because most of it is concentrated at the bottom of the call graph as we mentioned in Subsection 5.4.1; see Figure 5.14), this might not be the case for most numeric programs. If the program has its traffic more evenly distributed throughout the call graph or concentrated in functions which are part of a cycle, Policy G^s-lf will again generate less RSR traffic than any static policy and, possibly, less overall data memory traffic. However, we do not have any data to support this statement because SPICE is the only numeric application that we have measured.

Figure 5.44 shows the remaining data memory traffic after performing inter-procedural optimizations (compare this figure with Figure 5.1). As expected, the data memory traffic distribution is identical for both approaches, with and without architectural support.

Finally, let us mention how the compilation process is organized. By default, the intra-procedural register allocator only assigns variables to 16 TBP registers since this is a convenient number for non-numeric applications. If the inter-procedural optimizer concludes that more TBP registers should be used (like for SPICE), the optimizer asks the programmer to recompile the programs with an option to get more local scalar variables assigned to registers. This option can be given to the compiler by the programmer or can be available directly to the compiler from the data base (see Figure 5.2). Thus, if we assume that non-numeric applications account for most of workload of the system,¹⁷ most of the programs will not need to be recompiled because the assembler can modify the register assignment and the register saving/restoring instructions.

5.5.3 Speed-Up

In this section we evaluate the speed-up factor that we might obtain for a RISC-like machine. The model used to compute this speed-up factor is the one described in Sec-

¹⁷If this is not the case, the above assumption can be changed so that, by default, the register allocator always assigns variables to 24 TBP registers.

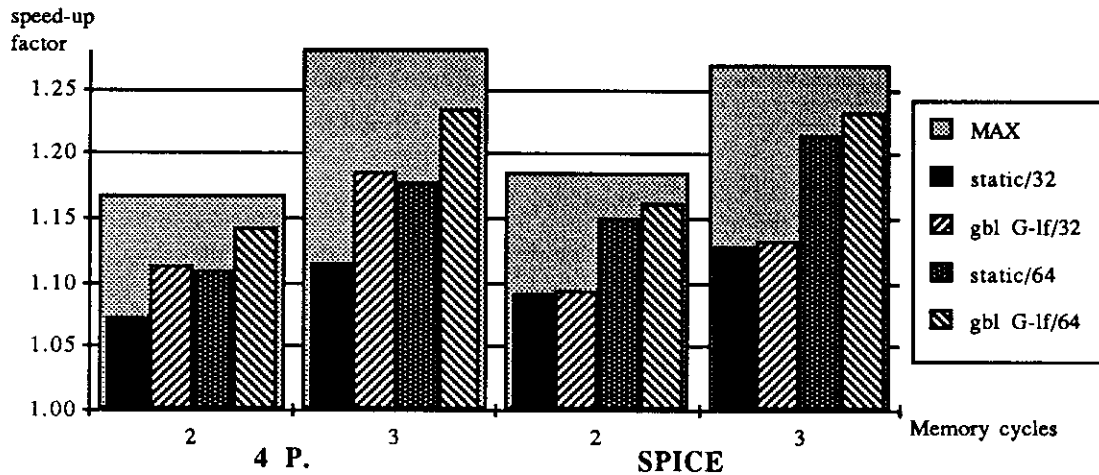


Figure 5.45: Speed-Up for Inter-Procedural Optimizations w.r.t. Policy B-lf

tion 4.6. The execution speed has been computed for the average of the four programs and for SPICE for the following four schemes:

static/32: The inter-procedural optimizer uses the best static policy for each program as described at the beginning of Section 5.5 with the $16\ TBP + 8\ GS + 2\ LR$ register-set partition for the four programs and with the $24\ TBP + 2\ GS + 1\ LR$, for SPICE. These are the partitions which generate the least traffic for a 32-register file (see Subsections 5.5.1 and 5.5.2).

gbl G-lf/32: The inter-procedural optimizer uses the dynamic Policy G^s-lf with the same register-set partitions as *static/32*.

static/64: The inter-procedural optimizer uses the best static policy for each program, as described at the beginning of Section 5.5, with the $32\ TBP + 16\ GS + 8\ LR$ register-set partition.

gbl G-lf/64: The inter-procedural optimizer uses the dynamic Policy G^s-lf with the $32\ TBP + 16\ GS + 8\ LR$ register-set partition.

The execution speed of these configurations is compared with a maximum speed-up. This has been computed for the case when there are 32 TBP registers for locals and 16 registers for globals, and when the RSR traffic is eliminated as well as the traffic caused by the return address.

The speed-up factor obtained by these schemes with respect to an intra-procedural optimizer with Policy B-lf and one with Policy G-lf is shown in Figures 5.45 and 5.46, respectively. For a 32-register file and the average of the four programs:

1. When only compiler support is provided, the inter-procedural optimizer with the best static policy has a speed-up factor of 1.07, when 2 processor cycles are required

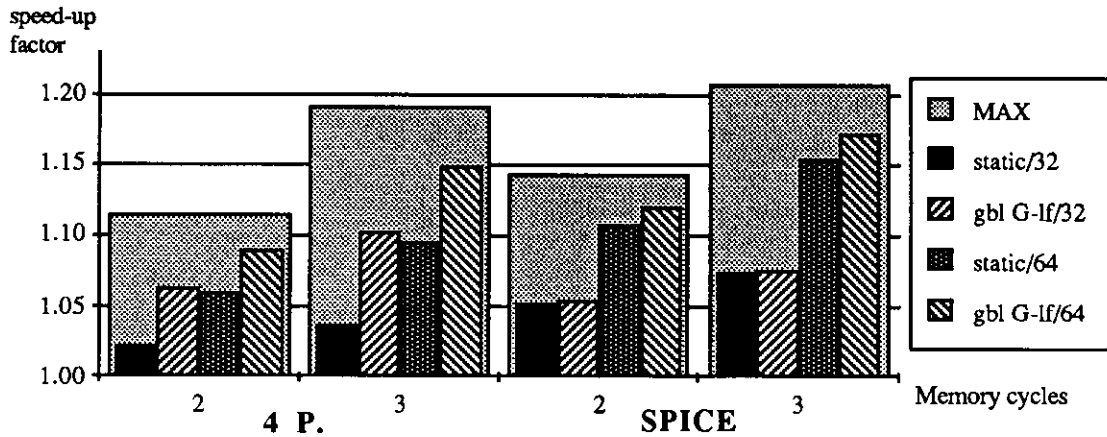


Figure 5.46: Speed-Up for Inter-Procedural Optimizations w.r.t. Policy G-lf

to access memory, and 1.11 when 3 are required with respect to the intra-procedural optimizer with Policy B-lf.

- When both architectural and compiler support are provided, the speed-up factor is similar (1.06 and 1.10) for the inter-procedural optimizer with Policy G⁵-lf with respect to the intra-procedural optimizer with Policy G-lf.

For SPICE, the speed-up factors are of 1.09, 1.13, 1.05, and 1.08, respectively.

If we compare the inter-procedural optimizer with the dynamic Policy G⁵-lf with respect to the inter-procedural optimizer with the best static policy, for the four programs the speed-up factor is 1.04, when 2 processor cycles are required to access memory, and 1.06 when 3 are required. However, for SPICE the program execution speed is the same with and without architectural support because the data memory traffic generated by both approaches is also the same.

Finally, notice that for the four programs, the execution speed of a program with the *static/64* configuration is almost equivalent to the execution speed with the *gbl G-lf/32* configuration.

5.6 Summary

In this chapter we have presented how the data memory traffic can be further reduced with *inter-procedural optimizations*. Our optimizer concentrates its efforts in three areas: global scalars, return addresses, and register savings and restorings. While static information is sufficient for the intra-procedural optimizer to perform an efficient register allocation (see Table A.1), we have shown that this is not the case for the inter-procedural optimizer (see Table 5.2). Thus, the *average of three execution profiles* for each program

is used by our optimizer (see Section 5.2).

To reduce the *global scalar traffic*, the optimizer selects the most frequently used variables indicated by the average execution profile and assigns them to registers (if they do not have any alias). We have shown that most of the traffic is produced by a few variables (see Table 5.2 and Figures 5.3 and 5.4). For instance, 8 registers cut the global scalar traffic by 49% for the average of the four programs and by 44% for SPICE. On the other hand, if only static information is used to select the variables to allocate, the traffic reduction is only 21% and 6%, respectively. For this reason, only dynamic information is used by the inter-procedural optimizer.

To reduce the *return-address traffic*, two different approaches have been evaluated. On one hand, architectural support in the form of multiple PCs, which are handled as a multiple-window register file with a window size of one. In this case, no memory traffic is generated while there are no overflows or underflows. Our measurements have shown that the RA traffic is almost eliminated if at least an 8-register file is provided (see Table 5.5).

On the other hand, with compiler support, the optimizer selects a few general-purpose registers to be used as link registers. All the functions at a given height, except recursive functions and functions called through a pointer, share the same link register (see Section 5.3). The average execution profile is used by the optimizer to select the functions at a given height which generate the most of the RA traffic. When the program does not have a large percentage of recursive functions, the RA traffic can be significantly eliminated with a few registers (see Table 5.4 and Figures 5.9 and 5.10). For instance, with 4 registers, 90% of the RA traffic is eliminated for SPICE, 99% for SORT, and about 60% for the other programs.

The tradeoff between selecting architectural support and compiler support is that for the former, the PC traffic is almost eliminated, but a special programming environment has to be defined to handle overflows and underflows and the processor flow has to be interrupted when these conditions arise. For the latter, the optimizer makes a better use of the general-purpose registers available, but the amount of traffic eliminated is limited by the program's characteristics (e.g., recursive functions).

To reduce the *register saving/restoring traffic*, four inter-procedural policies have been introduced based on the intra-procedural Policies A-live, A-lvOpt, B-lf, and G-lf. The inter-procedural policies perform register assignment in such a way that the RSR traffic is reduced. Policies A^s-live and A^s-lvOpt eliminate the register saving/restoring instructions performed at a specific call when the registers being saved/restored are not used by any of the functions that might be called from this point. Policy B^s-lf eliminates the register saving/restoring instructions performed at function entry and return when the registers being saved/restored are not used by any of the functions that might call this function. Thus, these static policies find a disjoint register assignment such that the maximum number of register saving/restoring instructions can be eliminated.

The static inter-procedural optimizations not only generate less RSR traffic than

their respective intra-procedural optimizations (see Tables 5.6, 5.7 and 5.8), but also the RSR traffic produced decreases for larger register sets (see Figures 5.23 and 5.31). In Chapter 4 we mentioned that when intra-procedural register allocation and assignment are performed, there is no need for having more than 12 TBP registers for non-numeric applications because the decrease in the local scalar traffic is balanced with the increase in the RSR traffic. This is not case when inter-procedural register assignment is performed. Therefore, a larger register set can be efficiently utilized when the compiler uses Policies A^s-live, A^s-lvOpt, or B^s-lf.

However, our measurements have not shown which is the best static policy to use for any program and for a given register-set configuration, because the RSR traffic reduction provided by these policies closely depends on the characteristics of the program (its maximum depth, where most of its RSR traffic is generated, the number of registers allocated by the intra-procedural optimizer, etc.). Thus, the optimizer should have all three of them available so that it can select the most appropriate one for a program based on the profile information (as discussed in Subsection 5.4.3).

The fourth inter-procedural policy to reduce the RSR traffic is for the dynamic Policy G. Policy G^s-lf assigns registers in a round-robin fashion for the functions in the same path, whenever this is possible. The optimizer, in this case, does not have to eliminate any register saving/restoring instruction because this is already being done by the dynamic behavior of the policy itself.

Policy G^s-lf generates less RSR traffic than any of the static inter-procedural optimizations (see Table 5.10 and Figures 5.37 and 5.38). For the average of the four programs, the best static policy has between 445% (for 12 TBP registers) and 816% (for 32) of the RSR traffic generated by Policy G^s-lf. For SPICE, Policy A^s-live, which is the best static policy for any register-set configuration, has between 103% and 155%.

Policy G^s-lf not only generates the least RSR traffic, but also it simplifies the compiler implementation and reduces the RSR traffic independently of where the functions that generate most of the RSR traffic are located in the call graph and of the number of recursive functions in the program (see Subsection 5.4.4).

The traffic generated by each optimization and the number of registers required for each one have been evaluated to find the best partition of the register set to obtain the maximum overall traffic reduction (see Figures 5.39 and 5.40). For the four programs and a 32-register file (with 16 TBP registers, 8 registers for globals, and 2 link registers), the inter-procedural optimizer with Policy G^s-lf has 86% of the traffic produced by the inter-procedural optimizer with the best static policy (selected by the optimizer itself). In this case, the speed-up factor that we might obtain for a RISC-like processor is of 1.04, when 2 cycles are required to access memory, and 1.06 when 3 are required. However, for SPICE, the data memory traffic and the execution speed are the same with and without architectural support (see Figures 5.42, 5.43, 5.45, and 5.46).

Since Policy G^s-lf usually generates less data memory traffic than any static policy (with the exception of SPICE), we still believe that both architectural and compiler

support (i.e., Policy G^s-lf) is the best approach to simplify the compilation process and to reduce the overall data memory traffic, independently of the characteristics of the programs.

Chapter 6

Conclusions and Future Work

In this dissertation, architectural and compiler support that may be provided to make function calls more efficient has been discussed and evaluated. In Section 6.1 we comment on the most important results obtained and in Section 6.2 we outline future work.

6.1 Conclusions

Chapter 3 presented six new architectural policies to reduce the register saving and restoring overhead during function calls for single-window architectures. These policies make use of *dynamic* information to know which registers have been used during program execution. The six dynamic policies have been evaluated and compared with the two conventional static policies: to save/restore registers at the caller (**Policy A**) and to save/restore at the callee (**Policy B**). We concluded that one of the dynamic policies, **Policy G**, is our best candidate for implementation since it is the one that generates the least RSR traffic. Policy G has between 12% and 31% of the traffic generated by Policy B and between 5% and 20% of Policy A when there are between 6 and 32 TBP registers available to the optimizer. These measurements have been performed using the code generated by the Portable C Compiler (PCC) for four programs: the NROFF word processor, the SORT program, the VAX-11 assembler (ASM), and the Portable C Compiler for VAX-11 (VPCC).

To perform the above-mentioned evaluations a new tool has been designed: a *Block-and-Actions Generator* (**BKGEN**). BKGEN reduces drastically the overhead introduced by a conventional simulator—the tool traditionally used to obtain these types of measurements. BKGEN obtains a version of the program being measured which is directly executable on the existing machine (EM) and associates with each block of proposed machine (PM) instructions a set of actions that reflect the measurements to be taken for the PM. When the program is executed on the EM, the measurements associated with the PM are collected. To be able to do so, a mapping is defined to associate each EM block with a PM block of actions. The main advantage of BKGEN is that since

the execution time is substantially reduced, *large typical* programs (like compilers, word processors, etc.) can be measured.

The implementation of Policy G has also been sketched. We have shown that most of the activities required by Policy G are performed in parallel with the main CPU activities and that the number of cycles required to perform the register saving/restoring operations is the same as the number required by the conventional static policies. Although the feasibility of the implementation of Policy G depends on many design requirements and constraints, we have not found any major drawbacks for which its implementation should be discarded. Therefore, we encourage the designer to consider its inclusion in his/her design.

We have also introduced six new compiler optimizations to reduce the RSR overhead. Two of these optimizations are *intra-procedural* and based on live-variable analysis, Policies **A-lvOpt** and **G-live**, and the other four are *inter-procedural*, Policies **A^g-live**, **A^g-lvOpt**, **B^g-lf**, and **G^g-lf**.

Chapter 4 presented the two new intra-procedural policies to reduce the RSR traffic and compared them with the already existing policies, Policies **A-live**, **B-lf**, and **G-lf**. Policy **A-lvOpt** always generates less RSR traffic than the standard Policy **A-live** for any program and any number of TBP registers (6, 8, 12, 16, 24, and 32). Policy **G-live** also generates less RSR traffic than Policy **G-lf**; however, since the implementation of this optimization requires the execution of additional instructions, it has been discarded because the increase in instruction memory traffic is larger than the overall decrease in data memory traffic.

When no architectural support is provided, our measurements have not shown which static policy is best to use, Policy **A-lvOpt** or Policy **B-lf**. Depending on the program's characteristics, one policy performs better than the other. Thus, we propose that the compiler should let the programmer decide which of the two policies he/she wants to use. When architectural support is provided, Policy **G-lf** generates the least RSR traffic.

Our measurements do not justify the use of more than 12 TBP registers for non-numeric applications, because for larger register sets the reduction in local scalar traffic is compensated for by the increase in RSR traffic. This is not the case for SPICE. For SPICE, the optimizer can efficiently use a larger register set because of its heavy use of registers. When 12 registers are available to the allocator of the GNU C Compiler, the overall data memory traffic reduction when both architectural and compiler support are provided is 15% for the average of the four programs and 8% for SPICE with respect to when only compiler support is provided.

Chapter 5 presented our inter-procedural optimizer. The optimizer not only reduces the RSR traffic, but also the global scalar traffic and the traffic caused by the return address. Our measurements have shown that each inter-procedural RSR policy reduces significantly the RSR traffic generated by its equivalent intra-procedural policy. When architectural support is provided, Policy **G^g-lf** generates the least RSR traffic. However, when only compiler support is provided, our measurements have not shown which of the

three static policies is the best to use because their performance is closely related to the program's characteristics. Thus, all three policies should be available to the inter-procedural optimizer so that it can select the most appropriate one for each program based on a *dynamic execution profile*.

The inter-procedural optimizer can efficiently use a larger register set not only to reduce the RSR traffic, but also the global scalar traffic and the return-address traffic. For the average of the four programs and a 32-general purpose file, with 16 TBP registers, 8 registers for globals, and 2 link registers, the inter-procedural optimizer with Policy G^s-lf has 67% of the data memory traffic generated by an intra-procedural optimizer with Policy B-lf, 79% of the traffic generated by the intra-procedural optimizer with Policy G-lf, and 86% of the traffic produced by the inter-procedural optimizer with the best static policy (selected by the optimizer itself).

For SPICE, the overall data memory traffic generated when both architectural and compiler support are provided is the same as the one generated when only compiler support is provided.

Since Policy G^s-lf usually generates less data memory traffic than any static policy (with the exception of SPICE), the best approach is to provide both architectural and compiler support (i.e., Policy G^s-lf). Policy G^s-lf simplifies the compilation process and reduces the overall data memory traffic, independently of where the functions, which generate most of the RSR traffic, are located in the call graph and of the percentage of recursive functions in the program.

6.2 Suggestions for Future Work

Although throughout the dissertation several questions have been answered with quantitative measurements, there have been others which have only been broached. In this section, we outline future research in the following three areas:

1. VLSI implementation of Policy G.
2. Cache memory and registers.
3. Register allocation.

VLSI Implementation of Policy G

In Section 3.5 we discussed several factors (communication delays due to longer data buses, fan-out increase, etc.) which might affect the processor cycle time. We also mentioned that the effect of these factors on the processor cycle time depends on many design requirements and constraints for a given implementation and that although an implementation was performed, the effect of Policy G in this implementation could not be generalized to other alternative implementations—even for the the same architecture.

However, it is important to have an idea of at least one implementation to obtain a better understanding of the factors which might affect the processor cycle time.

Cache Memory and Registers

In this dissertation we have considered the data memory traffic as the evaluation parameter to compare the performance of the different optimizations proposed. However, this parameter does not reflect the overall performance of the system when a cache memory is present. Since a cache memory is required to reduce the data memory traffic produced by the local and global arrays and structures (the most significant part of the data memory traffic once intra-procedural optimizations have been performed; see Figure 4.28), we would like to perform measurements to obtain an answer for the following questions:

1. Is the “real” (i.e., non-cached) data memory traffic reduced when global scalar variables are assigned to registers? Since the few global scalars which are assigned to registers are frequently referred, it might be that they usually are available in the cache.
2. When inter-procedural optimizations are performed less data memory traffic is generated so that less conflicts are produced in the cache memory. Thus, can the cache memory size be reduced for an equivalent performance when inter-procedural optimizations are performed?

No measurements are proposed for local scalar variables since we prefer to assign local scalar variables to registers rather than to the cache, as we mentioned in Subsection 2.2.1.

Register Allocation

In Subsection 2.1.3.2 we mentioned that the register allocator can assign variables to registers in three different *scopes*: per basic block, per procedure or function, and for the whole program.

Intra-procedural register allocators can assign local scalar variables to registers with any of the first two alternatives, per basic block or per function. When all local scalar variables can be assigned to TBP registers, the local scalar traffic generated by both approaches should be the same (zero). Otherwise, the first alternative might generate less local scalar traffic than the second one if the spill traffic for the first alternative is smaller than the non-assigned local scalar traffic plus the RSR traffic for the second one. Since our Policy G cannot be used when variables are assigned per basic block (because the saving traffic is not performed at function entry), measurements should be taken to determine which alternative generates the least traffic.

Inter-procedural allocators can assign local scalar variables to registers with any of the three alternatives. In fact, Chow’s allocator [Chow84] does it for the first (per basic

block); our inter-procedural optimizer, for the second (per function); and Wall's allocator [Wall86], for the third (for the whole scope of the program). When the whole call graph can be mapped to the TBP registers available, the local scalar traffic should be the same for any alternative since all RSR instructions and spill instructions should be eliminated. If only compiler support is provided, the remaining local scalar traffic corresponds to the variables with aliases and the variables defined in recursive functions. If architectural support is provided, part of the local scalar traffic generated by the variables defined in recursive functions can also be eliminated.

When the whole call graph cannot be mapped to the TBP registers available, then we should compare the traffic generated by the non-assigned variables with the RSR traffic produced by our inter-procedural optimizer.¹ We expect that, for large programs, the RSR traffic is smaller than the traffic produced by the variables which cannot be assigned for the whole scope of the function. However, measurements are needed to confirm this conjecture.

¹We only mention allocation per function and not per basic block because we are interested in comparing Wall's allocator with our inter-procedural optimizer.

Appendix A

On Intra-Procedural Register Allocation

In this appendix we evaluate the intra-procedural register allocation policies performed by the compilers used in this dissertation (the GNU C Compiler [Stal88], the Portable C Compiler [John79], and the Amsterdam Compiler Kit [Tane83]). These register allocation policies have already been described.¹ In Sections 4.2 and 4.5 the performance of these compilers has already been evaluated with respect to the data memory traffic generated by them.

In Section 4.2 the RSR traffic generated by the three compilers has been compared and it has been concluded that when local scalar variables with disjoint lifetimes *share* the same register (as it is done by the GNU C Compiler), the RSR traffic generated by Policy B-lf (used by the three compilers) is reduced with respect to when each register is *dedicated* to a single local variable during the whole execution of the function (as it is done by ACK and PCC). For instance, GNU generates between 54% (for 32 TBP registers) and 61% (for 6) of the RSR traffic produced by PCC (see Table 4.7 and Figure 4.14).

In Section 4.5 the overall traffic generated by PCC and by GNU has also been compared for the average of the four programs (ASM, NROFF, SORT, and VPCC) and for SPICE.² It has been shown that GNU generates less local scalar traffic (see Figure 4.24) and that when the number of TBP registers is small (less than 16), GNU performs a better selection of the local scalar variables which are to be allocated to registers.

Although it is not the goal of this dissertation to develop any new register allocation scheme, it was considered necessary to evaluate the register allocators provided by these compilers to determine how close they are from an *optimal* allocation (as defined below),

¹See the introduction of Chapter 3 for a description of the PCC register allocation policy and Section 4.2 for a description of the other two.

²See Section 4.5 for a discussion of why the ACK overall traffic was not compared with the PCC and the GNU overall traffic and Section 4.3 for a discussion of why this program (SPICE) was separated from the other four.

because the RSR traffic generated by the different policies depends on the number of registers allocated. Thus, our goals in this appendix are the following:

- To compare two selection mechanisms for local scalar variables: by *sequential order of definition* as used by PCC and by *static linear priority* as used by ACK and GNU. Both mechanisms are compared with an *optimal* register allocation (Section A.1). It is concluded that the selection by static linear priority performs quite close to the optimum (93%–95%) when the local scalar traffic is significant (i.e., for SPICE) and, therefore, no new selection mechanisms need to be developed like, for example, dynamic profiling for inter-procedural optimizations (see Chapter 5).
- To evaluate the local scalar traffic reduction obtained when a register is *shared* among different local scalars with disjoint lifetimes, as it is done by GNU, with respect to when a register is *dedicated* to each local scalar during the whole execution of the function, as it is done by PCC and ACK (Section A.2). As mentioned above, GNU generates less RSR traffic and less overall traffic than PCC. However, the previous comparisons are performed with two different selection mechanisms, by sequential order of definition and by static linear priority, and include the overall traffic. In Section A.2 it is shown that when variables are selected by static linear priority and registers are shared, the local scalar traffic generated is between 72% (for 32 TBP registers) and 92% (for 8) of the traffic generated when registers are dedicated, for SPICE, and between 7% (for 12) and 72% (for 6) for the average of the four programs.

A.1 Selection Mechanism for Local Scalars

In this section two selection mechanisms for local scalar variables, *sequential order of definition* and *static linear priority*, are compared with an *optimal* register allocation. The comparison is based on the data memory traffic generated by the non-assigned local scalar variables. The traffic generated by the optimal register allocation corresponds to the minimum traffic generated when the most frequently used local scalars (for this specific run) are assigned to registers. This criterion is similar to Belady's optimal page replacement algorithm [Bela66] since the "future" program behavior needs to be known in advance. Notice that under different input data another set of local scalar variables might generate less data memory traffic.

Table A.1 shows the traffic generated by local scalar variables (*lsv*) normalized with respect to the traffic generated per function when no TBP registers are available. This traffic does not include the RSR traffic. From the table the following can be concluded:

1. The number of registers required per function is usually small. For instance, if the most frequently used local scalar of every function was assigned to a register (i.e., *optimal* allocation), SPICE would cut its traffic by one fourth, ASM by one third, SORT and VPCC in half, and NROFF by two thirds.

no. regs.	register allocation	ASM	NROFF	SORT	VPCC	4 P.	SPICE
0	(lsv traffic)	1 (43.85)	1 (13.77)	1 (43.18)	1 (22.01)	1 (22.77)	1 (47.6)
1	seq. by def.	0.89	0.61	0.81	0.71	0.73	0.89
	priority	0.68	0.35	0.74	0.59	0.58	0.82
	optimal	0.65	0.32	0.58	0.48	0.48	0.78
2	seq. by def.	0.76	0.22	0.79	0.57	0.56	0.75
	priority	0.46	0.09	0.67	0.37	0.40	0.67
	optimal	0.42	0.07	0.36	0.26	0.25	0.63
3	seq. by def.	0.57	0.08	0.71	0.31	0.41	0.67
	priority	0.36	0.05	0.44	0.23	0.26	0.57
	optimal	0.27	0.03	0.28	0.13	0.17	0.56
4	seq. by def.	0.44	0.04	0.64	0.19	0.33	0.63
	priority	0.20	0.02	0.23	0.14	0.14	0.52
	optimal	0.17	0.01	0.22	0.07	0.12	0.50
6	seq. by def.	0.12	0.01	0.55	0.06	0.24	0.55
	priority	0.27	0.00	0.19	0.05	0.10	0.46
	optimal	0.07	0.00	0.14	0.02	0.06	0.44
8	seq. by def.	0.10	0.00	0.14	0.01	0.06	0.50
	priority	0.10	0.00	0.14	0.01	0.07	0.42
	optimal	0.02	0.00	0.08	0.00	0.03	0.40
10	seq. by def.	0.07	0.00	0.12	0.01	0.05	0.46
	priority	0.06	0.00	0.07	0.00	0.03	0.39
	optimal	0.01	0.00	0.04	0.00	0.02	0.37
12	seq. by def.	0.04	0.00	0.10	0.01	0.04	0.44
	priority	0.03	0.00	0.06	0.00	0.02	0.37
	optimal	0.00	0.00	0.02	0.00	0.01	0.35
16	seq. by def.	0.03	0.00	0.03	0.00	0.01	0.43
	priority	0.01	0.00	0.01	0.00	0.01	0.34
	optimal	0.00	0.00	0.00	0.00	0.00	0.32
20	seq. by def.	0.02	0.00	0.00	0.00	0.00	0.42
	priority	0.00	0.00	0.00	0.00	0.00	0.33
	optimal	0.00	0.00	0.00	0.00	0.00	0.30
24	seq. by def.	0.00	0.00	0.00	0.00	0.00	0.41
	priority	0.00	0.00	0.00	0.00	0.00	0.30
	optimal	0.00	0.00	0.00	0.00	0.00	0.28
32	seq. by def.	0.00	0.00	0.00	0.00	0.00	0.37
	priority	0.00	0.00	0.00	0.00	0.00	0.27
	optimal	0.00	0.00	0.00	0.00	0.00	0.26

Table A.1: Local Scalar Traffic Reduction



Figure A.1: Local Scalar Traffic for the Four Programs

- When the variables are selected either by sequential order of definition (labeled “seq. by def.”) or by static linear priority (labeled “static”), 6–8 TBP registers are enough for the four programs (see also Figure A.1), but not for SPICE (see also Figure A.2). Since SPICE uses more local scalars per function than the four programs and most of these are double floating-point variables, SPICE requires more registers. However, the small traffic reduction obtained for more than 12 TBP registers is mitigated by the increase in RSR traffic (when no architectural support is available) as it has been shown in Section 4.5 (see Figure 4.27). Thus, as we concluded in Chapter 4, 12 TBP registers are enough for intra-procedural allocators when no architectural support is available.
- The selection by static linear priority performs quite close to the optimum when the local scalar traffic is significant (i.e., mainly for SPICE; see Figures A.1 and A.2). This is true for every program except when there are fewer than 4 TBP registers. For this reason, no other selection mechanism has been evaluated like, for example, allocation by static weighted priority or by dynamic profiling (as it has been done for the inter-procedural optimizer; see Chapter 5).
- As expected, the allocation by sequential order of definition performs a worse selection of local scalars than the allocation by static linear priority. However, since the number of TBP registers required per function for the four programs is small, the allocation performed by any of the two mechanisms is equivalent. For this reason, as it has been shown in Section 4.2, the conclusions obtained in Chapter 3 and in Section 4.1 with PCC are similar to the ones obtained with ACK and GNU.

On the other hand, for SPICE the allocation by sequential order of definition performs a worse selection than allocation by static linear priority because of the (already mentioned) larger number of registers required for this program. For this reason, only GNU has been used to evaluate the data memory traffic generated by

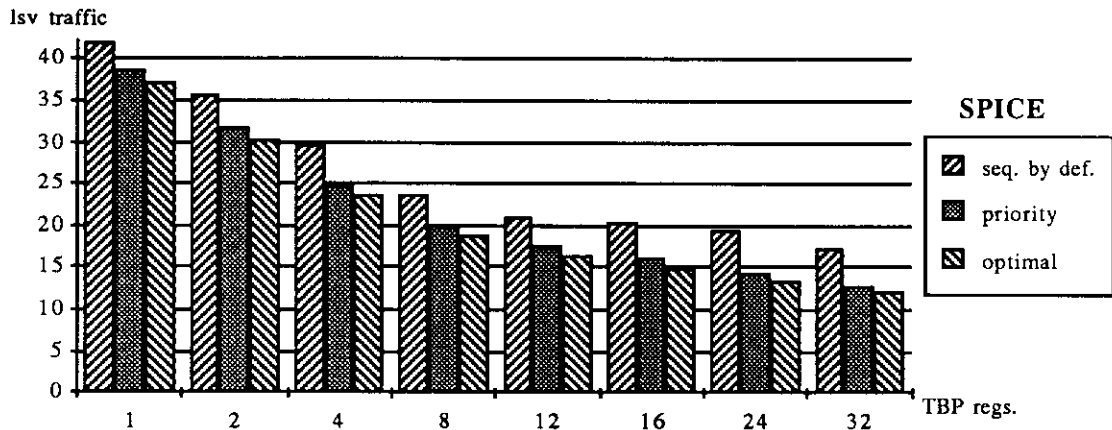


Figure A.2: Local Scalar Traffic for SPICE

SPICE.

Therefore, the selection mechanisms used by PCC, ACK, and GNU for the four programs and by GNU for SPICE perform quite close to an optimal allocation and, for this reason, our conclusions should be valid for any intra-procedural register allocator which assigns local scalar variables for the whole scope of the function (see Section 6.2 for a discussion of intra-procedural allocators which assign local scalar variables for the basic block scope).

A.2 Dedicated versus Shared Registers for Local Scalars

In Section 4.2 an advantage of sharing registers among local scalar variables with disjoint lifetimes is shown for Policy B-lf: the RSR traffic is cut between one third and one half (depending on the number of TBP registers available; see Figure 4.14). Although this comparison is performed between the RSR traffic generated by GNU and by PCC (both with Policy B-lf), the comparison is valid, because (as it was shown in the previous section) the RSR traffic generated should be similar with any register allocation scheme. Moreover, in Section 4.5 it has been shown that GNU generates less overall data memory traffic than PCC. For instance, when 12 TBP registers are available, the overall traffic generated by GNU is 85% of the one produced by PCC for the four programs (see Figures 4.25 and 4.26). In this section we evaluate exclusively the reduction of local scalar traffic caused by sharing the registers among locals versus having a dedicated register to each local.

Table A.2 shows the traffic generated by local scalar variables when they are assigned to only one register and when the ones with disjoint lifetimes share the same register. The traffic is normalized with respect to the local scalar traffic generated per function

no. regs.	register allocation	ASM	NROFF	SORT	VPCC	4 P.	SPICE
0	(lsv traffic)	1 (43.85)	1 (13.77)	1 (43.18)	1 (22.01)	1 (22.77)	1 (47.6)
1	dedicated	0.68	0.35	0.74	0.59	0.58	0.82
	shared	0.64	0.31	0.74	0.66	0.59	0.81
	optimal	0.64	0.26	0.70	0.55	0.53	0.80
2	dedicated	0.46	0.09	0.67	0.37	0.40	0.67
	shared	0.49	0.14	0.52	0.38	0.37	0.67
	optimal	0.41	0.08	0.50	0.32	0.32	0.66
3	dedicated	0.36	0.05	0.44	0.23	0.26	0.57
	shared	0.32	0.05	0.42	0.27	0.26	0.59
	optimal	0.28	0.04	0.38	0.19	0.22	0.58
4	dedicated	0.20	0.02	0.23	0.14	0.14	0.52
	shared	0.23	0.03	0.32	0.19	0.19	0.53
	optimal	0.17	0.02	0.26	0.11	0.14	0.52
6	dedicated	0.27	0.00	0.19	0.05	0.10	0.46
	shared	0.08	0.00	0.14	0.05	0.07	0.45
	optimal	0.06	0.00	0.13	0.03	0.06	0.44
8	dedicated	0.10	0.00	0.14	0.01	0.07	0.42
	shared	0.06	0.00	0.06	0.01	0.03	0.39
	optimal	0.02	0.00	0.06	0.01	0.02	0.38
10	dedicated	0.06	0.00	0.07	0.00	0.03	0.39
	shared	0.03	0.00	0.02	0.00	0.01	0.36
	optimal	0.01	0.00	0.02	0.00	0.01	0.35
12	dedicated	0.03	0.00	0.06	0.00	0.02	0.37
	shared	0.01	0.00	0.00	0.00	0.00	0.34
	optimal	0.00	0.00	0.00	0.00	0.00	0.33
16	dedicated	0.01	0.00	0.01	0.00	0.01	0.34
	shared	0.00	0.00	0.00	0.00	0.00	0.30
	optimal	0.00	0.00	0.00	0.00	0.00	0.29
20	dedicated	0.00	0.00	0.00	0.00	0.00	0.33
	shared	0.00	0.00	0.00	0.00	0.00	0.28
	optimal	0.00	0.00	0.00	0.00	0.00	0.27
24	dedicated	0.00	0.00	0.00	0.00	0.00	0.30
	shared	0.00	0.00	0.00	0.00	0.00	0.25
	optimal	0.00	0.00	0.00	0.00	0.00	0.24
32	dedicated	0.00	0.00	0.00	0.00	0.00	0.27
	shared	0.00	0.00	0.00	0.00	0.00	0.21
	optimal	0.00	0.00	0.00	0.00	0.00	0.21

Table A.2: Local Scalar Traffic with Dedicated and Shared Registers

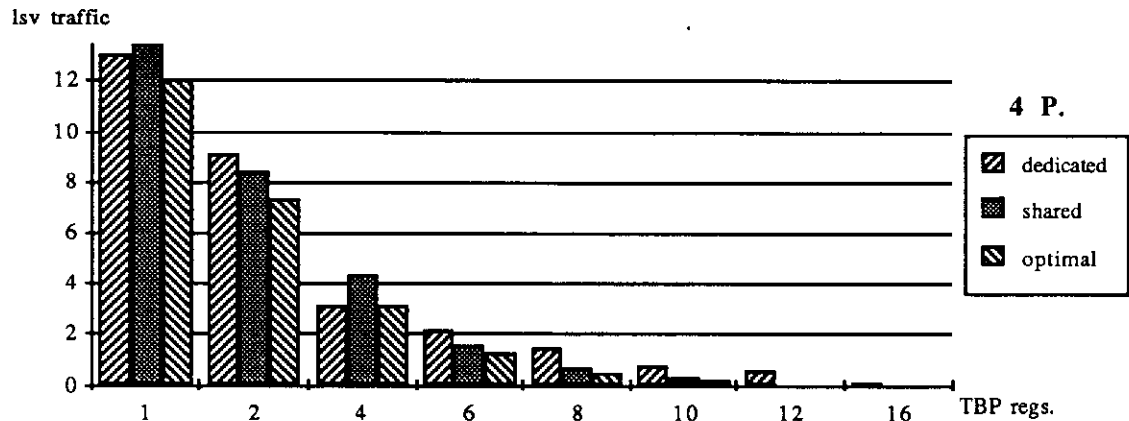


Figure A.3: Dedicated versus Shared Registers for the Four Programs

when there are no TBP registers. The variables to be allocated have been selected by static linear priority. The table also shows the traffic generated by an optimal register allocation which is similar to the optimal allocation described in the previous section, but with register sharing in this case. From the table we can conclude the following:

1. Static linear priority still offers an efficient selection mechanism, because the traffic generated when registers are shared is quite close to the optimum for all programs when there are at least 6 TBP registers available (see also Figures A.3 and A.4).
2. For some programs, when a small number of TBP registers (up to 4) are available, dedicated registers generate less traffic than shared registers. This implies that the addition of the static linear priorities associated with two (or more) local scalars is greater than the static linear priority associated with a single variable. For this reason, the combination of locals is assigned to the register rather than the single local scalar, but this assignment turns out to be a bad choice. However, this anomaly is ignored because only occurs for small register sets.
3. In general, shared registers reduce the local scalar traffic generated with respect to dedicated registers.
 - For SPICE, the local scalar traffic generated when 8–16 TBP registers are shared among locals with disjoint lifetimes is about 90% of the traffic produced when the same number of registers is dedicated to a single local scalar variable and about 80% when 24–32 TBP registers are available.
 - For the four programs, the traffic reduction becomes even more significant: 72% for 6 TBP registers, 43% for 8, and 7% for 12. For larger register sets, the traffic becomes zero first for shared registers and, afterwards, for both dedicated and shared.

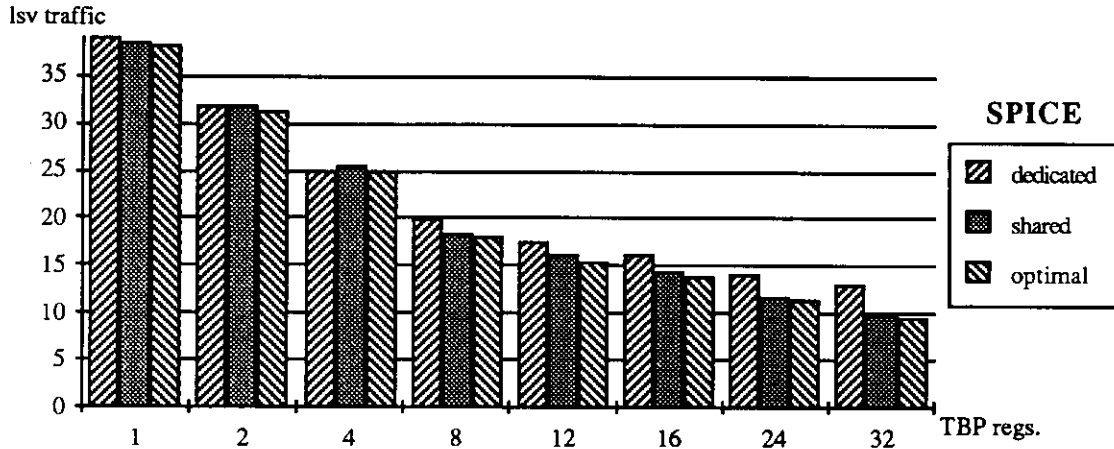


Figure A.4: Dedicated versus Shared Registers for SPICE

In conclusion, live-variable analysis should be performed not only when the intra-procedural register allocator uses either Policy A-live or Policy A-lvOpt as the RSR policies, but also when Policy B-lf is used. In the latter case, live-variable analysis reduces the RSR traffic (see Figure 4.14) and the local scalar traffic by sharing registers among local variables with disjoint lifetimes (see Figures A.3 and A.4) and by assigning dead local variables to TBD registers rather than to TBP registers (see Figure 4.24).

Bibliography

- [Aboa87] H.A. Aboalsamh and B. Furht, "Multiple Register Window File for LISP-Oriented Architectures," in *IEEE Miami Technicon '87*, 1987.
- [ACK85] *Amsterdam Compiler Kit: Reference Manual*, UniPress Software, Edison, NJ 08817, 1985.
- [Adam85] G. Adams, B.K. Bose, L. Pei, and A. Wang, "The Design of a Floating-Point Processor Unit," in R.H. Katz, editor, *Proceedings of CS292i: Implementation of VLSI Systems (Spring 1985)*, Computer Science Division (EECS), University of California, Berkeley, CA 94720, September 1985. Published as Technical Report UCB/CSD 86/259.
- [Aho86] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.
- [Alex75] W.G. Alexander and D.B. Wortman, "Static and Dynamic Characteristics of XPL Programs," *IEEE Computer*, vol. 8, no. 11, November 1975, pp. 41-46.
- [Alle80] F.E. Allen, J.L. Carter, J. Fabri, J. Ferrante, W.H. Harrison, P.G. Loewner, and L.H. Trevillyan, "The Experimental Compiling System," *Journal of Research and Development*, vol. 24, no. 6, November 1980, pp. 695-715.
- [Ankl82] P. Anklam, D. Cutler, R. Heinen, Jr., and M.D. MacLaren, *Engineering a Compiler: VAX-11 Code Generation and Optimization*, Digital Press, 1982.
- [Atki87] R.R. Atkinson and E.M. McCreight, "The Dragon Processor," in *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, October 1987, pp. 65-69.
- [Ausl82] M. Auslander and M. Hopkins, "An Overview of the PL.8 Compiler," in *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, June 1982, pp. 22-31.
- [Back67] J.W. Backus *et al.*, "The FORTRAN Automatic Coding System," in S. Rosen, editor, *Programming Systems and Languages*, McGraw-Hill, 1967, pp. 29-47.

- [Band87] S. Bandyopadhyay, V.S. Begwani, and R.B. Murray, "Compiling for the CRISP Microprocessor," in *Spring COMPCON '87*, February 1987, pp. 96–100.
- [Barb81] M.R. Barbacci, "Instruction Set Processor Specifications (ISPS): The Notation and Its Applications," *IEEE Transactions on Computers*, vol. C-30, no. 1, January 1981, pp. 24–40.
- [Basa83] E. Basart and D. Folger, "RIDGE 32 Architecture—A RISC Variation," in *IEEE International Conference on Computer Design: VLSI in Computers*, November 1983, pp. 315–318.
- [BBN81] *C/70 Hardware Reference Manual*, BBN Computer Company, Cambridge, MA 02238, 1981.
- [Beat74] J.C. Beatty, "Register Assignment Algorithm for Generation of Highly Optimized Object Code," *IBM Journal of Research and Development*, vol. 18, no. 1, January 1974, pp. 20–39.
- [Beel84] M. Beeler, "Beyond the Baskett Benchmark," *Computer Architecture News*, vol. 12, no. 1, March 1984, pp. 20–31.
- [Bela66] L.A. Belady, "A Study of Replacement Algorithms for a Virtual-Storage Computer," *IBM Systems Journal*, vol. 5, no. 2, 1966, pp. 79–101.
- [Bere82] A. Berenbaum, M. Condry, and P. Lu, "The Operating System and Language Support Features of the BELLMAC-32 Microprocessor," in *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, March 1982, pp. 30–38.
- [Bere87a] A.D. Berenbaum, D.R. Ditzel, and H.R. McLellan, "Architectural Innovations in the CRISP Microprocessor," in *Spring COMPCON '87*, February 1987, pp. 91–95.
- [Bere87b] A.D. Berenbaum, D.R. Ditzel, and H.R. McLellan, "Introduction to the CRISP Instruction Set Architecture," in *Spring COMPCON '87*, February 1987, pp. 86–90.
- [Bern89] D. Bernstein *et al.*, "Spill Code Minimization Techniques for Optimizing Compilers," in *SIGPLAN'89 Conference on Programming Language Design and Implementation*, June 1989, pp. 258–263.
- [Birn85] J.S. Birnbaum and W.S. Worley, Jr., "Beyond RISC: High-Precision Architecture," *Hewlett-Packard Journal*, vol. 36, no. 8, August 1985, pp. 4–10.
- [Blom83] R.A. Blomseth, *A Big RISC*, Technical Report UCB/CSD 83/143, Computer Science Division (EECS), University of California, Berkeley, CA 94720, November 1983.

- [Borr87] G. Borriello, A.R. Cherson, P.B. Danzig, and M.N. Nelson, "RISCs vs. CISCs for Prolog: a Case Study," in *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, October 1987, pp. 136–145.
- [Brig89] P. Briggs, K.D. Cooper, K. Kennedy, and L. Torczon, "Coloring Heuristics for Register Allocation," in *SIGPLAN'89 Conference on Programming Language Design and Implementation*, June 1989, pp. 275–284.
- [Bush87] W.R. Bush, A.D. Samples, D. Ungar, and P.N. Hilfinger, "Compiling Smalltalk-80 to a RISC," in *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, October 1987, pp. 112–116.
- [Call86] D. Callahan, K.D. Cooper, K. Kennedy, and L. Torczon, "Interprocedural Constant Propagation," in *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, June 1986, pp. 152–161.
- [Camp85] W.B. Campbell, *The Efficient Modeling of Processor Behavior and Performance*, Master's thesis, Department of Computer Science, University of Utah, December 1985.
- [CEL85] *ACCEL Architecture Overview*, Celerity Computing, San Diego, CA 92126, September 1985.
- [Chai81] G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, and P.W. Markstein, "Register Allocation via Coloring," *Computer Languages*, vol. 6, 1981, pp. 47–57.
- [Chai82] G.J. Chaitin, "Register Allocation & Spilling via Graph Coloring," in *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*. June 1982, pp. 98–105.
- [Chan88] A. Chang and M.F. Mergen, "801 Storage: Architecture and Programming," *ACM Transactions on Computer Systems*, vol. 6, no. 1, February 1988, pp. 28–50.
- [Chen85] C. Chen, S. Kong, D. Lee, and T. Stetzler, "Design Notes for the SPUR Processor," in R.H. Katz, editor, *Proceedings of CS292i: Implementation of VLSI Systems (Spring 1985)*, Computer Science Division (EECS), University of California, Berkeley, CA 94720, September 1985. Published as Technical Report UCB/CSD 86/259.
- [Chow83] F.C. Chow, *A Portable Machine-Independent Global Optimizer—Design and Measurements*, PhD dissertation, Computer Systems Laboratory, Stanford University, Stanford, CA 94305-2192, December 1983. Published as Technical Report 83-254.

- [Chow84] F. Chow and J.L. Hennessy, "Register Allocation by Priority-based Coloring," in *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, June 1984, pp. 222–232.
- [Chow86] F. Chow, M. Himmelstein, E. Killian, and L. Weber, "Engineering a RISC Compiler System," in *COMPCON '86*, 1986, pp. 132–137.
- [Chow87a] F. Chow, S. Correll, M. Himmelstein, E. Killian, and L. Weber, "How Many Addressing Modes Are Enough?," in *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, October 1987, pp. 117–121.
- [Chow87b] P. Chow and M. Horowitz, "Architectural Tradeoffs in the Design of MIPS-X," in *Proceedings of the 14th Annual International Symposium on Computer Architecture*, June 1987, pp. 300–308.
- [Chow88] F.C. Chow, "Minimizing Register Usage Penalty at Procedure Calls," in *SIGPLAN'88 Conference on Programming Language Design and Implementation*, June 1988, pp. 85–94.
- [Chu74] Y. Chu, editor, "Special Issue on Hardware Description Languages," *IEEE Computer*, vol. 7, December 1974, pp. 18–51.
- [Clar80] D.W. Clark and W.D. Strecker, "Comments on 'The Case for the Reduced Instruction Set Computer,'" *Computer Architecture News*, vol. 8, no. 6, October 1980, pp. 34–38.
- [Clar82] D.W. Clark and H.M. Levy, "Measurement and Analysis of Instruction Use in the VAX-11/780," in *Proceedings of the 9th Symposium on Computer Architecture*, 1982, pp. 9–17.
- [Cohe88] P.E. Cohen, "An Abundance of Registers," *SIGPLAN Notices*, vol. 23, no. 6, June 1988, pp. 24–34.
- [Colw85] R.P. Colwell *et al.*, "Computers, Complexity, and Controversy," *Computer*, vol. 18, no. 9, September 1985, pp. 8–20.
- [Coop86] K.D. Cooper, K. Kennedy, and L. Torczon, "Interprocedural Optimization: Eliminating Unnecessary Recompile," in *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, June 1986, pp. 58–67.
- [Cort87a] J. Cortadella, *Mecanismos para la ejecución eficiente de los saltos en arquitecturas RISC*, PhD dissertation, Facultat d'Informàtica, Universitat Politècnica de Catalunya, June 1987.
- [Cort87b] J. Cortadella and J.M. Llabería, "Low Cost Evaluation Methodology for New Architectures," in *Proceedings of the 5th International Symposium on Applied Informatics*, February 1987, pp. 192–195.

- [Cort88] J. Cortadella and T. Jové, "Designing a Branch Target Buffer for Executing Branches with Zero Time Cost in a RISC Processor," in *Proceedings of the 14th Symposium on Microprocessing and Microprogramming*, August 1988.
- [Cout86a] D.S. Coutant, "Retargetable High-Level Alias Analysis," in *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages*, January 1986, pp. 110–118.
- [Cout86b] D.S. Coutant, C.L. Hammond, and J.W. Kelley, "Compilers for the New Generation of Hewlett-Packard Computers," *Hewlett-Packard Journal*, January 1986, pp. 4–18.
- [Curn76] H.J. Curnow and B.A. Wichmann, "A Synthetic Benchmark," *The Computer Journal*, vol. 19, no. 1, February 1976, pp. 43–49.
- [Dann79] R.B. Dannenberg, "An Architecture with Many Operand Registers To Efficiently Execute Block-Structured Languages," in *Proceedings of the 6th Annual Symposium on Computer Architecture*, April 1979, pp. 50–57.
- [Davi84] J.W. Davidson and C.W. Fraser, "Register Allocation and Exhaustive Peephole Optimization," *Software Practice & Experience*, vol. 14, no. 9, September 1984, pp. 857–865.
- [Davi87] J.W. Davidson and R.A. Vaughan, "The Effect of Instruction Set Complexity on Program Size and Memory Performance," in *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, October 1987, pp. 60–64.
- [Day70] W.H.E. Day, "Compiler Assignment of Data Items to Registers," *IBM Systems Journal*, vol. 9, no. 4, 1970, pp. 281–317.
- [DEC79] *VAX-11 Architecture Handbook*, Digital Equipment Corporation, 1979.
- [DeMo86] M. DeMoney, J. Moore, and J. Mashey, "Operating System Support on a RISC," in *COMPCON '86*, 1986, pp. 138–143.
- [Ditz82] D.R. Ditzel and H.R. McLellan, "Register Allocation for Free: The C Machine Stack Cache," in *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, March 1982, pp. 48–56.
- [Ditz87a] D.R. Ditzel and H.R. McLellan, "Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero," in *Proceedings of the 14th Annual International Symposium on Computer Architecture*, June 1987, pp. 2–9.
- [Ditz87b] D.R. Ditzel, H.R. McLellan, and A.D. Berenbaum, "Design Tradeoffs To Support the C Programming Language in the CRISP Microprocessor," in *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, October 1987, pp. 158–163.

- [Ditz87c] D.R. Ditzel, H.R. McLellan, and A.D. Berenbaum, "The Hardware Architecture of the CRISP Microprocessor," in *Proceedings of the 14th Annual International Symposium on Computer Architecture*, June 1987, pp. 309–319.
- [Eick87] R.J. Eickemeyer and J.H. Patel, "Performance Evaluation of Multiple Register Sets," in *Proceedings of the 14th Annual International Symposium on Computer Architecture*, June 1987, pp. 264–271.
- [Elli86] J.R. Ellis, *A Compiler for VLIW Architectures*, The MIT Press, 1986.
- [Ferr78] D. Ferrari, *Computer Systems Performance Evaluation*, Prentice-Hall, 1978.
- [Feue82] A.R. Feuer and N.H. Gehani, "A Comparison of the Programming Languages C and PASCAL," *ACM Computing Surveys*, vol. 14, no. 1, March 1982, pp. 73–92.
- [Fish84] J.A. Fisher, J.R. Ellis, J.C. Ruttenberg, and A. Nicolau, "Parallel Processing: a Smart Compiler and a Dumb Machine," in *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, June 1984, pp. 37–47.
- [Fitz82] D.T. Fitzpatrick, J.K. Foderaro, M.G.H. Katevenis, H.A. Landman, D.A. Patterson, J.B. Peek, Z. Peshkess, C.H. Séquin, R.W. Sherburne, and K.S. Van Dyke, "A RISCy Approach to VLSI," *Computer Architecture News*, vol. 10, no. 1, March 1982, pp. 28–32.
- [Flynn87] M.J. Flynn, C.L. Mitchell, and J.M. Mulder, "And Now a Case for Complex Instruction Sets," *IEEE Computer*, vol. 20, no. 9, September 1987, pp. 71–83.
- [Folgs83] D. Folger and E. Basart, "Computer Architecture—Designing for Speed," in *COMPCON '83*, Spring 1983, pp. 25–31.
- [Fotl87] D.A. Fotland, J.F. Shelton, W.R. Bryg, R.V. La Fetra, S.I. Boschma, A.S. Yeh, and E.M. Jacobs, "Hardware Design of the First HP Precision Architecture Computers," *Hewlett-Packard Journal*, March 1987, pp. 4–17.
- [Fraz87] G.L. Frazier, *The Trickle-Back Register File: Design and Analysis*, cs259 Class Report, Department of Computer Science, University of California, Los Angeles, CA 90024, December 1987.
- [Frei74] R.A. Freiburghouse, "Register Allocation via Usage Counts," *Communications of the ACM*, vol. 17, no. 11, November 1974, pp. 638–642.
- [Full77] S.H. Fuller, P. Shaman, D. Lamb, and W.E. Burr, "Evaluation of Computer Architectures via Test Programs," in *Proceedings of the National Computer Conference*, AFIPS Press, 1977.
- [Furh85] B. Furht, "RISC Architectures with Multiple Overlapping Windows," in *Compsac '85*, October 1985.

- [Furh88] B. Furht, "A RISC Architecture with Two-Size, Overlapping Register Windows," *IEEE Micro*, vol. 8, no. 2, April 1988, pp. 67–80.
- [Garb77] B.S. Garbow, J.M. Boyle, J.J. Dongarra, and C.B. Moler, *Matrix Eigen System Routines—EISPACK Guide Extension*, Springer-Verlag, 1977.
- [Garn88] R.B. Garner *et al.*, "The Scalable Processor Architecture (SPARC)," in *COMPCON'88*, Spring 1988, pp. 278–283.
- [Gibs85] G. Gibson, "SPURBus Specification," in R.H. Katz, editor, *Proceedings of CS292i: Implementation of VLSI Systems (Spring 1985)*, Computer Science Division (EECS), University of California, Berkeley, CA 94720, September 1985. Published as Technical Report UCB/CSD 86/259.
- [Gilb81] J. Gilbreath, "A High-Level Language Benchmark," *BYTE*, vol. 6, no. 9, September 1981, pp. 180–198.
- [Gima87] C.E. Gimarc and V.M. Milutinovic, "A Survey of RISC Processors and Computers of the Mid-1980s," *Computer*, vol. 20, no. 9, September 1987, pp. 59–69.
- [Good85] J.R. Goodman, J. Hsieh, K. Liou, A.R. Pleszkun, P.B. Schechter, and H.C. Young, "PIPE: a VLSI Decoupled Architecture," in *Proceedings of the 12th Annual International Symposium on Computer Architecture*, June 1985, pp. 20–27.
- [Good88] J.R. Goodman and W-C. Hsu, "Code Scheduling and Register Allocation in Large Basic Blocks," in *1988 International Conference on Supercomputing*, July 1988, pp. 242–252.
- [Gros83] T. Gross and J. Gill, *A Short Guide to MIPS Assembly Instructions*, Technical Report 83-236, Computer Systems Laboratory, Stanford University, Stanford, CA 94305, November 1983.
- [Gros84] T. Gross, J.L. Hennessy, N. Jouppi, S. Przybylski, C. Rowen, A. Agarwal, and P. Steenkiste, "A Perspective on High-Level Language Architecture," in *Proceedings of the International Workshop on High-Level Computer Architecture 84*, May 1984, pp. 3.12–3.14.
- [Gupt89] R. Gupta, M.L. Soffa, and T. Steele, "Register Allocation via Clique Separators," in *SIGPLAN'89 Conference on Programming Language Design and Implementation*, June 1989, pp. 264–274.
- [Hans85] P. Hansen, "Coprocessor Interface Description," in R.H. Katz, editor, *Proceedings of CS292i: Implementation of VLSI Systems (Spring 1985)*, Computer Science Division (EECS), University of California, Berkeley, CA 94720, September 1985. Published as Technical Report UCB/CSD 86/259.
- [Heat84] J.L. Heath, "Re-evaluation of RISC I," *Computer Architecture News*, vol. 12, no. 1, March 1984, pp. 3–10.

- [Hech77] M.S. Hecht, *Flow Analysis of Computer Programs*, Elsevier North-Holland, 1977.
- [Henn82] J.L. Hennessy, N. Jouppi, J. Gill, R. Baskett, A. Strong, T. Gross, C. Rowen, and J. Leonard, "The MIPS Machine," in *COMPCON '82*, Spring 1982, pp. 2-7.
- [Henn83a] J.L. Hennessy, N. Jouppi, F. Baskett, and J. Gill, *MIPS: a VLSI Processor Architecture*, Technical Report 223, Computer Systems Laboratory, Stanford University, Stanford, CA 94305, 1983.
- [Henn83b] J.L. Hennessy, N. Jouppi, S. Przybylski, C. Rowen, and T. Gross, *Design of a High Performance VLSI Processor*, Technical Report 236, Computer Systems Laboratory, Stanford University, Stanford, CA 94305, February 1983.
- [Henn84] J.L. Hennessy, "VLSI Processor Architecture," *IEEE Transactions on Computers*, vol. C-33, no. 12, December 1984, pp. 1221-1246.
- [Henr84] R.R. Henry, *Graham-Glanville Code Generators*, PhD dissertation, Computer Science Division (EECS), University of California, Berkeley, CA 94720, May 1984. Published as Technical Report UCB/CSD 84/184.
- [Hill67] F.S. Hillier and G.J. Lieberman, *Introduction to Operations Research*, Holden-Day, 1967.
- [Hill83] D.D. Hill, "An Analysis of C Machine Support for Other Block-Structured Languages," *Computer Architecture News*, vol. 11, no. 4, September 1983, pp. 7-16.
- [Hill86] M.D. Hill *et al.*, "Design Decisions in SPUR," *IEEE Computer*, vol. 19, no. 11, 1986, pp. 8-22.
- [Hitc85] C.Y. Hitchcock III and H.M. Brinkley Sprunt, "Analyzing Multiple Register Sets," in *Proceedings of the 12th Annual International Symposium on Computer Architecture*, June 1985, pp. 55-63.
- [Hitc86] C.Y. Hitchcock III, *Addressing Modes for Fast and Optimal Code Generation*, PhD dissertation, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213, December 1986. Published as Technical Report CMU-CS-86-179.
- [Hopk84] M.E. Hopkins, "A Definition of RISC," in *Proceedings of the International Workshop on High-Level Computer Architecture 84*, May 1984, pp. 3.8-3.11.
- [Horw66] L.P. Horwitz, R.M. Karp, R.E. Miller, and S. Winograd, "Index Register Allocation," *Journal of the ACM*, vol. 13, no. 1, January 1966, pp. 43-61.
- [Hsu87] W.-C. Hsu, *Register Allocation and Code Scheduling for Load/Store Architectures*, PhD dissertation, Computer Sciences Department, University of Wisconsin-Madison, October 1987. Published as Technical Report #722.

- [Huck83] J.C. Huck, *Comparative Analysis of Computer Architectures*, Technical Report 83-243, Computer Systems Laboratory, Stanford University, Stanford, CA. 94305, May 1983.
- [Hugu85a] M. Huguet, *A C-Oriented Register Set Design*, Master's thesis, University of California, Los Angeles, CA 90024, June 1985. Published as Technical Report CSD-850019.
- [Hugu85b] M. Huguet and T. Lang, "A C-Oriented Register Set Design," in *Proceedings of the 29th Symposium on Mini and Microcomputers and Their Applications*, June 1985, pp. 182-189.
- [Hugu85c] M. Huguet and T. Lang, "A Reduced Register File for RISC Architectures," *Computer Architecture News*, vol. 13, no. 4, September 1985, pp. 22-31.
- [Hugu87] M. Huguet, T. Lang, and Y. Tamir, "A Block-and-Actions Generator as an Alternative to a Simulator for Collecting Architecture Measurements," in *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, June 1987, pp. 14-25.
- [Hwu89] W.W. Hwu and P.P. Chang, "Inline Function Expansion for Compiling C Programs," in *SIGPLAN'89 Conference on Programming Language Design and Implementation*, June 1989, pp. 246-257.
- [John73] R.K. Johnson, *A Survey of Register Allocation*, Technical Report, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA 15213, May 1973.
- [John75] R.K. Johnson, *An Approach to Global Register Allocation*, PhD dissertation, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213, December 1975.
- [John78] S.C. Johnson, "A Portable Compiler: Theory and Practice," in *Annual ACM Symposium on Principles Programming Languages*, January 1978.
- [John79] S.C. Johnson, "A Tour Through the Portable C Compiler," in *UNIX Programmer's Manual for Advanced Programmers*, Bell Telephone Laboratories, Murray Hill, New Jersey 07974, January 1979.
- [John81] S.C. Johnson, "Position Paper on Optimizing Compilers," in *8th Annual ACM Symposium on Principles of Programming Languages*, January 1981, pp. 88-89.
- [John82] R. Johnson and J. Wick, "An Overview of the Mesa Processor Architecture," in *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, March 1982, pp. 20-29.
- [John87] M.G. Johnson, "SPICE: Floating-Point Benchmark," Usenet Distribution, 1987.

- [Karg89] P.A. Karger, "Using Registers To Optimize Cross-Domain Call Performance," in *Proceedings of the Third Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, April 1989, pp. 194-204.
- [Kate83] M.G.H. Katevenis, *Reduced Instruction Set Computer Architectures for VLSI*, PhD dissertation, Computer Science Division (EECS), University of California, Berkeley, CA 94720, October 1983. Published as Technical Report UCB/CSD 83/141.
- [Katz85] R.H. Katz, "SPUR Architecture Design Rationale," in *Proceedings of CS292i: Implementation of VLSI Systems (Spring 1985)*, Computer Science Division (EECS), University of California, Berkeley, CA 94720, September 1985. Published as Technical Report UCB/CSD 86/259.
- [Keed78a] J.L. Keedy, "On the Evaluation of Expressions Using Accumulators, Stacks and Storage-to-Storage Instructions," *Computer Architecture News*, vol. 7, no. 4, December 1978, pp. 24-27.
- [Keed78b] J.L. Keedy, "On the Use of Stacks in the Evaluation of Expressions," *Computer Architecture News*, vol. 6, no. 6, February 1978, pp. 22-28.
- [Keed79] J.L. Keedy, "More on the Use of Stacks in the Evaluation of Expressions," *Computer Architecture News*, vol. 7, no. 8, June 1979, pp. 18-22.
- [Keed83] J.L. Keedy, "An Instruction Set for Evaluating Expressions," *IEEE Transactions on Computers*, vol. C-32, no. 5, May 1983, pp. 476-478.
- [Kern78] B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.
- [Kern81] B.W. Kernighan, *Why PASCAL Is Not My Favorite Programming Language*, Technical Report, Bell Laboratories, Murray Hill, New Jersey 07974, July 1981.
- [Kess83] P.B. Kessler, *The Intermediate Representation of the Portable C Compiler, as Used by the Berkeley Pascal Compiler*, Internal Report, Computer Science Division (EECS), University of California, Berkeley, CA 94720, April 1983.
- [Kess86] R.R. Kessler *et al.*, "EPIC—A Retargetable, Highly Optimizing LISP Compiler," in *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, June 1986, pp. 118-130.
- [Kim78] J. Kim, *Spill Placement Optimization in Register Allocation for Compilers*, Technical Report RC 7251, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598, August 1978.
- [Kral80] M. Kralej, R. Rettberg, P. Herman, R. Bressler, and A. Lake, "Design of a User-Microprogrammable Building Block," in *Proceedings of the 13th Annual Workshop on Microprogramming*, December 1980, pp. 106-114.

- [Lamp82] B. Lampson, "Fast Procedure Calls," in *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, March 1982, pp. 66–76.
- [Lang86] T. Lang and M. Huguet, "Reduced Register Saving/Restoring in Single-Window Register Files," *Computer Architecture News*, vol. 4, no. 3, June 1986, pp. 17–26.
- [Laru82] J.R. Larus, "A Comparison of Microcode, Assembly Code, and High-Level Languages on the VAX-11 and RISC I," *Computer Architecture News*, vol. 10, no. 5, September 1982, pp. 10–15.
- [Laru86] J.R. Larus and P.N. Hilfinger, "Register Allocation in the SPUR LISP Compiler," in *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, 1986, pp. 255–263.
- [Leve83] B.W. Leverett, *Register Allocation in Optimizing Compilers*, UMI Research Press, 1983.
- [Levy82] H.M. Levy and D.W. Clark, "On the Use of Benchmarks for Measuring System Performance," *Computer Architecture News*, vol. 10, no. 6, December 1982, pp. 5–8.
- [Lion79] J. Lions, *The Second Pass of the Portable C Compiler*, Bell Laboratories, Murray Hill, NJ 07974, June 1979.
- [Lowr69] E.S. Lowry and C.W. Medlock, "Object Code Optimization," *Communications of the ACM*, vol. 12, no. 1, January 1969, pp. 13–22.
- [Lucc67] F. Luccio, "A Comment on Index Register Allocation," *Communications of the ACM*, vol. 10, no. 9, September 1967, pp. 572–574.
- [Lund77] A. Lunde, "Empirical Evaluation of Some Features of Instruction Set Processor Architectures," *Communications of the ACM*, vol. 20, no. 3, March 1977, pp. 143–153.
- [MacL84] M.D. MacLaren, "Inline Routines in VAXELN Pascal," in *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, June 1984, pp. 266–274.
- [Mage87] D.J. Magenheimer, L. Peters, and D. Zuras, "Integer Multiplication and Division on the HP Precision Architecture," in *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, October 1987, pp. 90–99.
- [Mage88] D.J. Magenheimer, L. Peters, K.W. Pettis, and D. Zuras, "Integer Multiplication and Division on the HP Precision Architecture," *IEEE Transactions on Computers*, vol. 37, no. 8, August 1988, pp. 980–990.

- [Maho86] M.J. Mahon, R. Lee, T.C. Miller, J.C. Huck, and W.R. Bryg, "Hewlett-Packard Precision Architecture: The Processor," *Hewlett-Packard Journal*, August 1986, pp. 4–20.
- [May87] C. May, "MIMIC: a Fast System/370 Simulator," in *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, June 1987, pp. 1–13.
- [McDa82] G. McDaniel, "An Analysis of a Mesa Instruction Set Using Dynamic Instruction Frequencies," in *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, March 1982, pp. 167–176.
- [McKu84] M.K. McKusick, *Register Allocation and Data Conversion in Machine Independent Code Generators*, PhD dissertation, Computer Science Division (EECS), University of California, Berkeley, CA 94720, December 1984. Published as Technical Report UCB/CSD 84/214.
- [Miro82] J.C. Miros, *A C Compiler for RISC I*, Master's thesis, Computer Science Division (EECS), University of California, Berkeley, CA 94720, August 1982.
- [Mous86] J. Moussouris *et al.*, "A CMOS RISC Processor with Integrated System Functions," in *COMPCON '86*, 1986, pp. 126–131.
- [Muns86] A.A. Munshi and K.M. Schimpf, *Register Allocation via Two Colored Pebbling*, Technical Report UCSC-CRL-86-17, Computer Research Laboratory, University of California, Santa Cruz, CA 95064, July 1986.
- [Myer77] G.J. Myers, "The Case against Stack-Oriented Instruction Sets," *Computer Architecture News*, vol. 6, no. 3, August 1977, pp. 7–10.
- [Myer78] G.J. Myers, "The Evaluation of Expressions in a Storage-to-Storage Architecture," *Computer Architecture News*, vol. 6, no. 9, June 1978, pp. 20–23.
- [Myer82] G.J. Myers, *Advances in Computer Architecture*, John Wiley & Sons, 1982.
- [Naka67] I. Nakata, "On Compiling Algorithms for Arithmetic Expressions," *Communications of the ACM*, vol. 10, no. 8, August 1967, pp. 492–494.
- [Neff86] L. Neff, "CLIPPER Microprocessor Architecture Overview," in *COMPCON '86*, March 1986, pp. 191–195.
- [Nort83] R.L. Norton and J.A. Abraham, "Adaptive Interpretation as a Means of Exploiting Complex Instruction Sets," in *Proceedings of the 10th Annual International Symposium on Computer Architecture*, June 1983, pp. 277–282.
- [Olle85] B. Ollerton, *Performance Architecture for the UNIX Environment*, Technical Report, Celerity Computing, San Diego, CA 92126, 1985.

- [Patt80] D.A. Patterson and D.R. Ditzel, "The Case for the Reduced Instruction Set Computer," *Computer Architecture News*, vol. 8, no. 6, October 1980, pp. 25-33.
- [Patt82a] D.A. Patterson and R.S. Piepho, "RISC Assessment: a High-Level Language Experiment," in *Proceedings of the 9th Symposium on Computer Architecture*, April 1982, pp. 3-8.
- [Patt82b] D.A. Patterson and C.H. Séquin, "A VLSI RISC," *IEEE Computer*, vol. 15, no. 9, September 1982, pp. 8-21.
- [Patt83] D.A. Patterson, P. Garrison, M. Hill, D. Lioupis, C. Nyberg, T. Sippel, and K. Van Dyke, "Architecture of a VLSI Instruction Cache for a RISC," in *Proceedings of the 10th Annual International Symposium on Computer Architecture*, June 1983, pp. 108-116.
- [Patt84] D.A. Patterson, "RISC Watch," *Computer Architecture News*, vol. 12, no. 1, March 1984, pp. 11-19.
- [Patt85a] D.A. Patterson, "Reduced Instruction Set Computers," *Communications of the ACM*, vol. 28, no. 1, January 1985, pp. 8-21.
- [Patt85b] D.A. Patterson and J.L. Hennessy, "Response to 'Computers, Complexity, and Controversy,'" *IEEE Computer*, vol. 18, no. 11, November 1985, pp. 142-143.
- [Peek83] J.B. Peek, *The VLSI Circuitry of RISC I*, Technical Report UCB/CSD 83/135, Computer Science Division (EECS), University of California, Berkeley, CA 94720, August 1983.
- [Perk79] D.R. Perkins and R.L. Sites, "Machine-Independent PASCAL Code Optimization," in *Proceedings of the SIGPLAN Symposium on Compiler Construction*, August 1979, pp. 201-207.
- [Piep81] R.S. Piepho, *Comparative Evaluation of the RISC I Architecture via the Computer Family Architecture Benchmarks*, Master's thesis, Computer Science Division (EECS), University of California, Berkeley, CA 94720, August 1981.
- [Pond83] C. Ponder, *... But Will RISC Run LISP? (a Feasibility Study)*, Technical Report UCB/CSD 83/122, Computer Science Division (EECS), University of California, Berkeley, CA 94720, August 1983.
- [Post83] E. Post, "Real Programmers Don't Use PASCAL," Usenet Distribution, 1983.
- [Powe84] M.L. Powell, "A Portable Optimizing Compiler for Modula-2," in *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, June 1984, pp. 310-318.

- [Przy84] S.A. Przybylski, T.R. Gross, J.L. Hennessy, N.P. Jouppi, and C. Rowen, "Organization and VLSI Implementation of MIPS," *Journal of VLSI and Computer Systems*, vol. 1, no. 2, 1984, pp. 170-208.
- [PTC83] *RISC Theory as Applied in the Pyramid 90z*, Pyramid Technology Corporation, San Diego, CA 94039, September 1983.
- [Radi82] G. Radin, "The 801 Minicomputer," in *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, March 1982, pp. 39-47.
- [Raga83] R. Ragan-Kelley and R. Clark, "Applying RISC Theory to a Large Computer," *Computer Design*, November 1983, pp. 191-198.
- [Redz69] R.R. Redziejowski, "On Arithmetic Expressions and Trees," *Communications of the ACM*, vol. 12, no. 2, February 1969, pp. 81-84.
- [Rich89] S. Richardson and H. Ganapathi, "Code Optimization across Procedures," *IEEE Computer*, vol. 22, no. 2, February 1989, pp. 42-50.
- [RID83a] *RIDGE Assembler Reference Manual*, Ridge Computers, Santa Clara, CA 95054, October 1983.
- [RID83b] *RIDGE C Programming Notes*, Ridge Computers, Santa Clara, CA 95054, 1983.
- [Ritc79] D.M. Ritchie, "A Tour through the UNIX C Compiler," in *UNIX Programmer's Manual for Advanced Programmers*, Bell Telephone Laboratories, Murray Hill, NJ 07974, January 1979.
- [Ritc85] S.A. Ritchie, *TLB for Free: In-Cache Address Translation for a Multiprocessor Workstation*, Technical Report UCB/CSD 85/233, Computer Science Division (EECS), University of California, Berkeley, CA 94720, May 1985.
- [Rose84] C.W. Rose, G.M. Ordy, and P.J. Drongowski, "N.mPc: a Study in University-Industry Technology Transfer," *IEEE Design & Test*, February 1984, pp. 44-56.
- [Ryde86] B.G. Ryder and M.C. Paull, "Elimination Algorithms for Data Flow Analysis," *ACM Computing Surveys*, vol. 18, no. 3, September 1986, pp. 277-316.
- [Séqui82] C.H. Séquin and D.A. Patterson, *Design and Implementation of RISC I*, Technical Report UCB/CSD 82/106, Computer Science Division (EECS), University of California, Berkeley, CA 94720, October 1982.
- [Sach85] H. Sachs and W. Hollingsworth, "A High Performance 846,000 Transistor UNIX Engine—The Fairchild CLIPPER," in *IEEE International Conference on Computer Design: VLSI in Computers*, October 1985, pp. 342-346.

- [Samp85] A.D. Samples, M. Klein, and P. Foley, *SOAR Architecture*, Technical Report UCB/CSD 85/226, Computer Science Division (EECS), University of California, Berkeley, CA 94720, March 1985.
- [Samp86] A.D. Samples, D. Ungar, and P. Hilfinger, "SOAR: Smalltalk Without Bytecodes," in *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, September 1986, pp. 107–118.
- [Scha89] F. Schaffa, *Communications on VLSI*, PhD dissertation, Computer Science Department, University of California, Los Angeles, CA 90024-159610, September 1989.
- [Sche77] R.W. Scheifler, "An Analysis of Inline Substitution for a Structured Programming Language," *Communications of the ACM*, vol. 20, no. 9, September 1977, pp. 647–654.
- [Schu77] P.T. Schulthess and E.P. Mumprecht, "Reply to the Case against Stack-Oriented Instruction Sets," *Computer Architecture News*, vol. 6, no. 5, December 1977, pp. 24–27.
- [Seth70] R. Sethi and J.D. Ullman, "The Generation of Optimal Code for Arithmetic Expressions," *Journal of the Association for Computing Machinery*, vol. 17, no. 4, October 1970, pp. 715–728.
- [Seth75] R. Sethi, "Complete Register Allocation Problems," *SIAM J. Comput.*, vol. 4, no. 3, September 1975, pp. 226–248.
- [Sher84] R.W. Sherburne, Jr., *Processor Design Tradeoffs in VLSI*, PhD dissertation, Computer Science Division (EECS), University of California, Berkeley, CA 94720, April 1984. Published as a Technical Report UCB/CSD 84/173.
- [Shus78] L.J. Shustek, *Analysis and Performance of Computer Instruction Sets*, PhD dissertation, Stanford Linear Accelerator Center, Stanford University, Stanford, CA 94305, January 1978. Published as Technical Report STAN-CS-78-658.
- [Site78] R.L. Sites, "A Combined Register-Stack Architecture," *Computer Architecture News*, vol. 6, no. 8, April 1978, p. 19.
- [Site79a] R.L. Sites, "How to Use 1000 Registers," in *Caltech Conference on VLSI*, January 1979, pp. 527–532.
- [Site79b] R.L. Sites, "Machine-Independent Register Allocation," in *Proceedings of the SIGPLAN Symposium on Compiler Construction*, August 1979, pp. 221–225.
- [Smit82] A.J. Smith, "Cache Memories," *ACM Computing Surveys*, vol. 14, no. 3, September 1982, pp. 473–530.
- [Smit85] J.E. Smith and J.R. Goodman, "Instruction Cache Replacement Policies and Organizations," *IEEE Transactions on Computers*, vol. C-34, no. 3, March 1985, pp. 234–241.

- [Stal88] R.M. Stallman, *Internals of GNU CC*, Free Software Foundation, 675 Mass Ave., Cambridge, MA 02139, 1988.
- [Stan87] T.J. Stanley and R.G. Wedig, "A Performance Analysis of Automatically Managed Top of Stack Buffers," in *Proceedings of the 14th Annual International Symposium on Computer Architecture*, June 1987, pp. 272–281.
- [Stee80] G.L. Steele, Jr. and G.J. Sussman, "The Dream of a Lifetime: a Lazy Variable Extent Mechanism," in *Conference Record of the 1980 LISP Conference*, August 1980, pp. 163–172.
- [Stee87] P.A. Steenkiste, *LISP on a Reduced-Instruction-Set Processor: Characterization and Optimization*, PhD dissertation, Department of Electrical Engineering, Stanford University, Stanford, CA 94305, March 1987.
- [Stre78] W.D. Strecker, "VAX-11/780—A Virtual Address Extension to the DEC PDP-11 Family," in *AFIPS Conference Proceedings*, June 1978, pp. 967–980.
- [Svob76] L. Svobodova, *Computer Performance Measurement and Evaluation Methods: Analysis and Applications*, American Elsevier, 1976.
- [Swee82] R. Sweet and J.G. Sandman, Jr., "Static Analysis of the Mesa Instruction Set," in *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, March 1982, pp. 158–166.
- [Taba86] D. Tabak, "Which System Is a RISC?," *IEEE Computer*, vol. 19, no. 10, October 1986, pp. 85–86.
- [Tami81] Y. Tamir, *Simulation and Performance Evaluation of the RISC Architecture*, Technical Report UCB/ERL M81/17, Electronics Research Laboratory, University of California, Berkeley, CA 94720, March 1981.
- [Tami83] Y. Tamir and C.H. Séquin, "Strategies for Managing the Register File in RISC," *IEEE Transactions on Computers*, vol. C-32, no. 11, November 1983.
- [Tane83] A.S. Tanenbaum, H. van Staveren, E.G. Keizer, and J.W. Stevenson, "A Practical Tool Kit for Making Portable Compilers," *Communications of the ACM*, vol. 26, no. 9, September 1983, pp. 654–660.
- [Tayl85] G. Taylor, "SPUR Instruction Set Architecture," in R.H. Katz, editor, *Proceedings of CS292i: Implementation of VLSI Systems (Spring 1985)*, Computer Science Division (EECS), University of California, Berkeley, CA 94720, September 1985. Published as Technical Report UCB/CSD 86/259.
- [Tayl86] G.S. Taylor, P.N. Hilfinger, J.R. Larus, D.A. Patterson, and B.G. Zorn, "Evaluation of the SPUR LISP Architecture," in *Proceedings of the 13th Annual Symposium on Computer Architecture*, June 1986, pp. 444–452.
- [Trem87] M. Tremblay and T. Lang, "VLSI Implementation of a Shift-Register File," in *Proceedings of the Hawaii International Conference on System Sciences*, January 1987.

- [Unga84] D. Ungar, R. Blau, P. Foley, D. Samples, and D. Patterson, "Architecture of SOAR: Smalltalk on a RISC," in *Proceedings of the 11th Annual International Symposium on Computer Architecture*, 1984, pp. 188–197.
- [Unga86] D. Ungar, *The Design and Evaluation of a High Performance Smalltalk System*, PhD dissertation, Computer Science Division (EECS), University of California, Berkeley, CA 94720, 1986. Published as Technical Report UCB/CSD 86/287.
- [UNI81] *UNIX Programmer's Manual*, University of California, Berkeley, CA 94720, June 1981.
- [Vill85] S. Villalpando and T. Wisdom, "SPUR Cache Controller Datapath Description," in R.H. Katz, editor, *Proceedings of CS292i: Implementation of VLSI Systems (Spring 1985)*, Computer Science Division (EECS), University of California, Berkeley, CA 94720, September 1985. Published as Technical Report UCB/CSD 86/259.
- [Wall85] P. Wallich, "Toward Simpler, Faster Computers," *IEEE Spectrum*, vol. 22, no. 8, August 1985, pp. 38–45.
- [Wall86] D.W. Wall, "Global Register Allocation at Link Time," in *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, June 1986, pp. 264–275.
- [Wall87] D.W. Wall and M.L. Powell, "The Mahler Experience: Using an Intermediate Language as the Machine Description," in *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, October 1987, pp. 100–104.
- [Wall88] D.W. Wall, "Register Windows vs. Register Allocation," in *SIGPLAN'88 Conference on Programming Language Design and Implementation*, June 1988, pp. 67–78.
- [Wegm85] M.N. Wegman and F.K. Zadeck, "Constant Propagation with Conditional Branches," in *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages*, January 1985, pp. 291–299.
- [Weic84] R.P. Weicker, "Dhrystone: a Synthetic Systems Programming Benchmark," *Communications of the ACM*, vol. 27, no. 10, October 1984, pp. 1013–1030.
- [Weih80] W.E. Weihl, "Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables, and Label Variables," in *Conference Record of the 7th Annual ACM Symposium on Principles of Programming Languages*, January 1980, pp. 83–94.
- [Wein84] P.J. Weinberger, "Cheap Dynamic Instruction Counting," *AT&T Bell Laboratories Technical Journal*, vol. 63, no. 8, October 1984, pp. 1815–1826.

- [Wich76] B.A. Wichmann, "Ackermann's Function: a Study in the Efficiency of Calling Procedures," *BIT*, vol. 16, no. 1, 1976, pp. 103-110.
- [Wiec82] C. Wiecek, "A Case Study of VAX-11 Instruction Set Usage for Compiler Execution," in *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, March 1982, pp. 177-184.
- [Wirt86] N. Wirth, "Microprocessor Architectures: a Comparison Based on Code Generation by Compiler," *Communications of the ACM*, vol. 29, no. 10, October 1986, pp. 978-90.
- [Wong88] W.S. Wong and R.J.T. Morris, "Benchmark Synthesis Using the LRU Cache Hit Function," *IEEE Transactions on Computers*, vol. 37, no. 6, June 1988, pp. 637-645.
- [Wong89] W.F. Wong, "A Stack Addressing Scheme Based on Windowing," *Computer Architecture News*, vol. 17, no. 1, March 1989, pp. 63-69.
- [Wood86] D.A. Wood *et al.*, "An In-Cache Address Translation Mechanism," in *Proceedings of the 13th Annual Symposium on Computer Architecture*, June 1986, pp. 358-365.
- [Wulf71] W.A. Wulf, D.B. Russell, and A.N. Habermann, "BLISS: a Language for Systems Programming," *Communications of the ACM*, vol. 14, no. 12, December 1971, pp. 780-790.
- [Wulf75] W.A. Wulf, R.K. Johnsson, C.B. Weinstock, S.O. Hobbs, and C.M. Geschke, *The Design of an Optimizing Compiler*, American Elsevier, 1975.
- [Wulf81] W.A. Wulf, "Compilers and Computer Architecture," *IEEE Computer*, vol. 14, no. 7, July 1981, pp. 41-47.