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

ARCHITECTURAL AND COMPILER SUPPORT FOR EFFICIENT FUNCTION CALLS: A PROPOSAL

Miquel Huguet

June 1988
CSD-880030

Architectural and Compiler Support for Efficient Function Calls: A Proposal

A prospectus for Ph.D. dissertation

by

Miquel Huguet *

Computer Science Department
University of California, Los Angeles

June 1988

*The author has been partially supported by a FULBRIGHT/MEC Fellowship (1982-84), a CIRIT (*Generalitat de Catalunya*) Fellowship (1984-86), and a GTE Computer Science Fellowship (1987-88) during his studies at UCLA.

Abstract

With the current trend towards VLSI load/store architectures, registers have become one of the critical resources to increase processor performance. The use of registers reduces both the instruction and the data memory traffic. However, registers have to be saved every time a function (procedure) is called and restored when the function returns. One of the important factors that influences the design and use of a register file is the overhead produced by the register saving and restoring (RSR). Some current processors rely on hardware support, such as multiple-window register files, and/or compiler support to reduce this overhead. However, multiple-window architectures have mainly three 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. Fos this reason, 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 and leaf-function optimization. In this case, the complexity of the compiler is increased.

This work investigates both the support that the architecture can provide to implement efficient function calls for single-window architectures and the support that can be provided by the compiler. We present several architectural policies to reduce the RSR overhead for single-window architectures. These policies make use of *dynamic* information to determine which registers have been used during program execution. We evaluate these policies and compare them 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 our dynamic policies reduce significantly the RSR traffic with respect to the conventional static policies.

We also present six new compiler optimizations to reduce the RSR traffic. Two of them are based on live-variable analysis and are performed per function. The other four are applied to the whole program 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-existent compiler optimizations (leaf functions and live-variable analysis). We also plan to evaluate several *register allocation* schemes and show the data memory traffic reduction obtained by these schemes when they are used in conjunction with the architectural policies and with the compiler optimizations.

Finally, we plan to present the implementation of one of the dynamic policies in a RISC II-like processor. We expect to show that the implementation of this policy does not affect the processor cycle time (as multiple-window architectures do). We also expect to estimate the hardware (area) required for this implementation and the number of cycles necessary to perform the RSR operations, and to evaluate the speed-up obtained.

Contents

1 Introduction 1

1.1 The Problem and our Research Contributions 1

1.2 Previous Work 4

1.3 Related Work 26

2 Research Proposal and Expected Contributions 30

2.1 Architectural Support to Reduce the RSR Traffic 31

2.2 Compiler Support to Reduce the RSR Traffic 33

2.3 Register Allocation and Data Memory Traffic 35

2.4 Implementation of Policy G 36

2.5 BKGEN as a Tool for Collecting Dynamic Measurements 38

2.6 Programs Measured 42

2.7 Current Status and Future Work 44

3 Architectural Support for Register Saving/Restoring for Single-Window Register Files 45

3.1 Static Policies A and B 50

3.2 Policies C and D 52

3.3 Policies E, F, G, and H 55

3.4 Summary 61

4 Compiler Optimizations to Reduce Register Saving/Restoring 63

4.1 Intraprocedural Compiler Optimizations 64

4.1.1 Policy A-live 65
4.1.2 Policy A-live Optimized (A-lvOpt) 67
4.1.3 Policies B, C, and G with Leaf Functions 69
4.1.4 Policies C-live and G-live with Leaf Functions 72
4.1.5 Summary 72
4.2 Interprocedural Compiler Optimizations 74
4.2.1 Global A-live 77
4.2.2 Global A-lvOpt 83
4.2.3 Global B-lf 87
4.2.4 Global G-lf 93
4.2.5 Summary 95

5 Conclusions **97**

Bibliography **99**

Chapter 1

Introduction

In this introductory chapter we offer first a description of the problem that this work addresses and a summary of our contributions towards its solution (Section 1.1). Afterwards, we present a detailed summary of the work performed in this area (Section 1.2) and of some related work that is not directly discussed in this report (Section 1.3). The reader who is already familiar with the topic can read Section 1.1 and move directly to Chapter 2 which presents our research proposal and a more detailed discussion on the contributions that we expect this work will provide.

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, specially in *register allocation* optimization [Chai82,Ankl82,Chow83, Wall86,Stee87], have resulted in a better usage of the register set because the compiler can allocate more *simple* or *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. It 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 that influences the design and use of the register set is the overhead produced by the register saving and restoring (RSR) [Lund77,Patt82b].

For instance, Lunde [Lund77] affirms that the BLISS compiler spends 25% of its execution time to compile the program "Treesort" in call administration. Patterson and Séquin [Patt82b] have weighted the relative frequency of high-level language 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 generated by the C compilers for VAX-11, PDP-11, and MC68000). The RSR overhead becomes more significant for languages such as C [Kern78] which encourage the use of functions.

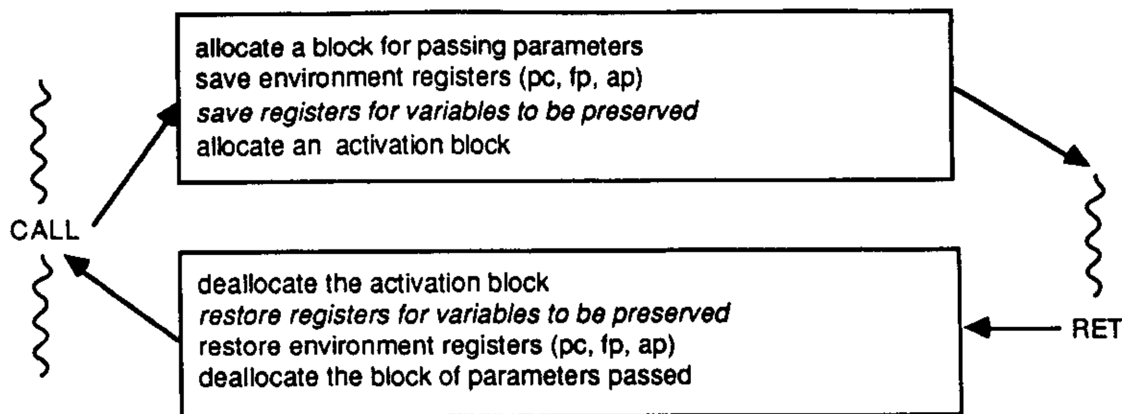


Figure 1.1: Operations to Be Performed on a Function Call

To reduce the RSR overhead some current processors rely on hardware support, such as multiple-window register files [Patt82b,Kate83], or compiler optimizations [Radi82,Chow84]. *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 mainly three 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]. These drawbacks are discussed in Section 1.2.1.

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 Section 1.2.3.4. 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 [Perk79,Alle80,Ausl82,Tane83,Powe84]. This is also true for the algorithms that perform register allocation [John75,Site79b,Leve83,Chow83].

This work 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 it would be executed much slower 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.

We present six 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. We evaluate these policies and compare them 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. The programs measured are *large typical* programs used in UNIX systems: the NROFF word processor, the SORT program, the VAX-11 assembler, and the Portable C Compiler for VAX-11 (see Section 2.6). 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. No measurements for these programs are available for multiple-window architectures at the moment of writing this proposal.

We also present six new compiler optimizations to reduce the RSR traffic. Two of them are based on live-variable analysis and are performed per function (*intraprocedural*): one is for Policy A and the second for Policy G. The new optimizations are also evaluated and compared to the already-existing compiler optimizations (live-variable analysis and leaf functions). We show that the optimized Policy A with live-variable analysis has between 59% (for 32 registers) and 64% (for 6) of the RSR traffic generated by the standard Policy A with live-variable analysis and between 53% (for 32) and 61% (for 6) of Policy B with leaf functions. The optimized dynamic Policy G with leaf functions generates the least traffic: it has between 22% (for 24) and 47% (for 6) of the RSR traffic generated by the optimized Policy A with live-variable analysis (which is the best among the static optimizations). No measurements are available yet for the new intraprocedural Policy G optimization with live-variable analysis.

The other four compiler optimizations are applied to the whole program (*interprocedural*) and their goal is to find a *register assignment* so 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. No measurements are available at the moment of writing this proposal. We expect to show that the interprocedural Policy G optimization will generate the least traffic as it has occurred in the previuos cases.

We also expect to show that to be able to use efficiently a large register set (more than 16 registers), the system has to provide some architectural support (i.e., Policy G for single-window architectures) and/or the interprocedural optimizations.

We plan to evaluate several *register allocation* approaches and show the data memory traffic reduction obtained by these approaches when they are used in conjunction with the architectural policies and with the compiler optimizations. Our goal is to show that when the RSR traffic is reduced, the compiler can allocate more variables to registers so that the overall traffic gets also reduced. Also, we would like to show that both the architectural policies and the compiler optimizations can benefit any register allocation approach, not only the one used to obtained the measurements for this proposal (this is explained in Section 2.1).

Finally, we plan to present the implementation of the dynamic Policy G in a RISC II-like pro-

cessor. We expect to show that the implementation of Policy G does not affect the processor cycle (as multiple-window architectures do) because the operations required are performed in parallel with the main CPU activities. We also expect to estimate the hardware (area) required for its implementation and the number of cycles required to perform the register saving and restoring operations, and to evaluate the speed-up obtained.

1.2 Previous Work

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

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

These are discussed in turn.

1.2.1 Multiple-Window Architectures

With the 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]. 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 1.2 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.

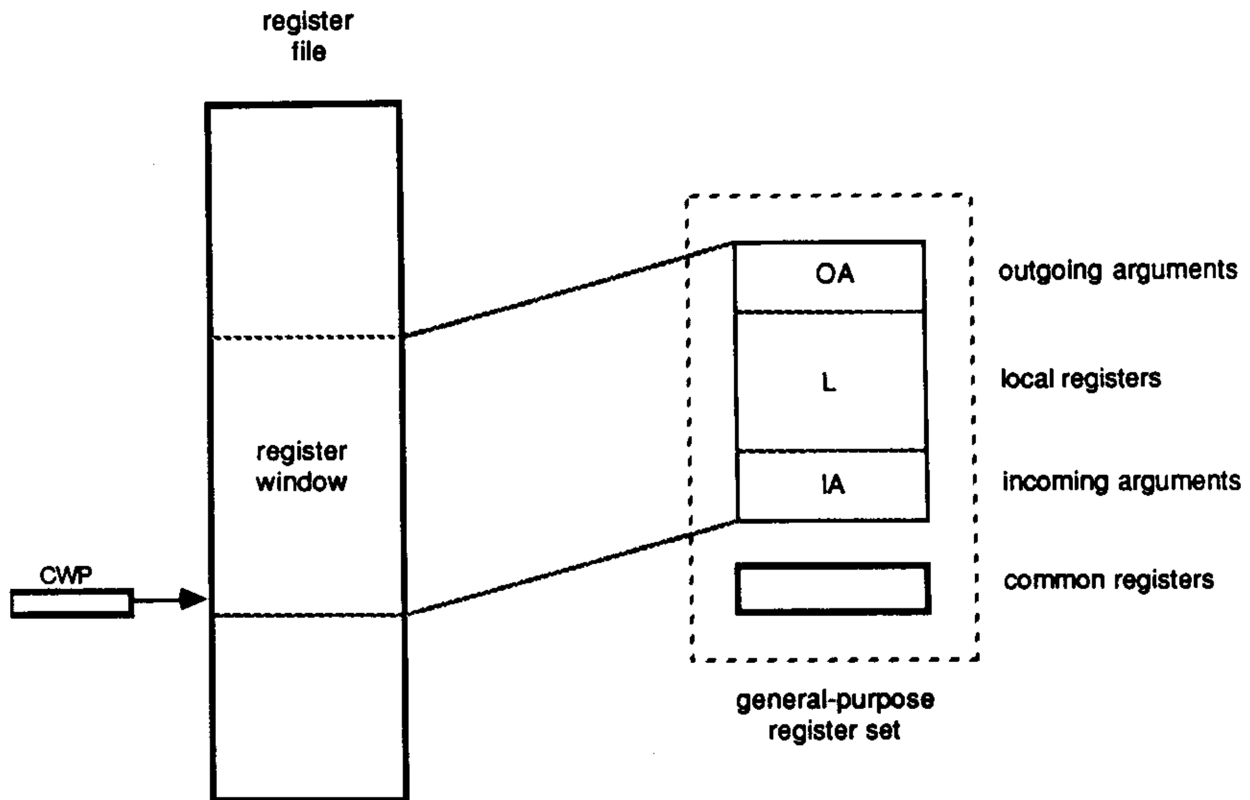


Figure 1.2: Multiple-Window Register Files

- A group of *local* registers. This 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.

We define the *window size* 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).

Most register file designs use *fixed-size windows* because of the simplicity in the implementation: C/70, RISC, PYRAMID 90x, C1200, SOAR, SPUR, and the LISP-Machine (see Table 1.1[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.

[1]The 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.

Processor	Description	References
C/70 (BBN Computer Co.)	• 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)	• 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.)	• variable-size windows • registers mapped to memory • register file size = 32 • 7 special registers (*psw*, *pc*, ...)	[Ditz82,Bere87a,Bere87b] [Ditz87a,Ditz87b,Ditz87c]
Pyramid 90x (Pyramid Technology)	• 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)	• As Pyramid, but with 8 register files for fast context switching • 4096 registers	[CEL85,Olle85]
SOAR (UCB)	• 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]
SPUR (UCB)	• general-purpose registers as RISC II • 15 floating-point registers • 7 special registers (*psw*'s, *pc*'s, reg. window ptrs.)	[Adam85,Chen85,Gibs85] [Hans85,Katz85,Ritc85] [Tayl85,Vill85,Hill86] [Tayl86,Wood86,Borr87]
Dragon (Xerox)	• 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 *pc*'s and context ptrs.	[Atki87]
LISP-Machine (Univ. of Miami)	• register file size = 64 window size = 8 overlapped registers = 4 circular buffer organization	[Aboa87]

Table 1.1: Multiple-Window Processors

Variable-size windows [Ditz82] assign per function the exact number registers required for the variables selected for allocation. Thus, no unused registers are present in the file. Only two of the machines in Table 1.1 (CRISP and Dragon) use variable-size windows. Although variable-size windows utilize the register file more efficiently because the exact number of registers required are 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 upon 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 has 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.

   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 to determine 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 also 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 1.1). 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.

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, 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. This 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 to need a window of 8 (with no local registers and 8 overlapped registers) [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%.

In spite that 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.

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. The windows that 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. This is 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 discuss some other performance studies that have been performed on multiple-window architectures. These can be classified into two categories:

- Studies that compare the execution speed of different processors [Piep81,Patt82a,Patt82b].
- Studies that compare the data memory reduction obtained by a processor with and without multiple windows [Hitc85,Eick87].

The first type of comparisons are done across different machines. The machines have different instruction sets, different number 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.

The second type of comparisons gives 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). The conclusion in both papers is that multiple-window architectures generate less memory traffic than single-window architectures. However, they use a nonoptimizing compiler for their measurements. This type of compilers usually perform a good 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 Section 1.2.3.1. Thus, the combination of a single-window architecture and an optimizing compiler might provide a different conclusion. As we will discuss in Chapter 2, we expect to evaluate the data memory traffic reduction caused by different architectural approaches and compiler optimizations.

1.2.2 Single-Window Architectures

In spite of the RSR traffic reduction that multiple-window architectures offer, several processors prefer to have a conventional general-purpose register set (see Table 1.2[2]). The motivation for this is the drawbacks mentioned earlier for multiple-window architectures: 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]. Furthermore, Radin [Radi82, p. 47] has claimed that:

> *...all the registers which the CPU can afford to build in hardware should be directly and simultaneously addressable.*

He expects that an optimizing compiler can make better use of them. At this time there is no clear understanding of the data memory reduction obtained by multiple-window architectures and by

[2] Notice that the CLIPPER processor has been classified as a single-window architecture although their 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.: *pc*, su (*psw*), Ma (process id.), Ap (addr. mapping), Lo (shift amount), Hi (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.: *pc*, *psw*, 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.: *pc*, *psw*, system status	[Sach85,Neff86]
Titan (DEC)	• 63 general-purpose registers	[Wall87]

Table 1.2: Single-Window Processors

the compiler optimizations provided to increase the register usage and to reduce the RSR traffic. As Patterson [Patt85a, p. 15] has said:

> *If compiler technology can reduce the number of LOADs and STOREs to the extent that register windows can, an optimizing compiler will be clearly superior to a multiple register window scheme.*

We expect that our work will provide some quantitative measurements to obtain a conclusion in the above controversy (see Chapter 2).

Traditionally, single-window processors rely on compiler optimizations (such as live-variable analysis and leaf functions) to reduce the register saving and restoring overhead. These are discussed in Section 1.2.3.4. There has not been any work (to the knowledge of the author) on architectural support to reduce this overhead as the one presented in Section 2.1. 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 explicitly perform register saving and restoring, this 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 the same 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. This is because the instruction can pipeline the reading from the

register file and the writing to memory (and vice versa). For instance, this has been done in SOAR[3] [Samp86]. Eight registers are saved in nine cycles by a single instruction. As a contrast, 16 cycles would be required to save them individually.

1.2.3 Register Allocation and Assignment and Conventional RSR Policies

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 allocation and assignment phase (or phases).

The compiler usually divides the selection of variables and their assignment to registers into two different phases. In the first one, the *Register Allocation* phase, the compiler selects the *simple* or *scalar* variables to be allocated to registers. This selection is performed independently of the number of general-purpose registers available. The selected variables correspond to the temporary values generated by the compiler to perform expression evaluation, to the optimizing variables generated to perform several compiler optimizations (loop-invariant removal, common subexpression elimination, etc.), and to the scalar variables defined by the programmer. The variables considered for allocation depend on the number of passes the compiler performs on the program source code as we will discuss in Subsection 1.2.3.1. In the second phase, *Register Assignment*, the compiler assigns the selected variables to physical registers. 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].

In this section we present the problem of allocating programmer-defined scalar variables to registers, we introduce the previous work performed in both areas (register allocation and assignment), we comment on the partition of the register set usually done by the compiler, we discuss 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 (regarding register allocation only) of some compilers for load/store architectures.

1.2.3.1 Register Allocation and the Alias Problem

In this subsection we present 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 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 problem 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

[3] Although SOAR is a multiple-window machine, the RSR instructions provided are applied to the current window, not to the whole register file. Thus, these instructions could also be valid for a single-window architecture.

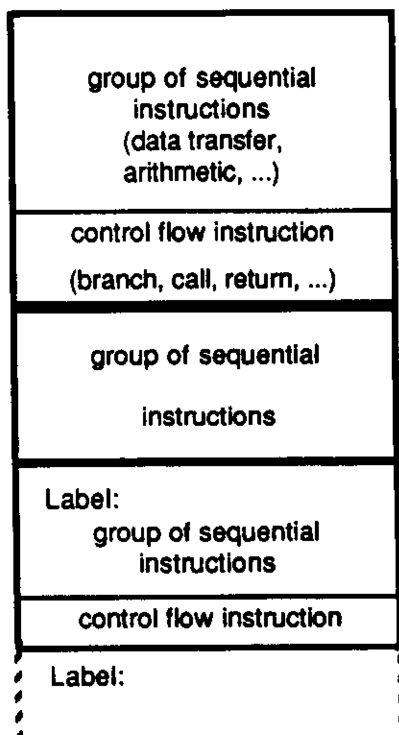


Figure 1.3: Basic Blocks

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 one pass through the program and, therefore, the storage class (memory or register) of a variable has to be decided upon function entry without any knowledge on the variable usage in the function.

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 1.3) [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 programmers specify which local scalar variables they want 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 to allocate 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 to select the variables to be assigned to registers so that the least data memory traffic is generated.

Multi-pass compilers only analyzes 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, they are portable from different source languages to different machines [Perk79,Alle80,Ausl82, Tane83,Powe84].

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[4]. 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 Section 1.2.1, this 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 an 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. For this reason, in Section 1.2.2 we mentioned that at this time there is no clear understanding of the data memory reduction obtained by multiple-window architectures and by compiler optimizations provided to increase the register usage and to reduce the RSR traffic. We expect that our work will provide this understanding.

1.2.3.2 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, as we will see below.

(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].

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.

[4] Notice 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. As a contrast, for multiple-window architectures a register window has to be saved/restored only on overflow/underflow.

For block allocation, the allocator goal is to minimize the number of loads and stores for the variables that 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 they are generated inside the basic block and they 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 this type 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 as it has been mentioned in the previous subsection.

(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 upon 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 (Intraprocedural).* In this case, the set of 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 1.2.3.4.
3. *In the Whole Program (Interprocedural).* To perform interprocedural register allocation the program *call graph* (i.e., the relation of which function calls which) is required. In this case, variables defined in functions that are never active simultaneously can share the same register [Wall86,Stee87]. Also, global scalar variables can be considered for allocation [Wall86]. The register saving and restoring alternatives for interprocedural allocation are discussed in Subsection 1.2.3.5.

Local and global register allocation are not mutually exclusive. Several compilers use both of them [Beat74,Chow83]. The local allocator handles the temporary variables and the global allocator the live variables that have to be preserved across blocks. The reason for this is that when both allocators are available, the global allocator has fewer variables to select for allocation and its implementation is simplified. However, since most of the basic blocks are small[5], most of the register allocation is performed by the global allocator and, therefore, the local allocator can be removed to simplify the optimizer [Laru86].

[5]This is true for most modern languages (C, PASCAL, MODULA-2, LISP, ...) that encourage the use of functions, but not for languages like FORTRAN.

Machine-independent "global" (intraprocedural) optimizers allocate the selected variables to an infinite number of *pseudo-registers*. In a later phase (register assignment) the pseudo-registers are mapped to the physical ones. To facilitate this selection the optimizer also provides for each pseudo-register its priority for allocation and its live range. The priority is used by the assignator to decide which pseudo-registers should be assigned to a physical register when there are not enough registers available. Variables with disjoint live range can share the same register. Algorithms that perform this type of register assignment are called *graph coloring algorithms* [Chai81,Chai82, Chow84,Muns86] since the problem is similar to that of coloring a graph with a limited number of colors (registers).

In this report we do not give a detailed description of the local and global algorithms. The reader is referred to [John73] or [John75, Section 1.3] for a survey on local allocation algorithms, to [Hsu87, Section 3.11.1] for a survey on intraprocedural allocation algorithms, and to [Chow83, Section 1.1] for a survey on machine-independent optimizing compilers. Since the number of registers required by a local allocator is small (see Section 1.2.4) and the traffic generated by several of these allocators is very similar [Hsu87, Section 3.8], we will concentrate our attention on intraprocedural and interprocedural register allocators (which correspond to the current compiler technology).

1.2.3.3 Register Set Partition

When the allocator uses both local and global strategies, the compiler writer has to distribute (to *partition*) the general-purpose registers provided by the architecture between them. This partition determines how registers are saved and restored (Subsection 1.2.3.4) and how allocated variables (i.e., pseudo-registers) are assigned to physical registers (Subsection 1.2.3.5). In this subsection we present the three alternatives that the compiler writer has in partitioning the register set: a unique register set, two independent parts, and two dependent parts.

To determine the number of registers available for the register allocator the compiler writer has to know the architectural characteristics of the register set. For instance, registers for parameter passing in multiple-window architectures, architecture-defined registers for environment registers, even/odd registers for floating point variables, register set partition between registers for data and registers for addresses, etc. To simplify our discussion let us assume a uniform general-purpose register set. We remove from it the registers that have a pre-defined use either defined by the architecture or by the compiler itself. Thus, this discussion applies to the truly "general-purpose" registers available for local and global register allocation. In this case, these registers can be partitioned in three alternative ways:

1. A *unique register set*. This is the alternative taken when only one register allocation approach is used (either local or global).
2. The register set is partitioned in *two independent parts*. One part is for the variables *to be destroyed* and the other for the ones *to be preserved*. In this case, the local allocator uses the to-be-destroyed registers and the global allocator the to-be-preserved ones. Each allocator is responsible for its own set and they do not share registers. This implies that when one allocator runs out of its share of registers, it has to insert load/store instructions even though registers might be available in the other part.
3. The register set is partitioned in *two dependent parts*. Since usually the local allocator runs

before the global allocator, when it uses all the registers in its partition, it allocates the temporary values to pseudo-registers. These registers will compete later for physical registers with the ones generated by the global 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 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 interprocedural allocation is performed. The reason for this is that an intraprocedural 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 since 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 since there are no references to the registers after return [Band87]. In this case, the restoring traffic might get reduced (if registers were not present in the register file).

1.2.3.4 Conventional Register Saving and Restoring Policies

The *conventional* register saving and restoring policies only apply to intraprocedural register allocators for single-window architectures. The reason is that (1) for multiple-window architectures registers are saved/restored during overflow/underflow or during program execution (for dribble-back register files) as has already been discussed in Section 1.2.1, and (2) for interprocedural allocation there are no *conventional* RSR policies because it is a current research topic. Subsection 1.2.3.5 mentions the RSR alternatives available for interprocedural allocators.

There are two conventional register saving/restoring policies for intraprocedural allocation:

1. To save the registers in the caller (before the call) and restore them upon return (after the call). This is called **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. In the former case, a unique partition can be used. In the latter, it is more convenient to have two partitions so that only registers "globally" allocated are preserved.
2. To save the registers in the callee (upon entry) and restore them before (or during) the return. This is called **Policy B**. This policy is usually used with two partitions so that all the to-be-preserved registers defined in the function are saved and restored, but the to-be-destroyed

registers are used without having to save/restore them. If the function is a *leaf function* (i.e., it does not call any other function), then the to-be-destroyed registers can be used to assign variables [Chow86,Cout86b]. In this case, none (or less) to-be-preserved registers have to be saved/restored.

The register saving/restoring policies determine the *threshold* on 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 that the traffic generated by a variable in memory be 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. This is because for Policy B a variable that is assigned to a register generates two memory references every time that the function is called. Thus, if the variable is referred 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. This numbers are more difficult to estimate during compilation time than the simple two references for Policy B.

Although these are the conventional policies used by nowadays compilers, the author has only been able to find one performance evaluation study which compares both of them.

McKusick [McKu84] has evaluated the size and the 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 resulted 8% bigger than with Policy B. Moreover, since the running time of the programs did not experiment much difference, he concluded that Policy B was better.

However, the compiler that he used to perform these measurements, the Graham-Granville compiler [Henr84], is an one-pass compiler and, therefore, only explicitly-defined variables are allocated to registers. Moreover, no live-variable analysis is performed by the compiler so that all the assigned registers in the function have to be saved/restored. As a consequence, his conclusion might be true for a nonoptimizing compiler, but not for an optimizing one with live-variable analysis. In fact, this is what we conclude from our work as presented in Chapters 3 and 4: Policy A generates less RSR traffic than Policy B when live-analysis is performed while Policy B performs better otherwise.

1.2.3.5 Register Assignment

In this phase, the pseudo-registers with the variables that have been selected for allocation have to be assigned (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 1.2.3.3. In this subsection we present two approaches for intraprocedural register assignment, we summarize how three intraprocedural register allocators assign variables to registers, and we discuss two approaches for interprocedural 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:

- Assign only a group of them to registers and assign the rest to memory (based on some static usage frequency).
- Insert an instruction to release a "busy" 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, 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. Thus, this is one of the points that this work will evaluate (see Section 2.3).

The register assignment depends on the optimization degree of the compiler. From all the possible register allocation combinations that have been presented in the Subsection 1.2.3.2, in this subsection we will only comment on three intraprocedural 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 lost 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 Compiler[6] [John78,John79]. The compiler uses two independent partitions. If the programmer has specified more register variables than the number of to-be-preserved registers, only the first defined ones 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 1.2.3.1). Notice that the compiler does not use spill instructions.
2. Intraprocedural 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 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.
3. Intraprocedural 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 (as a contrast with Chaitin's allocator).

We now present two approaches for assigning pseudo-registers allocated by an interprocedural allocator. These are:

[6] Intraprocedural register allocation is performed in this case by the programmers themselves since the compiler only assigns explicitly-defined register variables to registers (see Subsection 1.2.3.1).

1. To assign only the variables that fit in the register partition. This is the approach followed by Wall [Wall86]. In this case, the variables are selected based on the priority of the function where it is defined and 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 intraprocedural register allocation with interprocedural register assignment. This is the approach taken by Steenkiste [Stee87]. 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 its immediate ancestors (for 10 small programs and 3 large ones). However, this might not be true for any programming language or application. In Section 4.2 we present our approach that also combines intraprocedural register allocation with interprocedural 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.

Table 1.3 shows the characteristics (regarding register allocation and assignment) of several compilers for some of the machines given in Tables 1.1 and 1.2. The contents of the table should be self-explanatory, except for the notation $sp \equiv fp \equiv ap$. This 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].

1.2.4 Performance Studies on Register Allocation

No major performance study has been found in the literature that compares the register allocation and assignment 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 global allocators.
- To evaluate the percentage of data memory references that are eliminated when variables are assigned to registers.

The measurements obtained by different studies cannot easily compared. This is 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 global allocators while others only the traffic generated by the latter; etc.) and the size of the programs measured and the programming languages used are also different.

Compiler for	Description	References
RISC I & II	• Portable C Compiler (one-pass compiler; intraprocedural) • register allocation: Sequential by definition alias problem solved by hardware • register set usage: *r0* ≡ 0 *r1–r3*: environment registers (*r1* ≡ *sp*) *r4–r9*: temporary values (to be destroyed) *r10–r15*: incoming arguments & return address *r16–r25*: local regs.; *r16* ≡ *fp* ≡ *ap* *r26–r31*: only used for outgoing args. • 1 window transferred on overflow/underflow	[Miro82] [Tami83]
IBM 801	• register allocation by graph coloring (intraprocedural; Policy A with live information) • parameter passing through registers	[Chai81] [Ausl82] [Chai82]
RIDGE 32	• only integer register variables allocated • register set usage: *r0–r5*: to-be-destroyed registers *r6–r13*: to-be-preserved registers (*r11* ≡ ret. addr.) *r14* ≡ *sp*; *r15* ≡ *fp* ≡ *ap*	[RID83b]
MIPS	• reg. allocation by priority-based coloring (intraprocedural) • Policy A mainly used (B also supported) • parameters passed through registers (up to 4) • leaf-function optimization • *sp* ≡ *fp* ≡ *ap*	[Chow83] [Chow84] [Chow86]
SOAR	• one pass compiler (intraprocedural 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	• reg. allocation based on [Chow83] (intraprocedural) • 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	• intraprocedural register allocation (Policy B) 16 regs. to-be-preserved & 13 regs. to-be-destroyed 3 environment registers (*sp* ≢ *fp*) • parameters passed through registers (up to 4) • leaf-function optimization	[Cout86a] [Cout86b] [Mage87]
CRISP	• stack compression by live/dead analysis • catch instruction optimization • *sp* ≡ *fp* ≡ *ap*	[Band87]
Titan	• interprocedural register assignment 52 registers available no register saving/restoring overhead	[Wall86]

Table 1.3: Compiler Features for Several Processors

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 program 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 to-be-destroyed registers are enough to evaluate 95% of the executed arithmetic expressions and that 3 to-be-destroyed 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 to-be-destroyed registers. In this case, 18% of the (stack) data memory traffic is eliminated.

Therefore, the number of registers required by a local allocator is, on the average, small. For this reason, we decided to ignore local allocation in this work.

Intraprocedural 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 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 (for the programs given above) the number of local scalar variables defined per (executed) function to estimate the number of to-be-preserved registers for intraprocedural allocators: 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, we concluded that our measurements did not show any reason for increasing the register file size beyond 16 registers.

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 are 12 small programs (Towers of Hanoi, Queen, Puzzle, Quicksort, Erastothenes Sieve, ...). 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 for MC68000 uses Policy B. Chow claims that the allocation cost (i.e., the threshold) is lower for Policy B and, therefore, more variables get allocated.

Flynn et al. [Flyn87] 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 intraprocedural allocator does not become smaller for more than 8 to-be-preserved registers. For 2 to-be-preserved 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 global allocator. The programs measured are the Spice LISP interpreter, the LISP compiler for SPUR, and a circuit simulator. As we have indicated in Table 1.3, 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 investigate 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[7].

Interprocedural Allocators

Wall [Wall86] uses 52 to-be-preserved registers for his interprocedural allocator. He has measured six programs: four small ones (Livermore loops, Whetstone, Linpack, and the Stanford benchmark suite) and two medium-size 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 2.6), we concentrate our attention in 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. To increase the number of memory references eliminated, Wall performs intraprocedural coloring[8], execution profiling (to know the 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 are 83% (coloring), 92% (profiling), and 95% (both) for the simulator and 61%, 78%, and 83% for the verifier, respectively.

Steenkiste [Stee87] has implemented his interprocedural register allocator on MIPS-X for LISP. He has measured 10 small LISP programs. When only intraprocedural register allocation is performed, 55% of the stack references are eliminated (including RSR traffic). Wen interprocedural register assignment is made, 49% of the remaining stack references are eliminated. Therefore, 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 assume that the traffic reduction would be smaller for larger programs.

Flynn et al. [Flyn87] have also measured the number of register required by the Steenkiste's interprocedural register allocator (for the five PASCAL programs given above). The compiler was able to eliminate 47% of the data memory traffic for local scalar variables (including RSR traffic, but not the traffic caused by the local allocator) when 8 registers were available to the interprocedural

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

[8]He calls it "local" coloring (sic).

allocator, 55% when 16, 60% when 32, and 62% when 64. There was no data memory traffic reduction for more than 64 to-be-preserved registers.

In spite of the above, modern processors are constantly increasing their general-purpose register sets: 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. The last processor is the only one that was designed to be used with an interprocedural register allocator [Wall86] to use efficiently its 63 general-purpose registers. As we will discuss in Chapter 2, we expect to show that for the programs measured (given in Section 2.6) a register set larger than 16 needs the dynamic architectural policy proposed and/or the interprocedural compiler optimizations to use the register set efficiently.

1.2.5 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 a processor that is being designed, called the *Proposed Machine* (PM), a model that defines the processor characteristics must be used. Traditionally, this model is defined 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 report 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 the 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 is 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 2.6).

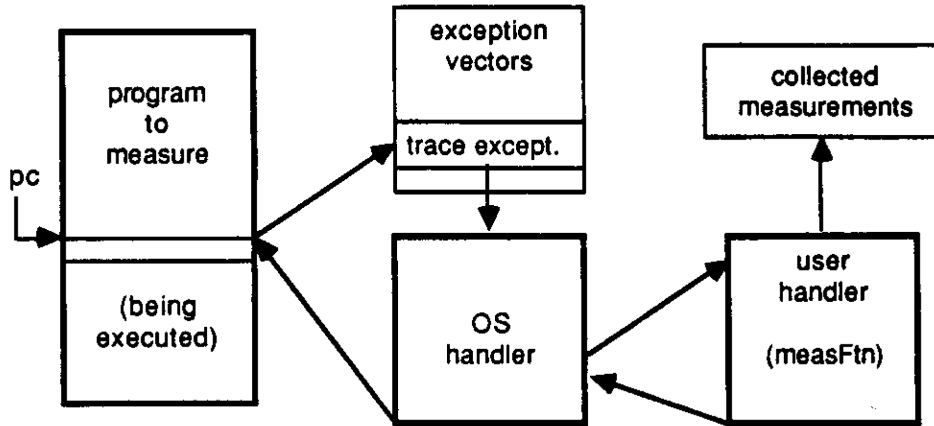


Figure 1.4: Gathering Measurements with the Step-by-Step Instruction Tracer

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 to partially decode and interpret each instruction to extract the measurements that 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. As an 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 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 1.4 shows this second implementation. The execution time for counting the number of machine instructions executed[9] for the UNIX SORT program is 1,540 times its normal execution time when **ptrace** is used[10] 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 that one might 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

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

[10]Note 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.

for collecting architectural measurements: the Block-and-Actions Generator (BKGEN). This is presented in Section 2.5. In the remaining of this section, we comment on 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 per 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 problem).

To verify the code generated by the MIPS compiler (before the hardware was available) an *objet-to-object translator* from MIPS (i.e., the PM) to VAX-11 (i.e., 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 to 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 is 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 since the program behavior is the same for any machine. They have measured the execution time of a small program (Quicksort) running in MicroVAX-II (2 sec.), with a conventional simulator (15 minutes), and with their new assembly-to-assembly translator (6 sec.). Therefore, the overhead introduced has been reduced 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 individual 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 to 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 with respect to the previous two approaches is that the translator has information with respect the whole code block so that a better usage of the resources (i.e., EM registers) is made. This is because the previous approaches translates each PM instruction individually. Thus, MIMIC can perform global register allocation rather than the local allocation used by the previous two approaches. May concludes 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).

1.3 Related Work

There are two topics that are closely related to registers, but that are not discussed in detail in this work. 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.

1.3.1 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 to perform expression evaluation, and to store optimizing variables (for loop-invariant expression, for common subexpressions, ...).

Although there is a general consensus on having a cache for the first two streams, there is some debate about the best alternative for the third one. 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 prefer to 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 Section 1.2.3.1).
4. Registers increase the complexity of the compiler (due to 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 of the size of the Motorola compiler [Wirt86].

Although both cache memory and registers are valid methods to get fast access to operands, the advantages of having registers are:

1. Cache memories cause some overhead in checking 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 require more area on the chip than registers (for the same number of entries). This is because space for tags has to be provided for a cache.
3. 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.
4. 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 storing the virtual address of the operands (once computed) in the instruction cache [Nort83,Ditz87c].
5. It has been shown that the reference pattern for local operands is different than the one generated by global references [Hsu87]. Thus, 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.
6. 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 of having registers outbalance its drawbacks and that the compiler complexity for a register-oriented machine is not a significant argument. This is because:

- The complexity of the compiler is reduced by having machine-independent register allocators [John75,Leve83,Chow83].
- With the present trend towards providing cache control instructions to the compiler to reduce unnecessary data memory traffic (for dead data) [Radi82,Birn85] or towards performing live-variable analysis and static frequency usage to increase the cache hit ratio [Band87], the cache memory is not any longer hidden to 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 the 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.

1.3.2 Instruction Format (RISC versus CISC)

There has also been a long discussion among computer architects about what is the most appropriate instruction format: 0-address, 2-address, register-to-memory, memory-to-memory, ... [Myer77, Schu77,Keed78b,Myer78,Keed78a,Site78,Keed79,Myer82,Keed83]. At the beginning of this decade, several processors (RISC I & II, IBM 801, MIPS, ...) were introduced with a new architectural

design style, called *Reduced Instruction Set Computers* (RISC) as a contrast to the tendency of increasing the complexity of the processors (which are called CISCs, *Complex Instruction Set Computers*).

There is some debate on the elements that define a Reduced-Instruction-Set-Computer approach [Hopk84,Patt85a,Wall85,Taba86,Gima87]. Patterson [Patt85a] has characterized the existing RISCs by the following factors: 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 basic idea is to execute fast the small number of instructions that have been shown to be very frequent [Alex75,Shus78,Swee82, McDa82,Wiec82,Clar82]. As a consequence, the previous controversy on instruction format is nowadays centered around RISC and CISC architectures and in the direction that computer architects should go 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 de facto standard for modern processors. It seems that the number of registers was mainly decided on the basis of 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 registers.

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 de facto standard for nowadays RISC processors. For instance, MIPS-X has eliminated the 2-address instructions that its predecessor (MIPS) had.

However, some authors (mainly pro-CISC designers) are claiming that the addition of more instruction formats reduces the program size [Davi87,Flyn87]. The advantages of a smaller program size are that less space is required to store the program (in disk), less pages in memory are necessary to keep the program working set, and mainly the instruction buffer size is reduced. 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 for 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 is still an open forum for discussion. 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 2

Research Proposal and Expected Contributions

This work investigates both the support that the architecture can provide to implement efficient function calls and the support that can be provided by the compiler. Our basic and main assumption is that registers are provided in the processor. Our goal is to show how these registers can be used more efficiently to reduce the program data memory traffic. We expect that our work will provide to both the computer architect and to the compiler writer the following:

- The design of several new architectural policies to reduce the register saving and restoring (RSR) 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 and they reduce significantly the RSR overhead with respect to the conventional *static* RSR policies: to save/restore registers at the caller and to save/restore them at the callee (see Section 1.2.3.4).
- The evaluation of the data memory traffic reduction provided by these policies and a comparison with the conventional static policies for single-window architectures and the existing schemes for multiple-window architectures: fixed-size windows [Kate83], variable-size windows [Ditz82], and multi-size windows [Hugu85a] (see Section 1.2.1).
- The design of six new compiler optimizations to reduce the RSR overhead. Two of these optimizations are intraprocedural and based on live-variable analysis [Aho86,Hech77]. The other four are interprocedural; their goal is to find a *register assignment* for the variables selected for allocation (in a previous phase) so that the unnecessary RSR traffic can be eliminated.
- The evaluation of these new optimizations and a comparison to some of the already-existing compiler optimizations to reduce the RSR traffic: leaf functions (as used in the compilers for MIPS [Chow86] and for HP-Spectrum [Cout86b]) and live-variable analysis [Aho86,Hech77].
- The comparison of the data memory traffic reduction obtained by several *register allocation* approaches, such as allocating only explicitly-defined register variables [John79], allocating in order of definition, by priority [Frei74], and coloring [Chai82,Chow84], when they are used in conjunction with the static and dynamic policies and with the compiler optimizations.

- The implementation of one of the dynamic architectural policies in a RISC II-like processor and its evaluation. This evaluation takes into account the hardware required for its implementation, the influence of the operations to be performed on the processor cycle time, and the number of cycles required to perform these operations.

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 [Tami81,Rose84]—the tool traditionally used to obtain this type of measurements [Svob76,Ferr78]. BKGEN is presented in Section 2.5.

Since the simulator overhead has been reduced drastically, we have been able to measure large typical programs for a UNIX environment: a C compiler, an assembler, a word processor, and a sort program. The averages presented here and in Chapters 3 and 4 correspond to these programs. We usually comment on the averages and pinpoint the anomalies for the programs that behave differently. Since these programs do not use any floating point scalar variables, we also perform some measurements using SPICE—a VLSI circuit simulator. However, these SPICE measurements will not be discussed in this proposal because they are not available yet. Section 2.6 presents the programs measured and discusses why we have selected these programs rather than *small specific programs* (such as the Ackermann's function, the Hanoi Towers, ...) or *synthetic benchmarks* (such as Whetstone, Dhrystone, ...).

The following sections present the conventional and the proposed register saving and restoring architectural policies (Section 2.1), the existent and proposed compiler optimizations to reduce the RSR traffic (Section 2.2), the register allocation research that we expect to perform (Section 2.3), and the expected implementation of one of the dynamic policies (Section 2.4). These sections also indicate the expected conclusions that this work will provide. A summary with the current status of this research and the work that still remains to be done are discussed in Section 2.7.

2.1 Architectural Support to Reduce the RSR Traffic

The two conventional register saving/restoring policies used nowadays in single-window architectures are: to save/restore registers at the caller (**Policy A**) and to save/restore them at the callee (**Policy B**). These policies 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. We propose six new[1] architectural policies that make use of *dynamic* information to reduce the RSR traffic. We compare the architectural support required by each dynamic policy and select one (**Policy G**) as the best candidate for implementation and future comparisons. A detailed description of these policies and their evaluation is presented in Chapter 3. In this section we discuss how the RSR traffic has been evaluated and summarize the most important conclusions obtained.

To be able to evaluate the RSR traffic reduction obtained for each policy it is necessary to use a specific register allocation and assignment performed by the compiler. The alternative of presenting each policy with each possible compiler optimization and each possible register allocation scheme

[1] Actually, one of the policies has already been implemented in software as we will mention in Section 3.2. However, no hardware implementation has been made to the knowledge of the author.

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 [John79]. This scheme has already been used to evaluate several register saving/restoring schemes by other authors [Patt82b,Kate83,Eick87,Hsu87] although it is not the optimal for an optimizing compiler. It is not the optimal because the overall traffic can be reduced with a better selection of which local simple variables should be allocated to registers (see Section 1.2.3). However, since for the moment we just consider the RSR traffic, not the overall traffic, this scheme is useful to evaluate the RSR traffic reduction caused by each policy because it implies a heavy register usage. Moreover, upon completion of this work we will verify that the conclusions obtained hold for others register allocation approaches (by priority, coloring, ...).

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

1. The dynamic Policy G reduces the register saving and restoring traffic with respect to the conventional static policies used by most processors. Policy G generates between 12% (for 24 to-be-preserved registers) and 31% (for 6) of the RSR traffic produced 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 to-be-preserved 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 two situations, the RSR traffic generated for the Policies A and B is 176% and 124%, respectively. Consequently, the overall data memory traffic might not be reduced for a larger register set when the conventional static RSR policies are used.

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 whole set of to-be-destroyed registers and the to-be-preserved registers that are currently used (i.e., that have been written), and to restore the to-be-destroyed registers. The to-be-preserved 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 (see Section 3.3).

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 section, this is not the case when live-variable analysis is performed.

A comparison of the RSR traffic caused by Policy G with the one caused by the already-existing schemes for multiple-window architectures is not offered in this proposal, but it will be available upon completion of this work.

2.2 Compiler Support to Reduce the RSR Traffic

Several compiler optimizations to reduce the RSR traffic are discussed for the Policies A, B, and G. A detailed description of these optimizations and their evaluation is presented in Chapter 4. Here we describe them briefly and summarize the most important conclusions obtained. The register allocation and assignment approach used to perform the measurements presented in this section and in Chapter 4 is the one described in the previous section.

The compiler optimizations are subdivided into two categories depending on their scope: intraprocedural optimizations and interprocedural optimizations. For the former, two existing optimizations are discussed and evaluated: live-variable analysis for Policy A (renamed **A-live**) and register assignment in leaf functions for Policy B (renamed **B-lf**) and for Policy G (renamed **G-lf**). As expected, the optimized policies generate less traffic than the unoptimized ones:

- Policy A-live generates from 38% (for 32 registers) to 53% (for 6) of the RSR traffic produced by Policy A.
- Policy B-lf generates from 84% (for 6) to 88% (for 32) of Policy B.
- Policy G-lf generates from 77% (for 6) to 97% (for 16) of Policy G.

Moreover, two new optimizations—based on live-variable analysis—are also presented. One is for Policy A (renamed **A-lvOpt**) and the other for Policy G (renamed **G-live**), although it can be used for any other dynamic policy.

Policy A-live saves and restores all the registers that contain live variables at each call instruction. Our optimization, A-lvOpt, saves only the registers with live variables that have been written since the last time 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 traffic of saving a register whose contents 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 is discussed in detail in Section 4.1.2. Our measurements have shown that Policy A-lvOpt generates between 59% (for 32 registers) and 64% (for 6) of the RSR traffic produced by the standard A-live.

Policy G-live eliminates the saving traffic for the registers that have been used previously, but whose contents is *dead*, i.e., for the registers such that the first operation to be performed on them after the return is a write. When the compiler is generating code for a function, it does not know which variables are live at its possible callers (remember that these are intraprocedural optimizations). Thus, it must be the caller that indicates which registers are dead before a call so that the callee does not need to save them. This implies that to generate less RSR traffic, more instruction memory traffic has to be generated. For this reason, we are not very positive about the expected gains to be obtained with this optimization as is discussed in detail in Section 4.1.4. No measurements are available at the moment of writing this proposal for Policy G-live (although they should be upon completion of this work).

When we compare the optimized policies (except Policy G-live), we can 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% (for 16 registers) and 95% (for 6) of Policy B-lf traffic and Policy A-lvOpt between 53% (for 16) and 61% (for 6). However, as we will discuss in Section 4.1.1, this is not true for every measured program.
2. The optimized dynamic Policy G, G-lf, generates less RSR traffic than the best static one. Policy G-lf generates between 47% (for 6 registers) and 22% (for 24) of the RSR traffic produced by Policy A-lvOpt.
3. 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 mentioned above when no optimizations were performed.

To be able to reduce the RSR traffic even further four new interprocedural optimizations are proposed. These optimizations perform the register assignment for the variables selected by an intraprocedural register allocator. Since at this phase the whole *call graph* is known, registers can be assigned is such a way that the RSR traffic is reduced. For instance, for Policy A 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. The optimizer looks for a *disjoint register assignment* for the registers required by this function and its descendants so that the maximum RSR traffic can be eliminated. 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**)—as we will see in Section 4.2. For this reason, we said *four new optimizations* rather than *one new optimization with four variations*.

To find a register assignment such that the least memory traffic is generated during program execution we obtain an *execution profile* for one or a few program executions. This profile provides the number of times that each function is executed for a specific input data. This is used to determine the functions that generate the most RSR traffic and, therefore, to detect the best candidates for a disjoint register assignment. We expect to show that this information (although for one or a few program executions) is much more accurate that the static information that can be collected during compilation time and sufficient to be valid for any program execution.

No measurements are available at the moment of writing this proposal for the interprocedural optimizations. We expect that Policy G^g-lf will generate less traffic than the static ones, as has always been the case for the previous measurements. However, Policy G will not be any more the only one that might benefit from a larger register set. We expect that for larger register sets, the interprocedural optimized static policies will get a reduction in their RSR traffic. This was not the case without interprocedural optimization as we have already mentioned. Moreover, we also expect to show that the benefits that we can obtain from the interprocedural optimized static policies are closely related to the program structure while this might not be the case for Policy G^g-lf. For instance, if the main portion of RSR traffic is generated for a set of recursive functions, the static Policies A^g-live, A^g-lvOpt, and B^g-lf cannot reduce this traffic, but the dynamic Policy G^g-lf might.

Finally, we also expect to show that to be able to use efficiently a large register set (more than

16 registers), the system has to provide some architectural support (i.e., Policy G for single-window architectures) and/or the interprocedural optimizations discussed above.

2.3 Register Allocation and Data Memory Traffic

Since the RSR traffic has always been considered the main drawback to allocate variables to registers, we want to evaluate several register allocation schemes and compare the data memory traffic reduction obtained by these schemes when they are used in conjunction with the static and dynamic policies and with the compiler optimizations to reduce the RSR traffic. To perform register allocation the following factors have to be taken into account[2]:

1. The *class* of scalar variables considered for allocation:
   - Globals (external and static).
   - Locals (automatic and register).
   - Temporary variables for expression evaluation. These variables are usually allocated to to-be-destroyed registers since their value is not usually preserved across functions.
   - Compiler-defined variables for values used by several compiler optimizations (common expression elimination, loop-invariant removal, ...). These variables might have to be preserved across function calls.
2. The *type* of scalar variables considered for allocation:
   - Integers and pointers.
   - Characters and short integers.
   - Floating point variables (floats and doubles).
3. The *scope* of the register allocation (see Section 1.2.3.2):
   - Intra-block.
   - Intraprocedural.
   - Interprocedural.
4. The *selection* mechanism to decide which scalar variables should reside in registers:
   - Allocate only explicitly-defined register variables. This scheme is used by one-pass compilers.
   - Allocate according to the order of variable definition. This is the scheme used for one-pass compilers when hardware support is provided to solve the alias problem (see Section 1.2.3.1). This has also been used to obtain the measurements in Chapters 3 and 4.
   - Static linear priority.
   - Static weighted priority (additive or multiplicative).
5. The *threshold* to decide when a variable is allocated to memory rather to a register even though there are registers available (see Section 1.2.3.4).

[2] Although this discussion is centered on C variables[Kern78], it can be generalized to any block-structured programming language (ADA [Hill83], Pascal, Modula-2, ...)

6. The *sharing* of registers among variables:
   - One register has only one variable assigned during the whole function (1:1).
   - The same register is shared among n variables during function execution (1:n). This is possible because the n variables have disjoint live intervals.

A discussion on each factor is out of the scope of this proposal and is left to be done in the final document.

Our goals are:

1. To verify that the conclusions that we have obtained for a specific register allocation scheme (i.e., sequential per variable definition) are still valid for the others (although the exact measurement results might differ somewhat).
2. To evaluate the data memory traffic reduction when our new architectural policies and optimization approaches are used. Also, to characterize the influence of the above mentioned factors on the total data memory traffic.
3. To offer an evaluation of the data memory traffic generated by three C Compilers (PCC [John79], ACK [Tane83,ACK85], and GNU [Stal88]) for VAX-11.

Although no new register allocation scheme is expected to be proposed, we want to study the possibility of using dynamic information to perform register allocation. The possibility of using dynamic information to perform some compiler optimizations has already been mentioned [Fish84,Elli86] and Wall [Wall86] has already used it for its interprocedural register allocator. However, no study has been made for intraprocedural register allocators to the knowledge of the author.

Finally, we also expect that our work will provide a quantitative analysis of the data memory traffic reduction provided by both the compiler support and the architectural support presented in this report. This will let us evaluate the benefits and the limitations of multiple-window architectures versus single-window architectures when an optimizing compiler is used.

2.4 Implementation of Policy G

It could be the case that although the total number of data memory references becomes smaller with Policy G, 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. Thus, it is important to consider the implementation of Policy G to show that it has no negative effects. We will present a possible implementation of Policy G for a RISC II-like processor. Our goals are:

1. To show that the implementation of Policy G does not affect the processor cycle because the operations required are performed in parallel with the main CPU activities. This is important because the processor cycle depends on to the register file size as discussed in Section 1.2.1.

Thus, multiple-window architectures have a larger processor cycle because of the register-file size. This is not the case for single-window architectures with Policy G.

2. To estimate the hardware required in both the data section and the control section for its implementation. This is necessary to consider the feasibility of a VLSI implementation.

3. To evaluate the number of processor cycles required for the Policy G saving and restoring operations and to compare them with the number of processor cycles required for the conventional saving and restoring operations.

4. To evaluate the speed-up of a RISC II-like processor with dynamic Policy G^s-lf with respect to the static compiler optimizations A^s-live (or A^s-lvOpt) and B^s-lf.

The reason for selecting a RISC II-like processor to study the implementation for Policy G is that the documentation on the RISC II implementation is the most publicly available [Kate83] of the modern VLSI processors.

To be able to obtain the above evaluations we will modify the RISC II processor such that it will only have a single window register file and it will have two additional instructions to perform the register saving and restoring:

savRegs *mask*: The *mask* indicates the registers that

- must be saved before a call instruction for Policy A^s-live or after function entry for Policy B^s-lf.
- might be saved after function entry for Policy G^s-lf if they have been used by any of the callers and have not been yet saved.

restRegs *mask*: This instruction indicates the registers that must be restored after a call instruction for Policy A^s-live or before a return instruction for Policy B^s-lf.

This instruction is not needed by Policy G^s-lf since the restoring is performed during a register-to-register instruction (if necessary; see Section 3.3).

Although the existent *call* and *return* RISC II instructions should also be modified from their current implementation for a multiple-window architecture to our single-window architecture, we do not need to do so. The reason is that these instructions have to be used by both the static and the dynamic policies. Thus, their implementation have the same influence in the processor speed and we will ignore them.

A possible implementation for the data section has already been sketched [Lang86]. However, this is not included in this proposal. The reason is that we prefer to have the implementation more polished before discussing it. A "paper" design will be done to estimate the number of cycles required to execute these instructions, the delay introduced in the pipeline to restore a register for Policy G^s-lf, and the area required for its implementation. However, no "physical" implementation is expected to be performed for this work.

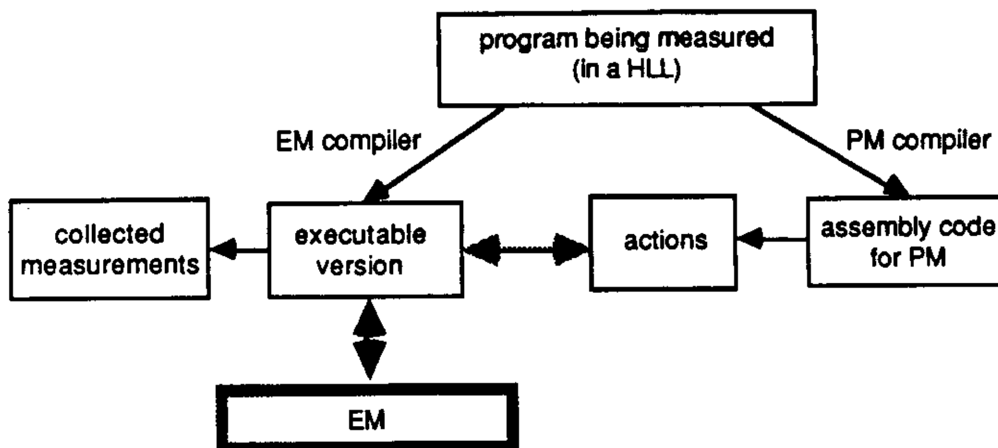


Figure 2.1: Gathering Measurements for the PM on the EM

2.5 BKGEN as a Tool for Collecting Dynamic Measurements

As we said in Section 1.2.5, 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. 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* so that when the program is executed, it collects the measurements associated with the PM (see Figure 2.1).

To present how BKGEN is implemented we explain first how to associate EM blocks to PM blocks and actions and afterwards, 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

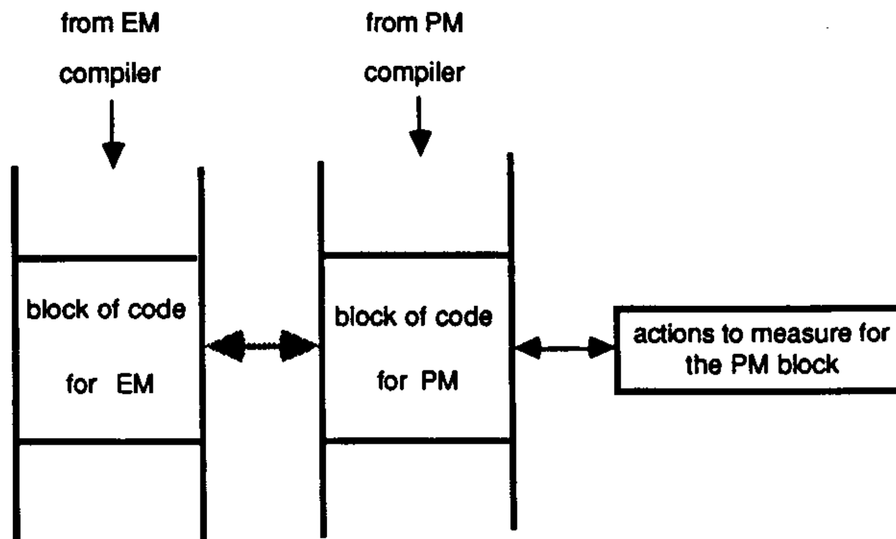


Figure 2.2: One-to-One Block Mapping

was identical, we would say that we have a *valid mapping* (see Figure 2.2). In this case, it is possible to directly execute machine language instructions of the EM 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 2.3). 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 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 Section 1.2.5).

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 simple variables, traffic caused by register saving and restoring with static policies, etc.

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, etc.

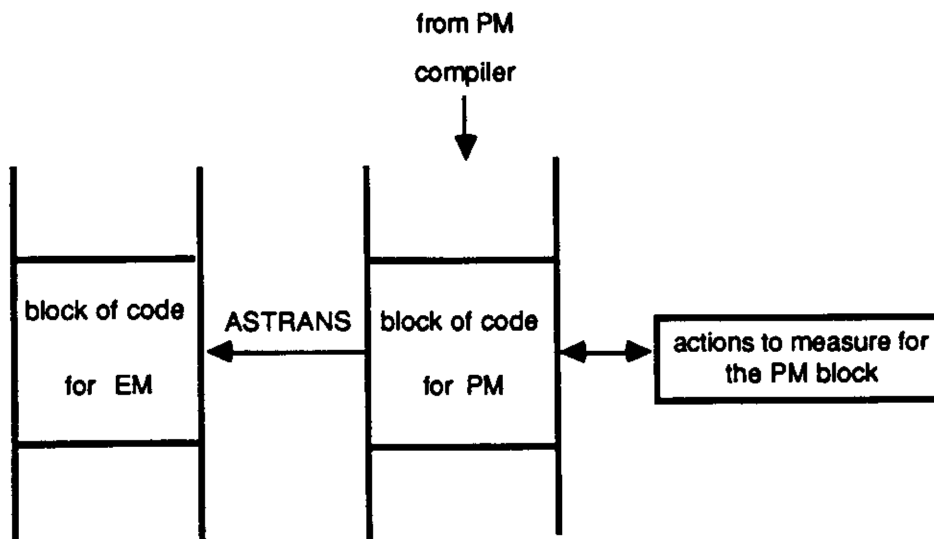


Figure 2.3: PM to EM Block Translation

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, etc.

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 to the PM instruction) have been executed. Since our measurements are of Types I and II and the complexity of BKGEN for measurements of Type III is greater than the one for the former types, we concentrate our attention on BKGEN for measurements of Types I and II. The reader is referred once more to [Hugu87] to find more information on BKGEN for measurements of Type III.

For measurements of Types I and II, the block corresponds to a basic block (as defined in Section 1.2.3). 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 they 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. This is because it is only necessary to know how many times each block has been executed (since 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 BKGEN has shown that it is more efficient to perform the measurements *in-line* due to the

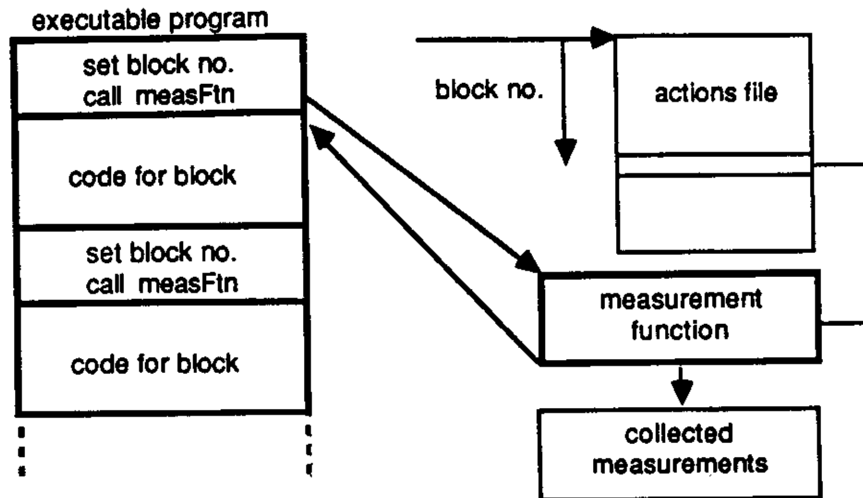


Figure 2.4: Gathering Type II Measurements with BKGEN

large overhead for writing a file of several Mbytes and the amount of disk space required for it; this case is shown in Figure 2.4.

To estimate the overhead introduced by the different alternatives we have counted the number of executed instructions in the SORT program[3]. As we said in Section 1.2.5, 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.

The interception code must not change the state of the EM (i.e., the registers, the condition codes, ...) to have a correct and successful execution of the measured program. 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 larger than the effort to develop a simulator, an emulator, or an instruction tracer. Since BKGEN directly executes machine code for the machine on which the

[3]Notice that this is a Type I measurement. The reason for performing this measurement as Type II is only to estimate the overhead introduced.

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 this type of measurements the designer can obtain meaningful dynamic characteristics of typical programs.

2.6 Programs Measured

In this section we first review some of the *standard benchmarks* that have been used to perform architectural evaluations and comment 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 Fibonnacci numbers, the Erastothenes Sieve [Gilb81], and the puzzle program [Beel84], or *synthetic programs* that reflect some high-level language characteristics or measure some architecture characteristics, such as the CFA benchmarks [Full77], Whetstone [Curn76], and Dhrystone [Weic84]. The first type of programs 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 a real system behavior [Hugu85a,Lang85]. For instance, the RSR traffic caused by Policy G for some of these benchmarks either is zero or has little variation from the one 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,Wong88] have also pointed to the limitations of these benchmarks to evaluate certain architecture features.

Table 2.1 shows some large programs used by some other studies. Small programs are enclosed in parenthesis. 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 (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 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 176 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.

Reference	Description
[Ditz82]	• First Pass of the Portable C Compiler • UNIX word processor • Second Pass of the PCC (generating PDP-11 code) • VLSI design ruler checker
[Blom83]	• UNIX Portable C Compiler • (Puzzle & TAK—heavy recursive function)
[Tami83]	• UNIX Portable C Compiler • (Puzzle & Towers of Hanoi)
[Laru86]	• LISP—Spice Lisp system from CMU • SLC—SPUR Lisp Compiler (based on Spice Lisp) • RSIM—circuit simulator
[Band87]	• UNIX system programs: cat, comm, diff, echo, mv, nroff, pr, rm, & wc
[Cort87a] [Cort88]	• Portable C Compiler for RISC • NROFF—UNIX word processor • YACC—parser generator • LEX—lexical analyzer
[Eick87]	• 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]	• 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 2.1: Large Programs Measured by Some Other Studies

SORT The UNIX sort program, sorting a file with 2250 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.

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.

The averages presented above and in Chapters 3 and 4 correspond to the first four programs (ASM, NROFF, SORT, VPCC). 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. Thus, we prefer to isolate this program from the rest and to use it to perform some specific measurements (for instance, allocation of floating point scalar variables to registers) as we will discuss in the final version of this document. In some parts of this document we use averages from *our previous measurements*. These are usually reported in [Hugu85a] and correspond to only three of the programs (NROFF, SORT, and VPCC).

These measurements do not include the library functions that these programs are using. However, we expect that the conclusions obtained would be the same if the library functions were included.

2.7 Current Status and Future Work

In summary, the current status of this research is the following:

1. The architectural policies for single-window architectures have been developed and its performance measured. Moreover, the implementation of Policy G has already been sketched, but it has not been included in this proposal.
2. The two new intraprocedural optimizations and the four new interprocedural optimizations have been designed. The performance of the two existing intraprocedural optimizations (A-live and B-lf and G-lf) and of one of the new ones (A-lvOpt) have also been evaluated.
3. BKGEN has been designed to perform the above measurements. It has proven to be not only a flexible tool to measure several architectural configurations, but also to measure some compiler optimizations.

The following remains to be done to complete this work:

1. The evaluation of several multiple-window architecture schemes (fixed-size, variable-size, and multi-size windows) and their comparison to the Policies A, B, and G for single-window architectures.
2. The implementation and evaluation of one of the intraprocedural optimizations and the four interprocedural optimizations.
3. The implementation and evaluation of the some of the remaining register allocation approaches (besides the one used to obtain the above measurements). We also have to verify that our conclusions obtained from this register allocation approach hold for the others.
4. The evaluation of the data memory traffic generated by three C Compilers: PCC, ACK, and GNU for VAX-11.
5. A comparison of the measurements obtained for SPICE with the ones obtained by the other four programs.
6. The evaluation of the implementation of Policy G once the design has been completed.

Chapter 3

Architectural Support for Register Saving/Restoring for Single-Window Register Files

When an intraprocedural register allocator is used, the two conventional register saving/restoring (RSR) policies used in single-window architectures are: to save/restore registers at the caller (**Policy A**) and to save/restore them at the callee (**Policy B**). These policies were introduced in Section 1.2.3.4. 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 comparisons.

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 discarted 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 the optimal for an optimizing compiler. It is not the optimal because the overall traffic can be reduced with a better selection of which local scalar variables should be allocated to registers (see Section 1.2.3). However, since for the moment we just consider the RSR traffic and not the overall traffic, this scheme is useful to evaluate the RSR traffic reduction caused by each policy because it implies a heavy register usage. Moreover, upon completion of this work we will verify that the conclusions obtained hold for others register allocation approaches (by priority, coloring, ...; see Section 2.3).

The PCC standard intraprocedural register allocation only allocates to registers integer and pointer *register variables* [Kern78] explicitly defined by the programmer (see Section 1.2.3.1). As a consequence, 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 registers*. This number only increases up to 1.98 when the number of

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

to-be-preserved registers is 32. In this case, the traffic for the local scalars that 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 all local scalar variables are allocated to registers[1]. The scalar variables considered for allocation not only includes integer and pointer variables (as PCC does), but also characters and short integers. As we mentioned in Section 2.6, 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 to-be-preserved registers available.

Moreover, the PCC intraprocedural 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 possible, registers are assigned in a *round-robin fashion*. This implies that consecutively defined functions use different registers if the number required is smaller than the total number of to-be-preserved registers. This does not guarantee to have disjoint registers for a caller and a callee, but it increases the probability for doing so. In Section 4.2, we propose a compiler 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

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

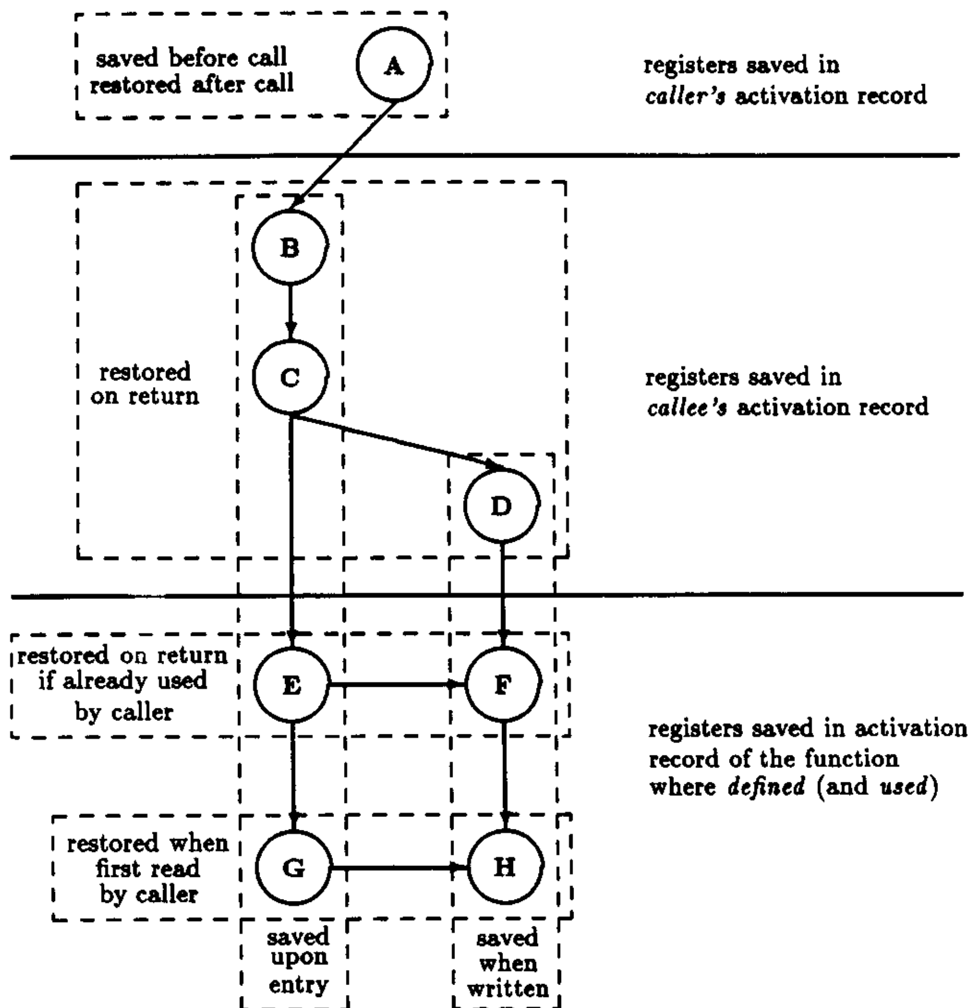


Figure 3.1: Dependencies among the Register Saving/Restoring Policies

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 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 dependencies 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 Section 1.2.3.4). Thus, the callee always receives a *clean* register set.

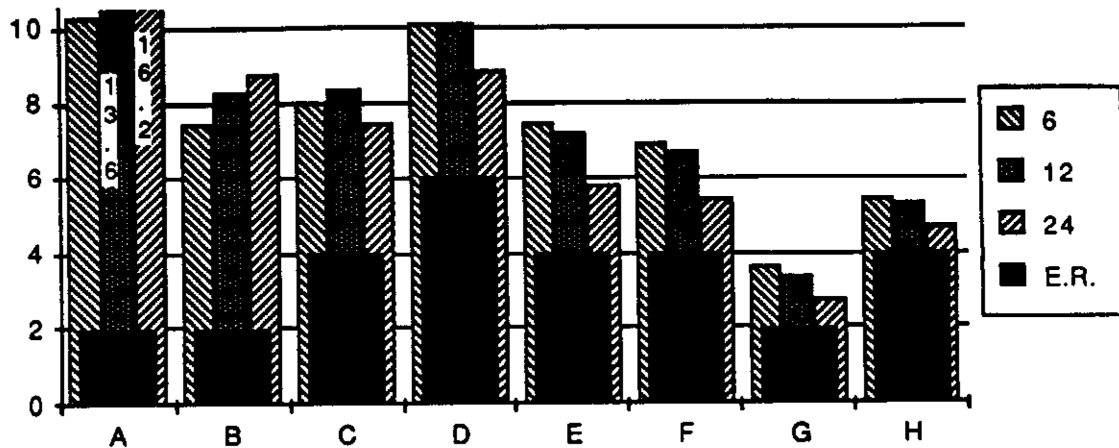


Figure 3.2: RSR Traffic per Function Call with Fixed Cost

2. Policies B, C, and D save the registers in the *callee's activation record.* In this case, 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. 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, upon 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 to-be-preserved 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 parenthesis. 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.

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, ... is identical to all the policies (for a given register set configuration). Thus, the RSR traffic allows us to measure the data memory 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 *program counter* (i.e., the return address)

no. regs.	program	A	B	C	D	E	F	G	H
		Policy							
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 (10.12)	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	0.61	0.45	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 (4.73)	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

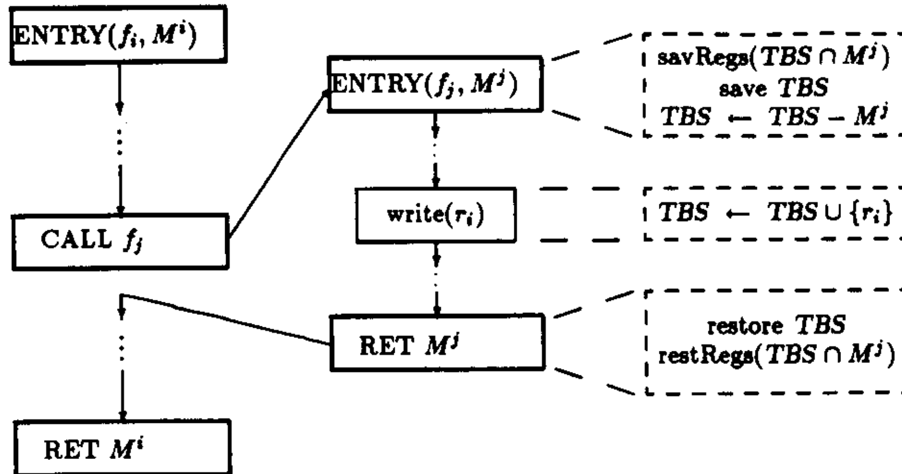


Figure 3.5: Policy C

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

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* ≡ 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 register is written.

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 3.5). Some authors have proposed to implement this RSR policy directly by the compiler without any architectural support [Stee80,Cohe88]. The algorithms to describe the dynamic policies (like the one given in Figure 3.5) 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.

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 3.2).

As we mentioned above, this policy (and the other dynamic policies) would perform better if disjoint registers are assigned to the caller/callee functions. The smaller the intersection is, the fewer registers that have to be saved/restored. For this reason, when the number of to-be-preserved registers increases, the RSR traffic may decrease although the number of locals assigned to registers also increases, Our measurements show a RSR traffic reduction of 14% when the number of to-be-preserved 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 (as we can see in Figure 3.2). Notice that there is a 2% RSR traffic increase from 24 registers to 32. This is a consequence of the round-robin assignment. We expect that the compiler optimization proposed in Section 4.2 will prevent 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 to-be-preserved registers and between 52% and 58% when there are 16–32. Thus, a larger register set gives us a larger traffic reduction with respect Policy B.

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 upon 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. Both masks have to be saved/restored. Although two dynamic masks have to be saved/restored, this does not imply that four memory references are always generated per function (as Figure 3.2 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-to-be-preserved-register set configuration). In this case, only two memory references would be generated. (We decided to show the most general case in Figure 3.2 to enphasize the fact that both masks have to be saved/restored.)

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 we check both masks: 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 3.6).

Since in this case there is no static mask, it is necessary to have some other way of differentiating between to-be-preserved registers and to-be-destroyed registers. Two alternatives exist for this: 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 93%–95% of the RSR traffic produced by Policy C when 8–16 to-be-preserved registers are available and 82%–83% when 24 and 32 registers are available. The performance for 6 registers has been a surprise since Policy D gives

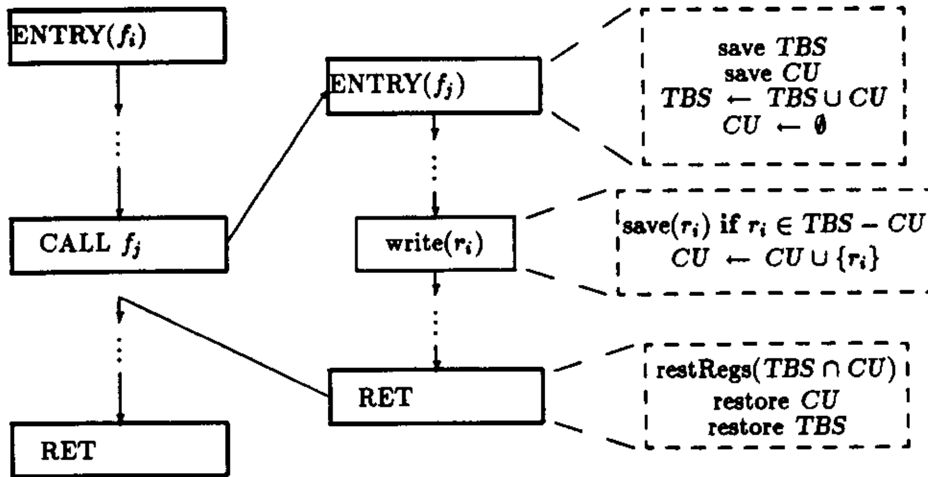


Figure 3.6: Policy D

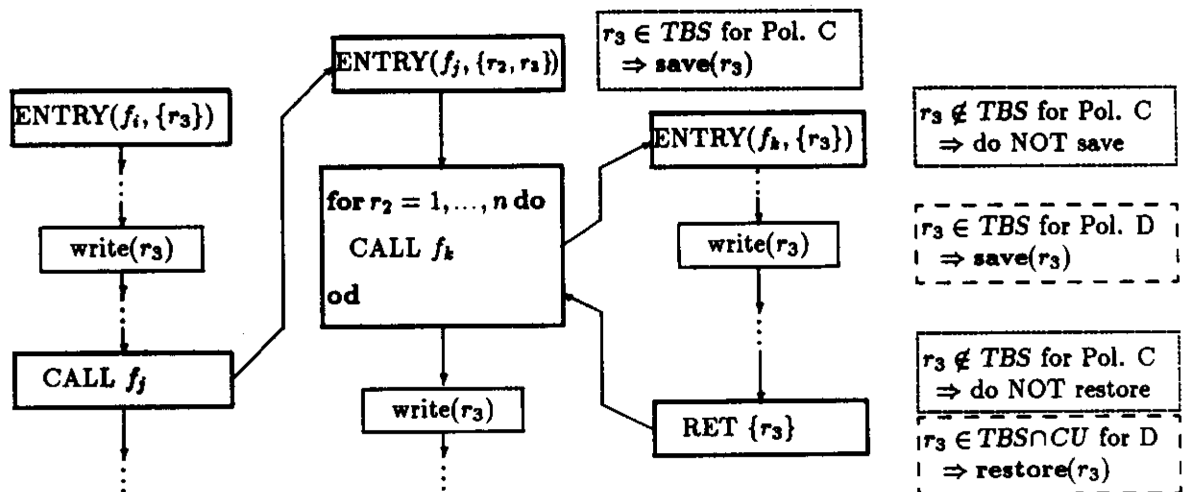


Figure 3.7: Policy D versus Policy C

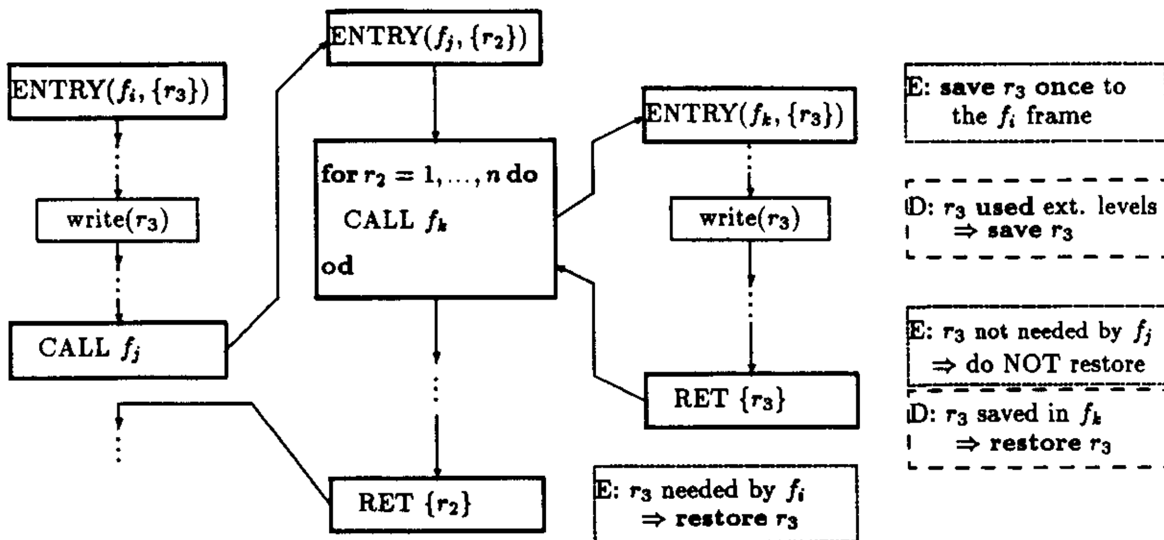


Figure 3.8: Policy E versus Policy D

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 3.7. Policy C would save register *r3* upon entry in the second function (since it would have been used previously) and, therefore, no further saving/restoring would be required upon entry in the third function (since the register would have not been used yet). However, Policy D would save/restore the register *r3* 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 upon entry to this function, it is more likely that these 80% of the functions being called will find a smaller intersection with the registers previously used.

ASM has also 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% 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 since in policies C and D there is still unnecessary saving and restoring. Consider for example the situation of Figure 3.8: in

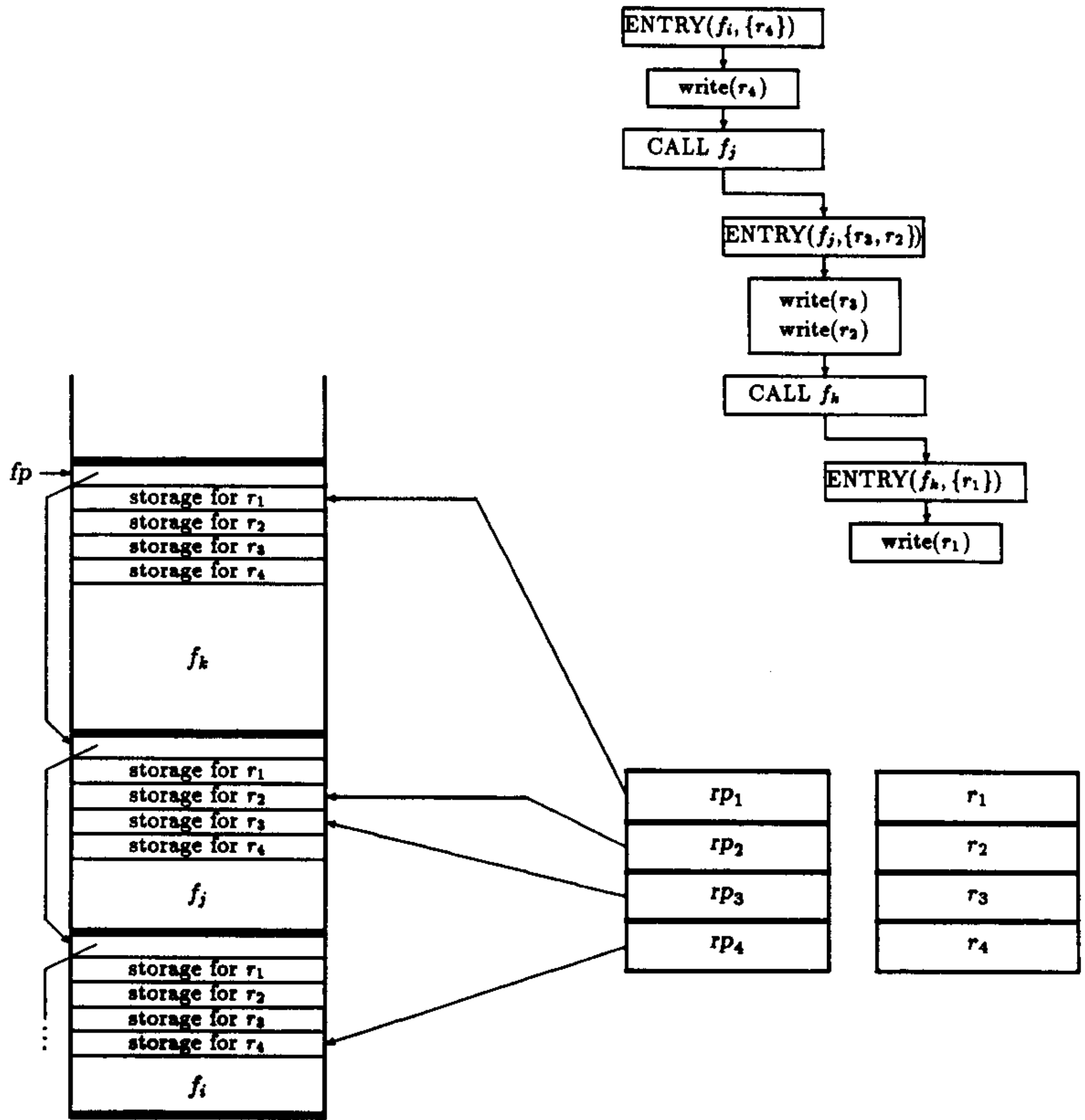


Figure 3.9: Register Pointers

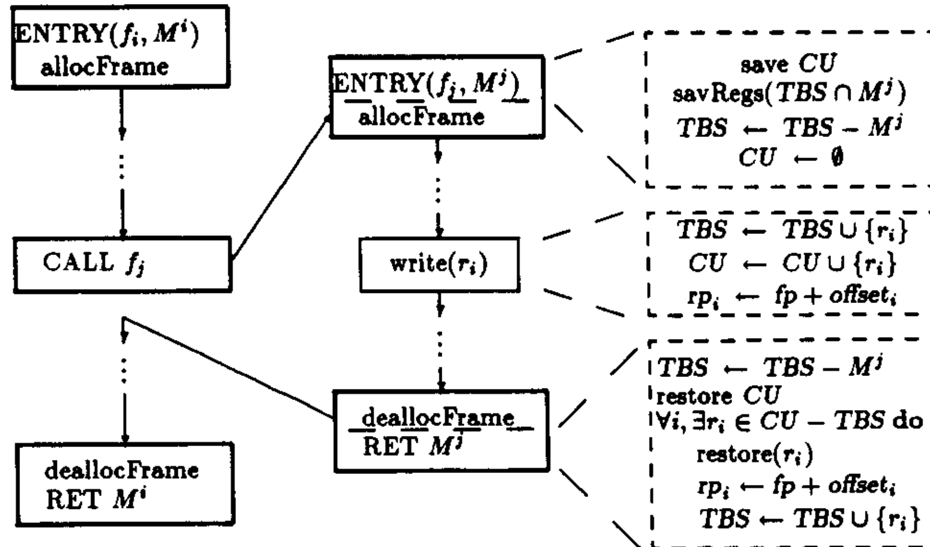


Figure 3.10: Policy E

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 done by **Policy E** (although the saving of the register at the activation record of the function which has defined it is a common feature for Policies E through H as shown in Figure 3.1). 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 3.9 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, and rp_1 to f_k. A drawback of this policy is that storage space for each register has to be provided in each frame. This is because the fp offsets where the register has to be saved are directly computed by the architecture.

Policy E requires the usage 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 3.10). A register is saved upon entry if its bit in both the TBS mask and the M mask for that function are 1 (same 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 3.10).

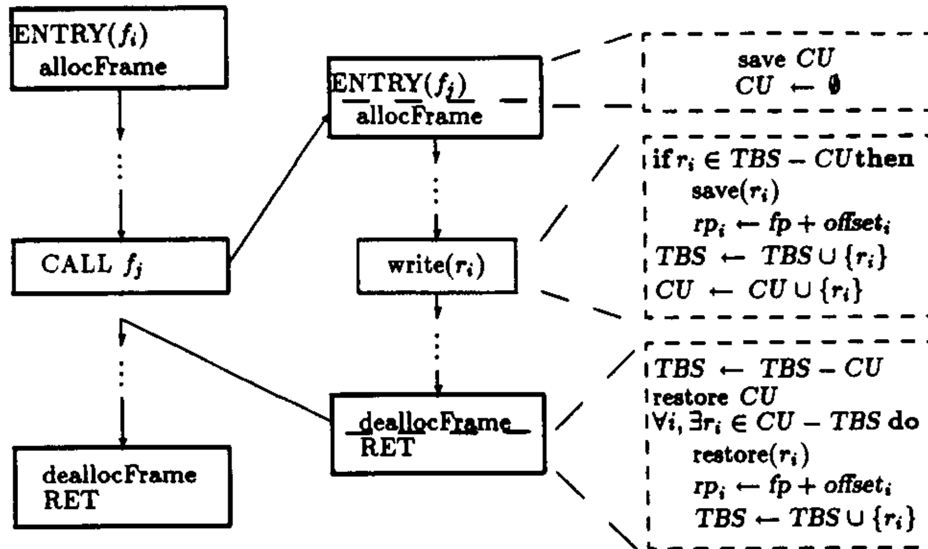


Figure 3.11: Policy F

The RSR traffic produced by Policy E can still be reduced 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 similar fashion as in Policy D. However, only one mask (CU) needs to be saved/restored (see Figure 3.11).

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 execution of the instruction. This policy uses the static mask and the dynamic *TBS* mask as Policy C, but the *TBS* mask does not need to be saved/restored across function calls (see Figure 3.12).

3. A 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 3.13). 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. This is because

- A register is not saved until the moment that it needs to be written and it was already used.
- 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 the optimal since *dead registers* (i.e., registers such that the first operation to be performed on them after the return is a write) are saved by this policy. This is because the architecture does not know that the contents of the register can be destroyed since it will never be read by the function that was using it. An optimizing compiler that performs live-variable analysis [Aho86,Hech77]

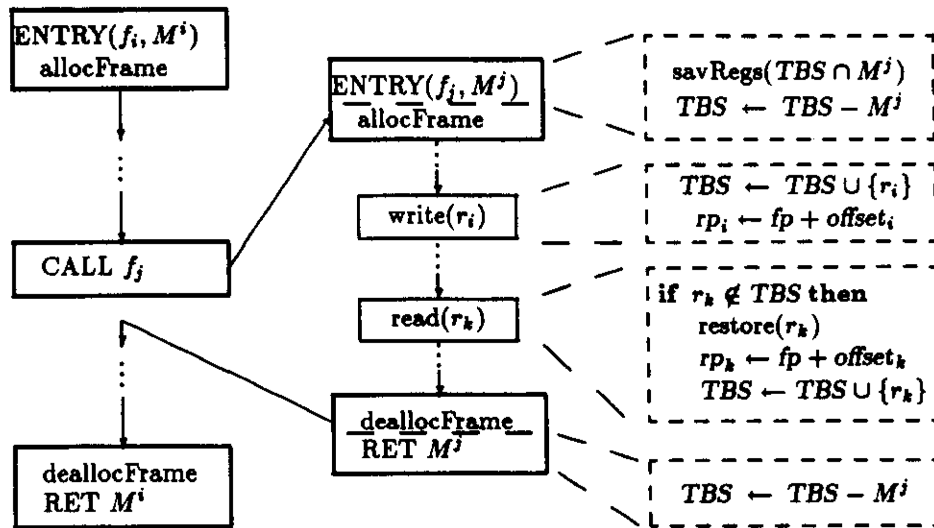


Figure 3.12: Policy G

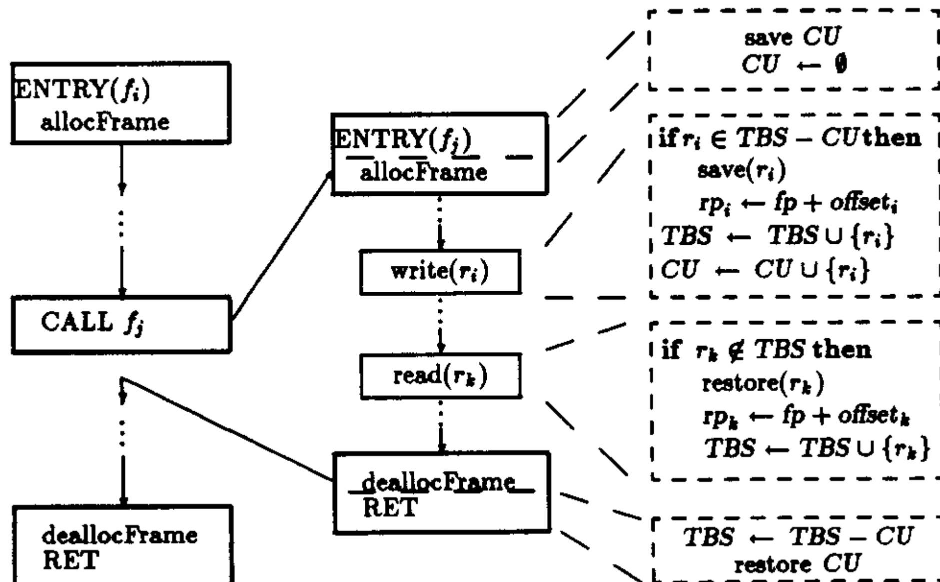


Figure 3.13: Policy H

knows this and, therefore, it could issue an instruction to clear the bits in the *TBS* mask associated to dead registers. In this case, the registers will not be saved. Notice that this can be done for each dynamic policy. This optimization is discussed in Section 4.1.4 for the 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 6–12 registers and between 10% and 12% for 16–24. As we have already discussed above, the RSR traffic reduction becomes more significant for larger register sets. However, this policy has the drawback that one mask has to be saved and restored, which increases the overall traffic (see Figure 3.2). Because of this, we select Policy G as the best candidate for implementation.

Since Policy G is the dynamic policy to be considered for implementation (see Section 2.4), 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). This is done next.

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 Policy B does), but in the activation record of the function that was using it (see Figure 3.9). In this manner, the register does not have to be restored when the function returns (as Policy B does), but when the function that was using it needs it (i.e., reads it). No explicit instruction is required for the restoring since this is performed implicitly when the register is read and is not present (see Figure 3.12). 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 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 to-be-preserved registers that are currently used (i.e., they have been written) and the whole set of to-be-destroyed registers. The whole set of to-be-destroyed registers have to be saved (and restored when the process is reschedule to run) because they are not part of the *TBS* mask (since we do not want to save them across function calls). The to-be-preserved 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 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 contents of the register is the same that 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 restoring[2] (since no write has been performed in the register). In this case, the restoring could be aborted. At this moment, it is not clear whether it is more convenient to detect this situation and to abort the restoring rather than to perform it

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

anyhow. This will be considered in the implementation of Policy G.

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 Summary

Figure 3.14 is an example that 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).

In conclusion, 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 to-be-preserved 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 to-be-preserved 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 two situations, the RSR traffic generated for the Policies A and B is 176% and 124%, respectively. Consequently, the overall data memory traffic might not be reduced for a larger register set when the conventional static RSR policies are used.

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 whole set of to-be-destroyed registers and the to-be-preserved registers that are currently used, and to restore the to-be-destroyed registers. The to-be-preserved 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.

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.

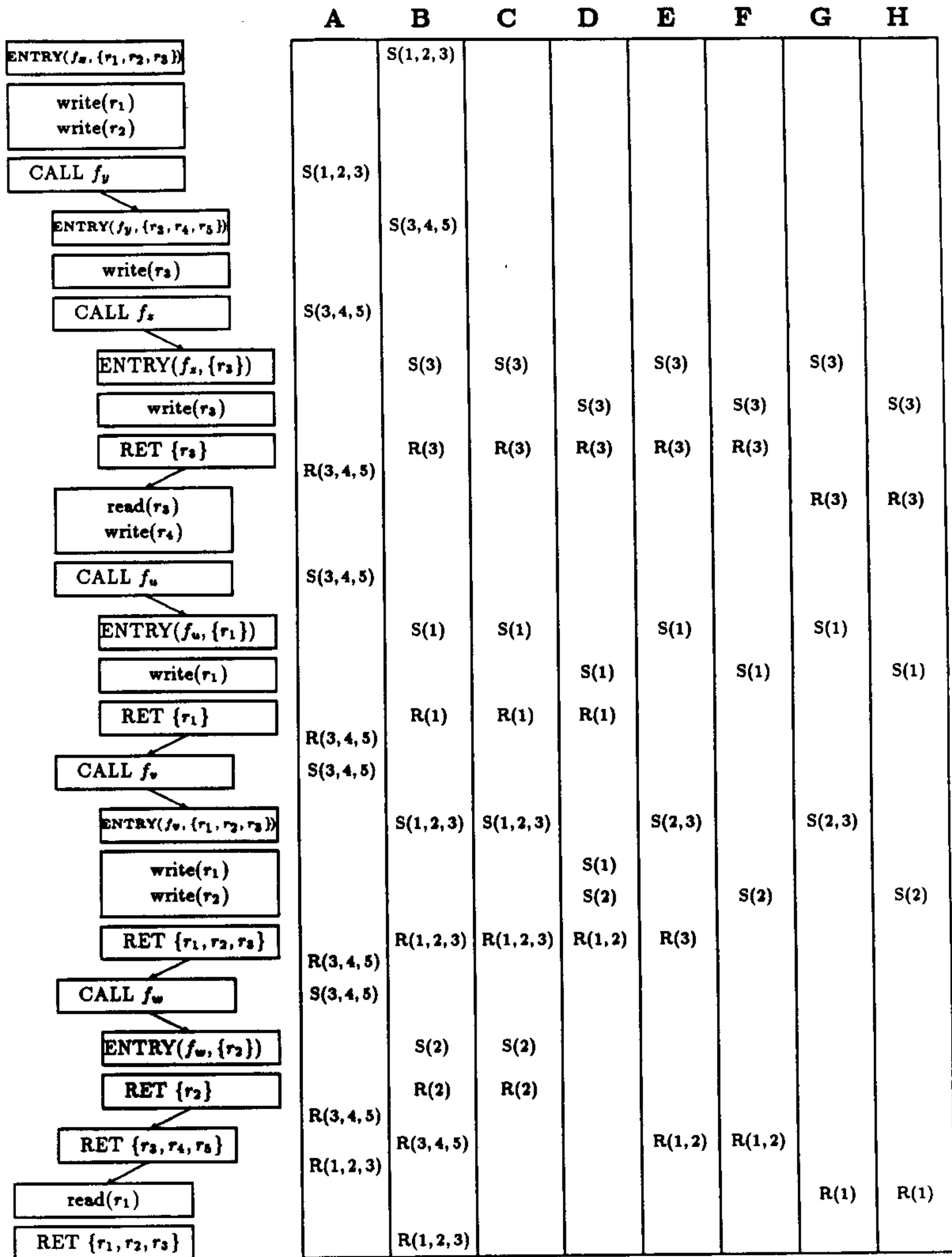


Figure 3.14: Example of RSR Traffic Caused by the Different Policies

Chapter 4

Compiler Optimizations to Reduce Register Saving/Restoring

This chapter presents several compiler optimizations to reduce the register saving and restoring traffic. From the eight register saving/restoring (RSR) policies discussed in Chapter 3, we only consider four policies for optimization: A, B, C, and G. The static Policies A and B are the conventional ones (see Section 1.2.3.4) and several optimizations have already been implemented. The dynamic Policy G is our best candidate for implementation because it generates the least memory traffic as discussed in 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 (see Section 3.2).

For each of these policies we discuss how the compiler can help to reduce the RSR traffic and we present some measurements for some of the optimizations. Measurements for each optimization will be available upon completion of this work. 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. 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 2.6).

The compiler optimizations are subdivided into two categories depending on their scope: *intraprocedural optimizations* and *interprocedural optimizations*. For the former, two existing optimizations are discussed and evaluated: live-variable analysis [Hech77,Aho86] for Policy A (renamed **A-live**) and register assignment in leaf functions for Policy B (renamed **B-lf**) and for 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 Policy G (renamed **G-live**), although it can 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]. 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 our work 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 compiler optimizations in this chapter) because several other factors have to be taken into account (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).

To be able to reduce the RSR traffic even further four interprocedural optimizations are proposed. These optimizations perform the register assignment for the variables selected by an intraprocedural register allocator. Since at this phase the whole *call graph* (i.e., the relation of which function calls which) is known, registers can be assigned is such a way that the RSR traffic is reduced. For instance, 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 for Policy A. 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**).

4.1 Intraprocedural Compiler Optimizations

This section discusses four different optimizations performed at function level. The compiler generates code per function without any knowledge on register usage required by the functions that might be called or the functions that 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 Sections 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 (Section 4.1.3). Finally, an optimization based on live-variable analysis is presented for the dynamic Policies C and G (Section 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 BKGEN (see Section 2.5) which has turned out to be a flexible tool to let us measure not only several architectural configurations, but also these intraprocedural optimizations. For instance, to measure the leaf-function optimization, the actions file (that indicates the events to measure per block) is modified as follows: the masks that indicate the read and write operations performed on the to-be-preserved registers (per block) are cleared if these registers can be transferred to the to-be-destroyed 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 BKGEN function available to measure the architectural policies can also be used. A detailed discussion on how BKGEN has been used to measure each compiler optimization is out of the scope of this report.

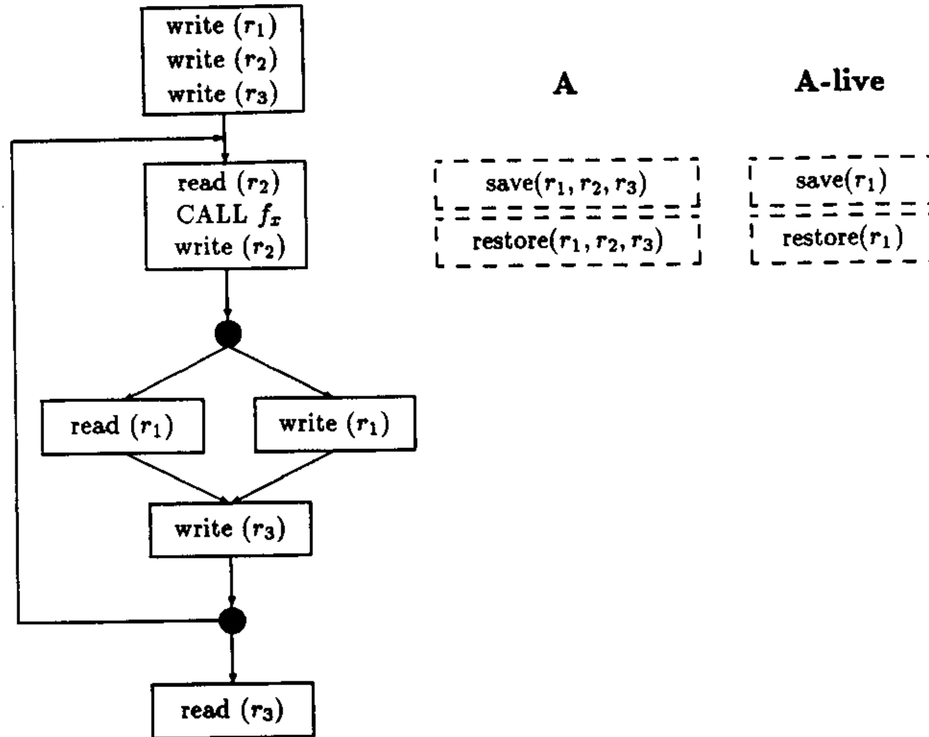


Figure 4.1: Policy A-live versus Policy A

4.1.1 Policy A-live

This is the standard intraprocedural register allocation for Policy A implemented by an optimizing compiler when variables are assigned to registers for the whole scope of the function (see Section 2.3). 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 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 to-be-preserved 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. This is because almost all the executed functions in these programs require a small number of registers. For instance, 99% of the NROFF executed functions require 6 to-

no. regs.	program	Policy A	Policy A-live	Policy A-lvOpt	Policy B	
6	ASM	1.20	0.63	0.33	1.0	(9.67)
	NROFF	1.36	0.44	0.25	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	0.38	1.0	(10.12)
	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	0.44	1.0	(10.22)
	NROFF	1.44	0.51	0.28	1.0	(4.73)
	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	0.76	0.48	1.0	(6.25)
16	ASM	2.52	1.10	0.50	1.0	(10.22)
	NROFF	1.44	0.51	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	0.52	1.0	(10.22)
	NROFF	1.44	0.51	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	0.82	0.49	1.0	(6.76)
32	ASM	4.08	1.33	0.56	1.0	(10.22)
	NROFF	1.44	0.51	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

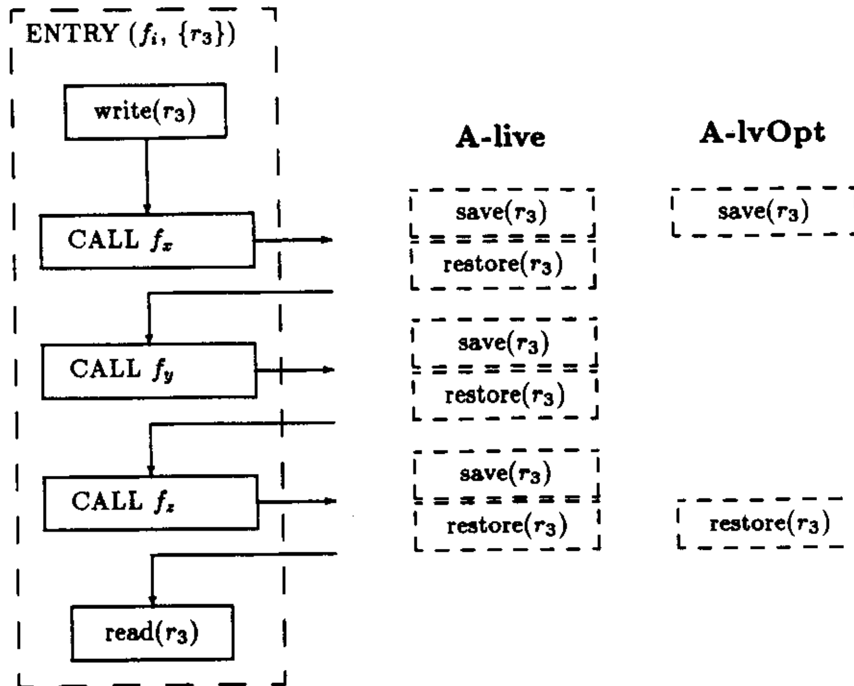


Figure 4.2: Policy A-lvOpt versus Policy A-live

be-preserved registers and 94% of the VPCC executed functions require 8. As a consequence, the RSR traffic generated by Policy A-live is about half of 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 to-be-preserved 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 to-be-preserved 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 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 the average number of live registers in a call is 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. Register r3 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**).

program	static	dynamic
ASM	14.7%	1.0%
NROFF	21.0%	5.5%
SORT	19.2%	19.5%
VPCC	23.3%	15.9%
4 P.	20.8%	10.5%

Table 4.2: Percentage of Consecutive Calls

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 contents 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 11% 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 be always 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 to-be-preserved registers are available (in contrast with Policy A-live).

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

[1]This is necessary to use the stack pointer as frame pointer and argument pointer (see Section 1.2.3.5).

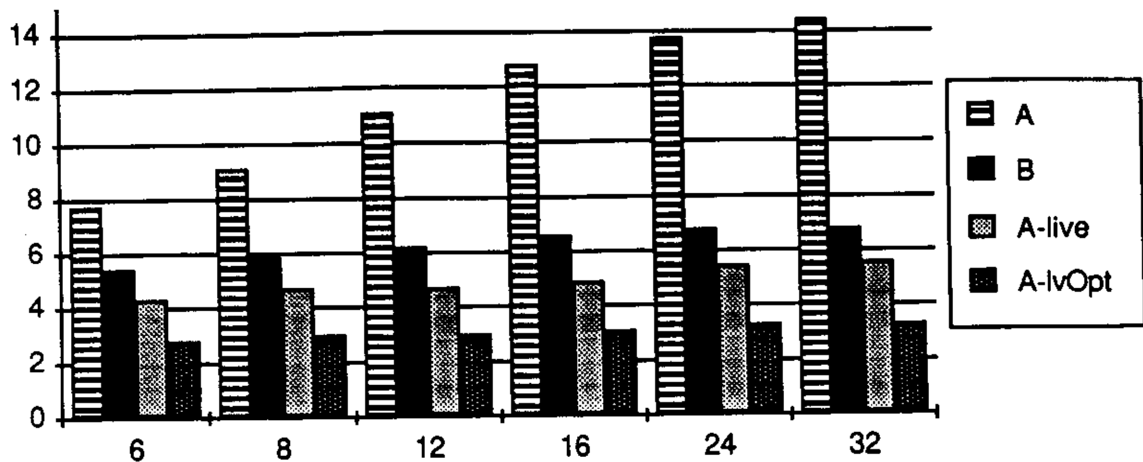


Figure 4.3: RSR Traffic for Policies A-live and A-lvOpt

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 to-be-destroyed registers are available, no RSR traffic is generated.

Moreover, to reduce the ER traffic the architecture can provide **call** and **return** instructions that save/restore the *return address* in a register (rather than to the stack). When the callee is not a leaf function, the return address is transferred 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. However, the register with the return address is saved although it is not used by the function because Policies B, C, and G save the registers upon function entry.

On the other hand, when the callee is a leaf function, the return address can remain in the register and no ER traffic is generated. This optimization can also be performed for Policies A-live and A-lvOpt if the register with the return address is saved upon 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. No measurements are available to evaluate the ER traffic reduction at the moment of writing this proposal (but they will be upon completion of this work).

Table 4.3 shows the RSR traffic generated by Policies B, C, and G with leaf-function optimization relative to the standard Policy B. Policy C with and without leaf functions includes the ER traffic caused by the *TBS* mask. Leaf measurements have been taken for 2 different number of to-be-destroyed registers: 4 and 6. The suffixes **-lf4** and **-lf6** have been added to each policy to denote the number of to-be-destroyed registers used. As we can see in the table, the results obtained for 6 to-be-destroyed registers are very similar to the ones obtained for 4. Since our previous measurements [Hugu85a] have showed that 4 is an appropriate number for the to-be-destroyed

no. regs.	program	Policy B	B-lf4	B-lf6	Policy C	C-lf4	C-lf6	Policy G	G-lf4	G-lf6
6	ASM	1.0 (9.67)	0.89	0.85	0.95	0.84	0.81	0.23	0.19	0.19
	NROFF	1.0 (4.71)	0.86	0.84	1.07	0.98	0.98	0.19	0.17	0.17
	SORT	1.0 (4.18)	0.63	0.63	1.37	1.09	1.09	0.56	0.28	0.28
	VPCC	1.0 (7.60)	0.92	0.88	1.07	0.99	0.97	0.39	0.34	0.34
	4 P.	1.0 (5.44)	0.84	0.82	1.11	0.99	0.98	0.31	0.24	0.24
8	ASM	1.0 (10.12)	0.89	0.86	0.94	0.84	0.80	0.23	0.20	0.19
	NROFF	1.0 (4.73)	0.86	0.85	1.09	1.04	1.03	0.17	0.16	0.16
	SORT	1.0 (5.00)	0.69	0.69	1.31	1.00	1.00	0.47	0.28	0.28
	VPCC	1.0 (8.73)	0.93	0.90	1.01	0.96	0.94	0.36	0.32	0.32
	4 P.	1.0 (5.88)	0.86	0.84	1.09	0.99	0.98	0.29	0.24	0.23
12	ASM	1.0 (10.22)	0.89	0.86	0.91	0.82	0.79	0.20	0.19	0.19
	NROFF	1.0 (4.73)	0.86	0.85	1.03	0.98	0.98	0.12	0.11	0.11
	SORT	1.0 (6.60)	0.76	0.76	1.12	0.94	0.94	0.40	0.22	0.22
	VPCC	1.0 (8.91)	0.93	0.90	0.96	0.91	0.88	0.25	0.22	0.20
	4 P.	1.0 (6.25)	0.86	0.84	1.02	0.94	0.93	0.23	0.17	0.17
16	ASM	1.0 (10.22)	0.89	0.86	0.83	0.79	0.79	0.16	0.15	0.15
	NROFF	1.0 (4.74)	0.86	0.84	0.83	0.78	0.77	0.09	0.09	0.08
	SORT	1.0 (8.15)	0.81	0.81	0.89	0.89	0.89	0.18	0.18	0.18
	VPCC	1.0 (9.03)	0.93	0.90	0.98	0.94	0.92	0.22	0.22	0.22
	4 P.	1.0 (6.59)	0.87	0.85	0.89	0.85	0.85	0.16	0.15	0.15
24	ASM	1.0 (10.22)	0.89	0.86	0.82	0.79	0.76	0.14	0.14	0.14
	NROFF	1.0 (4.74)	0.86	0.84	0.81	0.77	0.77	0.06	0.06	0.06
	SORT	1.0 (8.93)	0.83	0.83	0.78	0.69	0.69	0.13	0.11	0.11
	VPCC	1.0 (9.10)	0.93	0.90	0.82	0.80	0.80	0.16	0.16	0.16
	4 P.	1.0 (6.76)	0.88	0.86	0.81	0.76	0.76	0.12	0.11	0.11
32	ASM	1.0 (10.22)	0.89	0.86	0.79	0.76	0.76	0.15	0.14	0.14
	NROFF	1.0 (4.74)	0.86	0.84	0.74	0.69	0.69	0.05	0.05	0.04
	SORT	1.0 (8.93)	0.83	0.83	0.89	0.80	0.80	0.21	0.16	0.16
	VPCC	1.0 (9.10)	0.93	0.90	0.86	0.83	0.82	0.17	0.17	0.17
	4 P.	1.0 (6.76)	0.88	0.86	0.82	0.77	0.76	0.14	0.12	0.12

Table 4.3: RSR Traffic with Leaf-Function Optimization

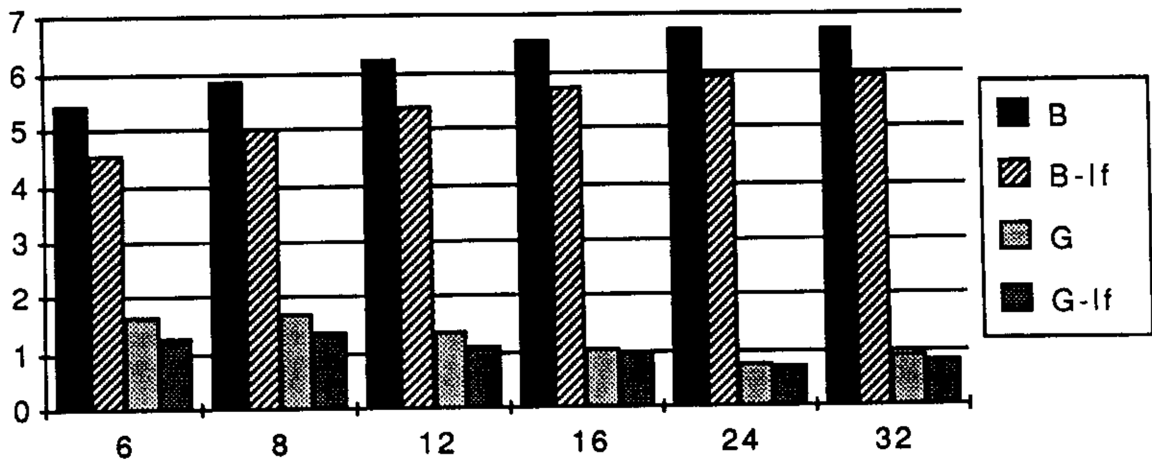


Figure 4.4: Leaf-Function Optimization (Policies B and G)

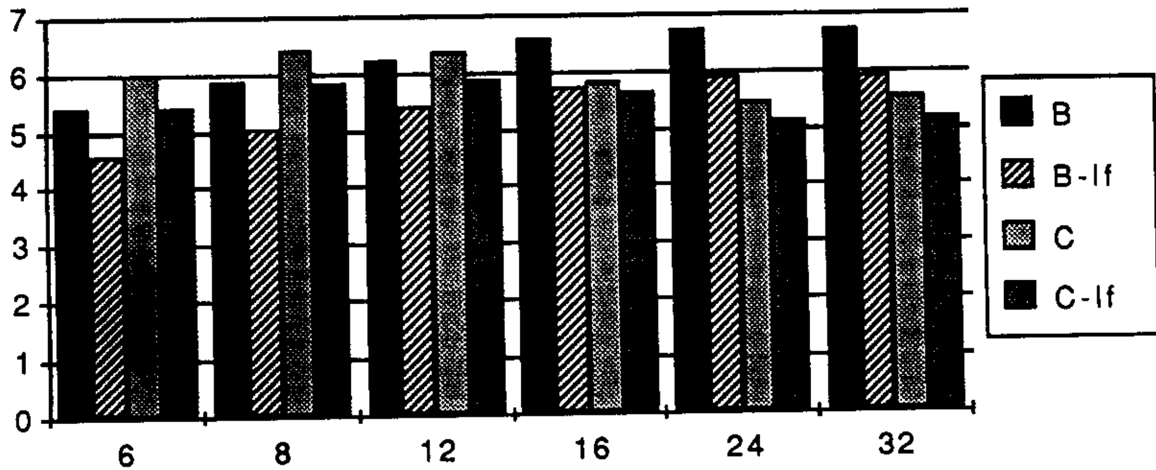


Figure 4.5: Leaf-Function Optimization (Policies B and C)

registers (see Section 1.2.4), we will restrict ourselves to this number for future measurements and comparisons. For the rest of this report, the suffix **-lf** corresponds to -lf4.

Policy B-lf generates between 84% and 88% of the RSR traffic produced by the standard Policy B. Policy G-lf generates between 77% and 94% of the RSR traffic produced by Policy G without the leaf-function optimization. The RSR traffic reduction is more significant for smaller register sets (see Figure 4.4). This is because when a larger number of registers is available, a leaf function probably finds that the defined registers have not been used previously.

Policy B-lf generates less traffic than Policy C-lf when there are 12 to-be-preserved registers or less (see Figure 4.5). Policy B-lf has between 85% and 91% of the traffic generated by Policy C-lf. However, for a larger register set Policy C-lf performs better: it has from 98% of the Policy B-lf traffic when 16 to-be-preserved registers are available to 88% when there are 32.

As we have already discussed in Chapter 3, less RSR traffic is generated for the dynamic Policy G when more to-be-preserved 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 Section 4.1.1). Thus, its contents 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).

Since this optimization is performed per function, the callee does not know which registers are alive for the caller. Thus, the compiler has to generate an extra instruction before the call to clean the bits of the *TBS* mask (To Be Saved; see Section 3.2) associated with the dead registers.

No measurements are available for this optimization yet. However, we can expect the following. We know that when, for instance, 12 to-be-preserved registers are available, there are, on the average, 3.13 registers defined per executed function and 2.36 of them are alive. Thus, at most we can get a reduction in RSR traffic of 25% if we assume that all the registers are always used at exterior levels. However, this is not the case. On the average, 70% of the registers have been previously used with Policy C (2.19 out of 3.13 registers per function) and only 22% of the registers have been previously used with Policy G (0.70 registers). Therefore, the RSR traffic reduction obtained is probably small (between 5% and 17% for 12 registers) and, in this case, it would not pay off the increase in instruction memory traffic caused by the extra instructions. Measurements will be taken to confirm or deny this hypothesis.

4.1.5 Summary

Figure 4.6 shows the RSR traffic for the optimized Policies B-lf, A-live, A-lvOpt, and G-lf. When we compare the optimized policies (except Policy G-live), we can conclude the following:

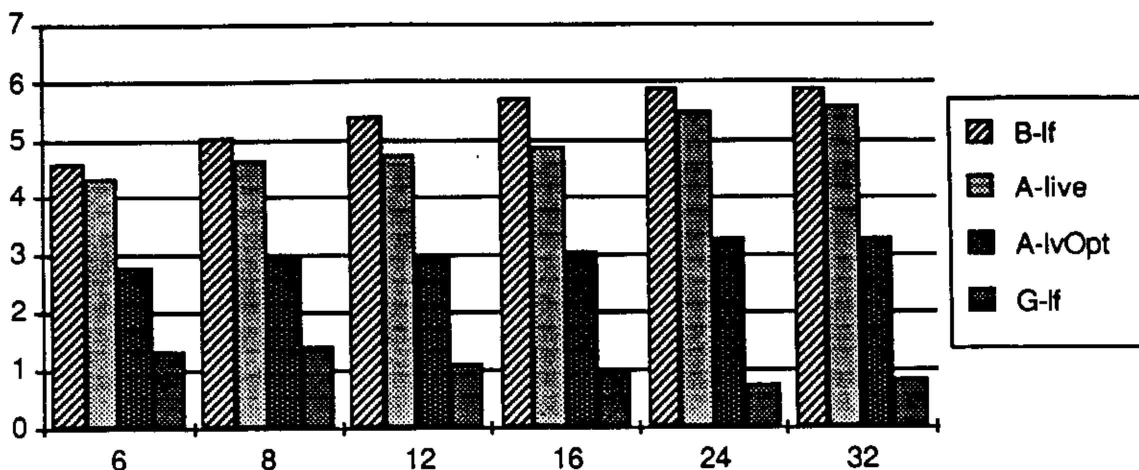


Figure 4.6: RSR Traffic for Optimized Policies

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 Section 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 has always less RSR traffic than Policy B-lf, but Policy A-live generates more RSR traffic when 16 or more to-be-preserved registers are available (see Figure 4.7). 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. However, this cannot be done with intraprocedural register allocation and assignment because the compiler does not have a global view of the program[2]. This will be possible with interprocedural register assignment as we will discuss in Section 4.2.

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 4.6). 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 to-be-preserved registers are available; however, Policy C-lf performs better than B-lf for 16 (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. For this reason, this policy is not considered in Section 4.2 for interprocedural optimization[3].

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 to-be-preserved registers) and 22% (for 24) of the RSR traffic generated by Policy A-lvOpt.

[2]In this case, the compiler should provide an option so that the programmer could specify the policy to be selected.

[3]Notice that if the hypothesis of Section 4.1.4 results to be wrong, then this conclusion will also be wrong and we will have to consider Policy C-live for interprocedural optimization.

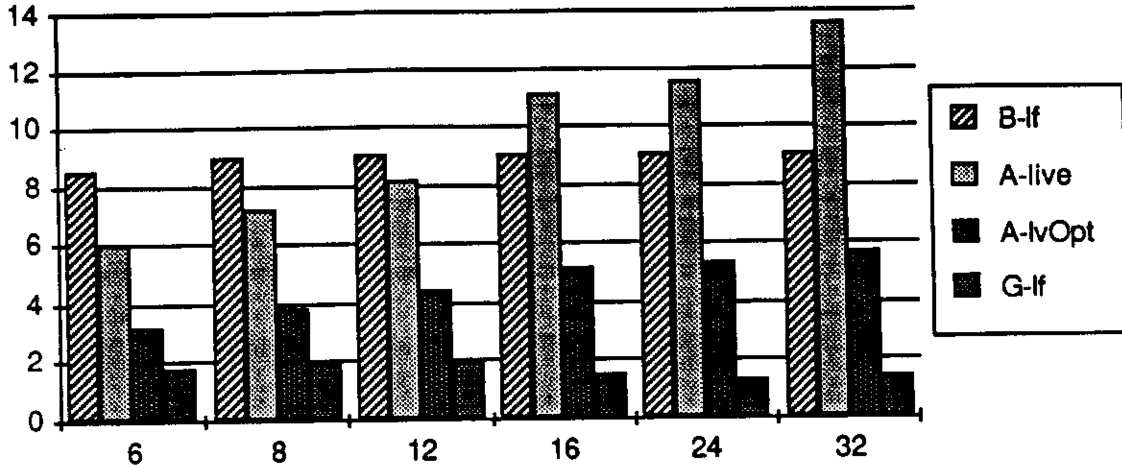


Figure 4.7: ASM RSR Traffic for Optimized Policies

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 mentioned in Section 3.4 when no optimizations were performed.

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

1. The compiler implementation is simplified.
2. The threshold value to decide when a variable is allocated (see Section 1.2.3.4) is smaller since the RSR traffic has been reduced. Consequently, more variables can be allocated and, therefore, the overall traffic might also be reduced.

4.2 Interprocedural Compiler Optimizations

This section presents four compiler optimizations that are performed globally, that is, over the whole program. These optimizations are done on Policies A-live, A-lvOpt, B-lf, and G-lf[4]. All four optimizations are based on having a *disjoint register assignment* for caller-callee pairs (whenever possible). Once functions have disjoint registers, instructions that perform unnecessary register saving/restoring for the static Policies A-live, A-lvOpt, and B-lf are eliminated. These instructions save/restore registers that are not going to be used in the functions that might be called (for A-live and A-lvOpt) or that are not used for the functions in the exteriors levels (for B-lf). Notice that

[4]If our hypothesis in Section 4.1.4 is false, then we will select G-live-lf as the candidate for interprocedural optimization.

this elimination is not necessary for Policy G-lf since the unnecessary RSR traffic is never generated because of the dynamic behavior of this policy (see Section 3.3).

In Section 1.2.3.5 we reviewed two current interprocedural register allocators [Wall86,Stee87]. 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 by the non-allocated variables is greater than the traffic that it would be generated if some registers were saved/restored to free registers for these variables. 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 Steekiste. His approach is based on Policy A-live. We discuss the differences between our approaches for Global Policy A-live in Section 4.2.1.

Sections 4.2.1 to 4.2.4 discuss the Global Policies 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 interprocedural optimization is that Policy A-lvOpt can only be partially optimized as we will explain in Section 4.2.2. Thus, we have to compare both optimized policies to verify whether Policy A^g-lvOpt is still better than Policy A^g-live.

Before commenting each policy, we discuss how the interprocedural optimizer can be incorporated in the compilation process and what is the type of information that should be made available to it.

Since C is a modular programming language and modules are compiled independently, the following steps are taken to be able to perform interprocedural optimization:

1. For each function, the compiler performs live-variable analysis (Policies A^g-live and A^g-lvOpt), selects the variables for allocation, and assigns them to registers. This assignment is performed sequentially and it might be changed later by the interprocedural optimizer. That is, if $V^i = \{v_1^i, v_2^i, \ldots, v_m^i\}$ is the set of variables selected for allocation in function f_i and $R = \{r_1, r_2, \ldots, r_n\}$ is the set of to-be-preserved registers, then variable v_1^i is assigned to register r_1, v_2^i to r_2, and so on.
2. For each module the compiler updates a data base with the following information:
   - Functions defined per module ($F = \{f_1, f_2, \ldots, f_p\}$).
   - Variables allocated per function (V^i, $\forall f_i \in F$). We assume that we have only one register set for local scalar variables and all registers are of the same type. Otherwise, we should also keep information about each variable.
   - 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).
     - A static priority associated with each call instruction[5] ($Q_{i \rightarrow j}$). This is to estimate the RSR traffic generated by this call.
     - For Policy A^g-live, live-variable information at each call ($L_{i \rightarrow j}$).
     - For Policy A^g-lvOpt, the registers to be saved and restored at each call ($S_{i \rightarrow j}$ and $R_{i \rightarrow j}$).

[5] Function calls inside a loop should have a higher priority.

Notice that in a given function f_i multiple calls to the same function f_j generate different $L_{i\rightarrow j}$, $S_{i\rightarrow j}$, $R_{i\rightarrow j}$, and $Q_{i\rightarrow j}$ sets even though this is not reflected in our notation[6].

- Functions that 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 interprocedural optimizations because no disjoint register assignment can be found (since all the functions might be called). To limit the scope of indirect calls to a (expected small) set of functions this information must be provided.

For each module the compiler generates assembly code instead of object code. The reason for this is given below.

3. Once all the user modules have been compiled, the source modules for the library functions that are required by the program are also compiled. The process of detecting which source modules needs 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.

4. Once the user modules and the library modules (required by the program) have been compiled, the interprocedural optimizer generates a *call graph*. In this graph, each defined function has a node associated with it and each directed arc corresponds to each single call made from the function. Each node has associated the set of variables V^i selected for allocation. Each arc has associated the static priority associated with the call ($Q_{i\rightarrow j}$) and some additional information depending on the policy to be considered ($L_{i\rightarrow j}, S_{i\rightarrow j}, R_{i\rightarrow j}$). Notice that the graph may have cycles if there are *recursive functions*, that is, functions that call themselves directly ($f_j \rightarrow f_j$) or through other functions ($f_{j_1} \rightarrow f_{j_2} \rightarrow \ldots \rightarrow f_{j_q} \rightarrow f_{j_1}$).

   The interprocedural optimizer performs disjoint register assignment and the registers that are saved and restored unnecessarily for Policies A^S-live, A^S-lvOpt, and B^S-lf are detected (as discussed in Sections 4.2.1, 4.2.2, and 4.2.3) 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 interprocedural optimizer is kept on the data base for next phase.

5. Finally, the assembly modules are translated to object code in such a way that:

   - Registers assignment is changed as defined by the interprocedural optimizer.
   - Instructions that perform unnecessary register saving/restoring are removed.

   If only the register numbers have been changed, then the assembly code would not be necessary since this operation could be directly performed on the object code. However, since some instructions might be eliminated and, therefore, some addresses might be modified, the assembly code is necessary.

[6] A 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.

Notice that global scalar variables are not considered for allocation. This is a consequence that our interprocedural optimizer runs after each module has been assembled. If an interprocedural alias analyzer was gathering information before the register allocation phase (as proposed in [Cout86a]), then global scalar variables could be considered for allocation. However, this is not considered in our work due to time limitations. Our measurements have shown that global scalar variables generate 14% of the data memory traffic while local scalar variables generate 61% (when no RSR traffic is considered).

Although the interprocedural optimization makes the compilation process more complex, it can be justified if the RSR traffic reduction is significant. However, no results are available at the moment of writting this proposal.

To perform the disjoint register allocation mentioned in the step number 4 above we need to estimate the frequency of each call (for Policies A^g-live, A^g-lvOpt, and G^g-lf) or the frequency that each function is called (for Policy B^g-lf). Wall proposes to do so by ***adding*** the ***static*** priority provided by the compiler[7] [Wall86]. For instance, if f calls g with priority 10 and g calls h with 20, then h is estimated to be executed 30 times. However, this does not seem to be appropriate because functions at the bottom of the graph (i.e., leaves) are going to have a much higher frequency than functions at the beginning (i.e., closer to the root). Therefore, we also plan to use ***dynamic*** information to know the frequency of each call for at least one program execution. The usage of dynamic information for compiler optimization is not new. Ellis [Elli86] uses dynamic information to detect parallelism among basic blocks and Wall also uses it to determine which variables should be allocated to registers.

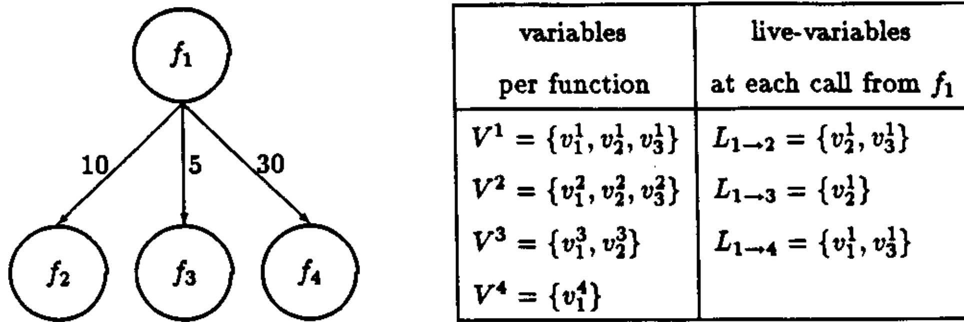
The dynamic information can be added to the data base for a non-optimized program version and be also available to the interprocedural optimizer. We expect to compare the compiler optimization results using both static and dynamic information and conclude that the optimizer can do a better job with dynamic information. We will also consider the number of different program executions (i.e., input data) required to obtain a good estimation. We expect that only one program execution will be enough for programs like ASM, NROFF, SORT, and VPCC, but not for SPICE. For SPICE we will have to perform several runs because of the mutually exclusive execution paths for different simulation parameters.

The following subsections describe what we expect that the optimizer performs for each interprocedural policy.

4.2.1 Global A-live

The Global A-live Optimization (**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. Although the goal (to reduce the RSR traffic) and the method (disjoint register assignment) is the same for each interprocedural optimization, the algorithms themselves are different for each policy. 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

[7] He claims that if the priority is ***multiplied***, the results gave worse estimations.



variables per function	live-variables at each call from f_1
$V^1 = \{v_1^1, v_2^1, v_3^1\}$	$L_{1\to 2} = \{v_2^1, v_3^1\}$
$V^2 = \{v_1^2, v_2^2, v_3^2\}$	$L_{1\to 3} = \{v_2^1\}$
$V^3 = \{v_1^3, v_2^3\}$	$L_{1\to 4} = \{v_1^1, v_3^1\}$
$V^4 = \{v_1^4\}$	

Figure 4.8: Example of Register Assignment for Policy A^S-live

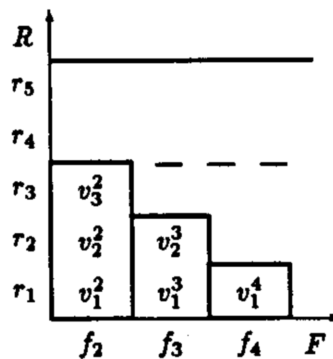


Figure 4.9: Registers Assigned to Functions f_2, f_3, and f_4

approach that we use to solve the problem; second, we present an algorithm to solve the problem when the program does not have recursive functions; third, we mention how the ER traffic can be reduced by this policy; and finally, we explain how to handle recursive functions in a program.

Our approach is based on the fact that functions which are never active simultaneously can share the same set of registers without having to save/restore them. We start assigning variables to registers for the leaf functions. Since these are never going to be active simultaneously, they can share the same set of registers. Next step is to select a function one level above (i.e., a function such that all its descendants 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. Let us illustrate this with the example given in Figure 4.8.

Figure 4.8 shows four functions that 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 to-be-preserved registers available ($R = \{r_1, r_2, r_3, r_4, r_5\}$). Figure 4.9 shows the registers already assigned for the functions f_2, f_3, and f_4. Let us discuss now how 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]. Day [Day70] used integer programming to perform register allocation, but not for assignment.

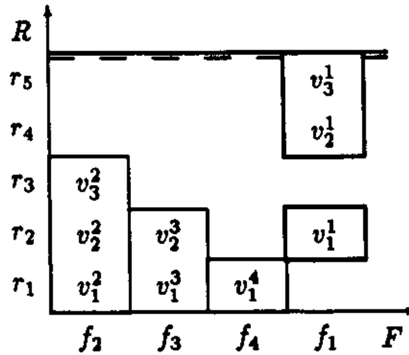


Figure 4.10: Variable-to-Register Assignment for Function f_1

The cost of assigning a variable to a register is given as the sum of RSR traffic generated at each call 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 cost[8] 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_{ij} be the cost of assigning variable v_j^1 to register r_i. For this example, the cost matrix C is:

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

Applying the procedure described in [Hill67] to find a solution to an assignment problem, we get that the optimal solution is the one shown in Figure 4.10. In this particular case, the RSR traffic generated by this assignment will be zero. 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 that are not used for any of the functions being called is zero, these registers are the best candidates to be selected for assignment. This is similar to the approach taken by Steenkiste [Stee87], except that he does not consider the cost matrix, but only the number of registers already assigned to the function descendants. However, this is not a good policy because the RSR traffic eliminated corresponds to functions that are at the bottom of the call graph which not necessarily are the ones that generate more RSR traffic.

To overcome this limitation, registers that 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 4.11 shows another variable-to-register assignment for $K = 11$. Here variable v_2^1 gets assigned

[8] Since the saving traffic is the same than the restoring traffic, only the saving is considered to compute the cost.

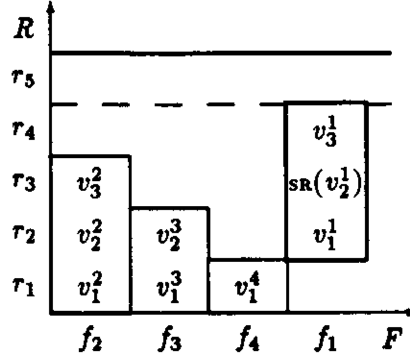


Figure 4.11: Variable-to-Register Assignment for Function f_1 when $K = 11$

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.

Our idea is to define K as a fraction of the number of calls in the path from the *main* function to the current one. Thus, functions in the bottom of the call graph with low execution frequency can share the same registers as their descendants. This is one of the reasons why we would like to have dynamic data on the number of executed calls. We expect to try several possibilities for K and report on them upon completion of this work.

We now present an algorithm that describes our approach to perform register assignment. This algorithm assumes that the call graph is acyclic, i.e., the program does not have recursive functions. At the end of this section we will discuss how to handle recursive functions.

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

Inputs: An acyclic 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 ($L_{i \to j}$), and the static or dynamic frequency of each call ($Q_{i \to j}$).

Outputs: A set of registers to be saved/restored per call ($SR_{i \to j}$) 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 4.12) and a set U^j per function f_j to indicate the registers used by this function and all its descendants in the call graph. Moreover, the following definitions are required:

$$\text{REG_USED}(r_i, U^j) = \begin{cases} 1 & \text{if } r_i \in U^j \\ 0 & \text{otherwise} \end{cases}$$

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

for each $f_j \in F$ visited in reverse depth-first search order **do**
if f_j is a leaf function **then**
assign V^j variables to registers:
$a_{1j} \leftarrow v_1^j,\ a_{2j} \leftarrow v_2^j,\ \ldots,\ a_{mj} \leftarrow v_m^j$
$U^j = \{r_1, r_2, \ldots, r_m\}$
else
for each $v_k^j \in V^j$ **do**
for each $r_i \in R$ **do**
if $r_i \in \bigcup_{\forall t, j \to t} U^t$ **then**
$c_{ik} = \sum_{\forall t, j \to t} \left\{ Q_{j \to t}\, \mathrm{REG_USED}(r_i, U^t)\, \mathrm{LIVE_VAR}(v_k^j, L_{j \to t}) \right\}$
else
$c_{ik} = K_j$
fi
od
od
Perform register assignment as given in [Hill67]
Let variables $v_1^j, v_2^j, \ldots, v_m^j$ be assigned to registers $r_{i_1}, r_{i_2}, \ldots, r_{i_m}$
Update column j for matrix A:
$a_{i_1,j} \leftarrow v_1^j,\ \ldots,\ a_{i_m,j} \leftarrow v_m^j$
Compute U^j:
$U^j \leftarrow \{r_{i_1}, r_{i_2}, \ldots, r_{i_m}\} \cup (\bigcup_{\forall t, j \to t} U^t)$
Compute $SR_{j \to t}$:
for each $v_k^j \in V^j$ **do**
for each $f_t \in F$ such that $j \to t$ **do**
if $r_{i_k} \in U^t$ **then** $SR_{j \to t} \leftarrow SR_{j \to t} \cup \{r_{i_k}\}$ **fi**
od
od
fi
od

Figure 4.12: Register Assignment for Policy A^s-live

Also, new L and Q sets have to be defined:

$$\forall k = j_1, j_2, \ldots, j_q \wedge t \neq j_1, j_2, \ldots, j_q \text{ such that } \exists k \rightarrow t$$
$$L_{s \rightarrow t} \leftarrow \emptyset$$
$$Q_{s \rightarrow t} \leftarrow Q_{k \rightarrow t}$$

Notice that we keep the frequencies to be able to compute K for the functions being called from $f_{j_1}, f_{j_2}, \ldots, f_{j_q}$. However, we do not keep live-variable information because registers have already been assigned as we described above. Finally, f_s has to be marked so that the algorithm given in Figure 4.12 can be modified to detect that registers have already been assigned for this pseudo-function and to compute U^s accordantly. Therefore, all cycles are removed from the call graph and the Algorithm 1 can be used to perform register assignment.

4.2.2 Global A-lvOpt

The goal of the Global Policy A-lvOpt (**A^g-lvOpt**) is the same as the one of 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 Algorithm 1 to perform this. The reason is that 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 descendant 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 interprocedural optimization so that we can verify whether A^g-lvOpt still performs better than A^g-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 our algorithm to perform register assignment for Policy A^g-lvOpt. As for Policy A^g-live, the ER traffic can also be reduced using the same approach. This is not discussed further.

The saving/restoring of a register which is not used by the function that its being called and its descendants 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 4.13.(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 descendants 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 4.13.(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 descendants 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.

Method: Apply algorithm given in Figure 4.12. Notice that the algorithm does not consider the assignment of some local scalar variables to to-be-destroyed registers for leaf functions. For intraprocedural 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 interprocedural allocation, we can assign local scalar variables to to-be-destroyed 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. □

The Policy A^s-live can also reduce the ER traffic if the architecture provides call and return instructions such that the programmer (i.e., the compiler) can specify any general-purpose register to save the *return address* (rather than to save it to the stack or to an implicit register). An intraprocedural allocator has to save the return address always in the same register (assuming that these instructions are provided) because the callee would not know the register used by the caller. In this case, the ER traffic can only be reduced by the leaf-function optimization (see Section 4.1.3).

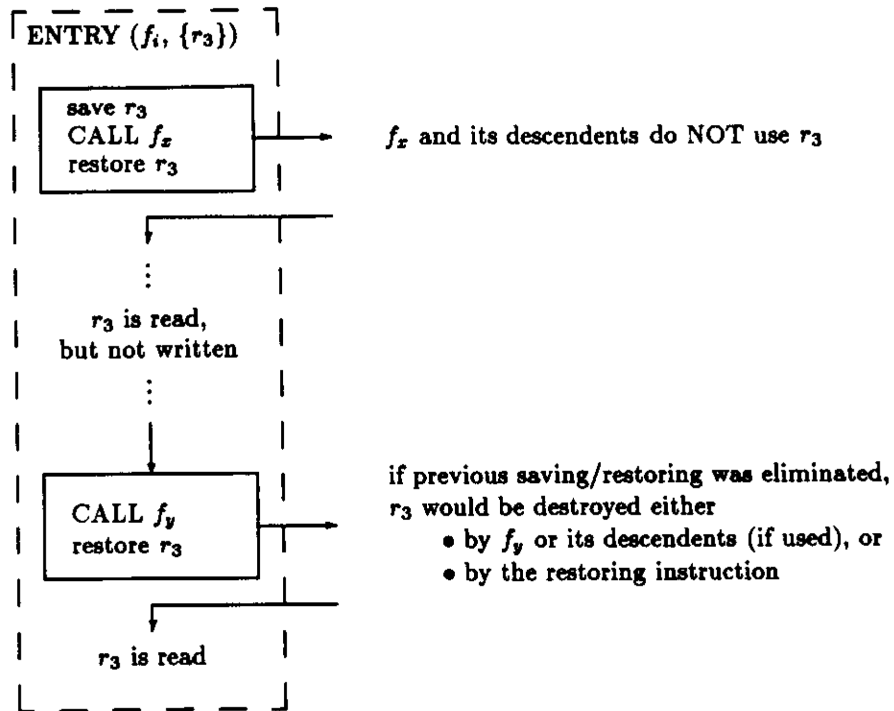
On the other hand, an interprocedural allocator can save it in different registers so that if the register is not used by any of the function descendants, no saving/restoring traffic for the return address is generated. Thus, the register with the return address also competes for a disjoint register assignment like the other registers with variable values do. However, this assignment is more restrictive than the one made for registers with variable values because once a function has decided to assign the return address to a specific register, all its callers must use this register. If no disjoint register assignment can be made, this register should be saved upon entry and restored before return to prevent the saving/restoring in every call (since it is alive from entry to exit).

Finally, let us discuss how to handle recursive calls. Let $f_{j_1}, f_{j_2}, \ldots, f_{j_q}$ be functions in the call graph that are enclosed in a cycle (i.e., $f_{j_1} \to f_{j_2} \to \ldots \to f_{j_q} \to f_{j_1}$). No RSR traffic can be eliminated for the registers defined by these functions because the registers might be used by one of their descendants (i.e., $f_{j_1} \xrightarrow{*} f_{j_1}$). Thus, we can group all these functions in a single *pseudo-function* ($f_s, s \neq 1, 2, \ldots, p$), add this to the call graph G and to the set of defined functions F, and remove the functions in the cycle from the call graph and from F. Since all the registers assigned to functions in the cycle are candidates to be used by one of the function descendants, the caller has always to save/restore the registers used by any of the functions in the cycle. Thus,

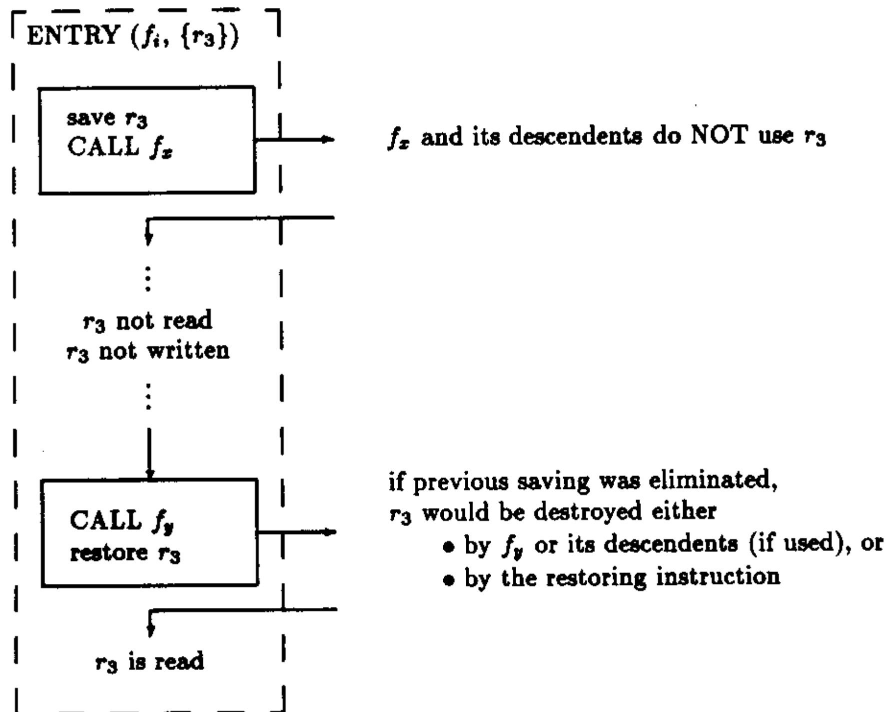
- All the functions in the cycle get assigned the same register numbers because the registers are always going to be saved/restored.
- The number of registers required by this pseudo-function is the maximum number of registers required by any of the functions in the cycle ($\max\{|V^{j_1}|, \ldots, |V^{j_q}|\}$).

Given the above, the columns $j_1, j_2, \ldots, j_q$ of matrix A and the sets SR associated with the calls for the functions in the cycle are easily computed as:

$$\begin{aligned}
&\forall k = j_1, j_2, \ldots, j_q \\
&\qquad a_{1k} \leftarrow v_1^k,\ a_{2k} \leftarrow v_2^k,\ \ldots,\ a_{mk} \leftarrow v_m^k \\
&\qquad SR_{k \to t} \leftarrow \text{registers assigned for variables in } L_{k \to t}
\end{aligned}$$



(a) Eliminated Saving by Policy A-lvOpt



(b) Eliminated Restoring by Policy A-lvOpt

Figure 4.13: Cases for which Policy A^{s}-lvOpt Cannot Be Optimized

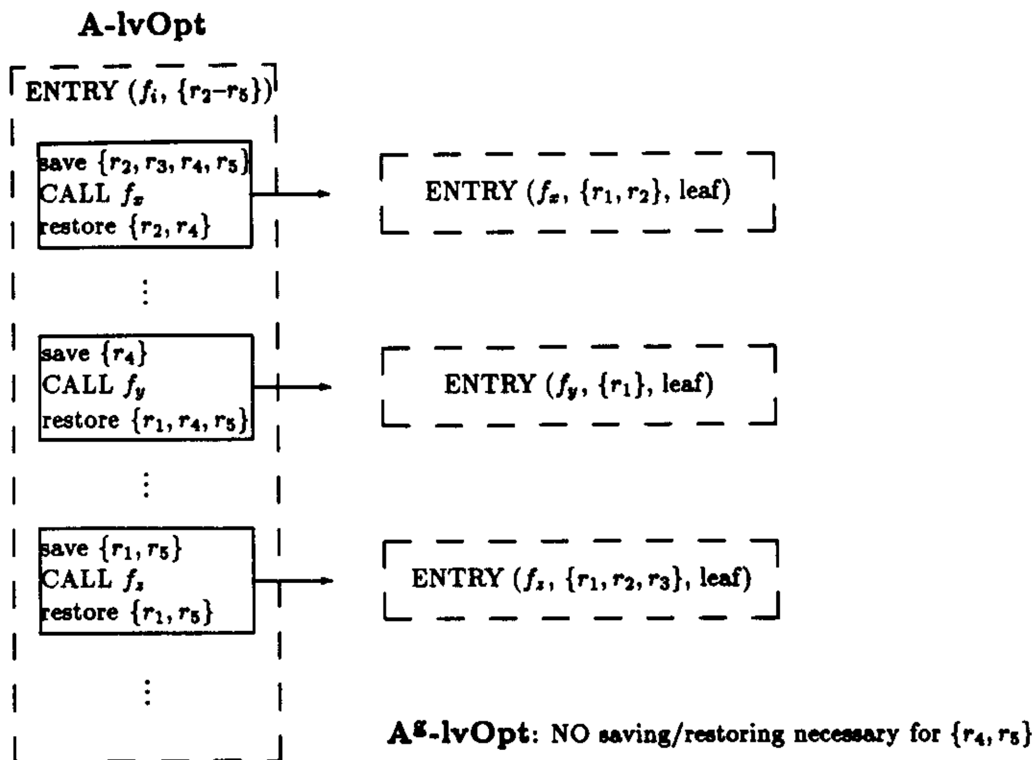


Figure 4.14: Policy A^s-lvOpt versus Policy A-lvOpt

The above problems could be solved if control-flow information was available to the interprocedural 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 interprocedural optimizer, except for the call graph (see step number 2 at the beginning of Section 4.2). The alternative of providing control-flow information to the interprocedural 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^s-lvOpt without control-flow information.

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 descendants of the function (see Figure 4.14). Algorithm 2 presents how register assignment is performed. We assume that cycles have been removed from the call graph as it has been described in the previous section.

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

Inputs: A acyclic call graph G, the set of defined functions (F), the sets of selected local scalar variables for allocation per function (V^i), the static or dynamic frequency of each call ($Q_{i \to j}$), and the registers to be saved ($S_{i \to j}$) and restored ($R_{i \to j}$) per each call[9].

[9]Notice that because of the way the compiler has assigned registers (see step 1 at the beginning of Section 4.2),

```
for each f_j ∈ F visited in reverse depth-first search order do
    if f_j is a pseudo-function then
        U^j ← {registers required by functions in cycle} ∪ ⋃_{∀t, j→t} U^t
    else if f_j is a leaf function then
        assign V^j variables to registers:
            a_{1j} ← v^j_1, a_{2j} ← v^j_2, ..., a_{mj} ← v^j_m
        U^j = {r_1, r_2, ..., r_m}
    else
        U^j ← (⋃_{∀t, j→t} U^t)
        for each v^j_k ∈ V^j do
            for each r_i ∈ R do
                if r_i ∈ U^j then
                    c_{ik} = Σ_{∀t, j→t} {Q_{j→t} MEM_REFS(r_i, S_{j→t}, R_{j→t})}
                else
                    c_{ik} = K_j
                fi
            od
        od
        Perform register assignment as given in [Hill67]
        Let variables v^j_1, v^j_2, ..., v^j_m be assigned to registers r_{i_1}, r_{i_2}, ..., r_{i_m}
        Update column j for matrix A:
            a_{i_1,j} ← v^j_1, ..., a_{i_m,j} ← v^j_m
        Eliminate unnecessary RSR:
            for each v^j_k ∈ V^j such that r_{i_k} ∉ U^j do
                for each f_t ∈ F such that j → t do
                    S_{j→t} ← S_{j→t} − {r_{i_k}}
                    R_{j→t} ← R_{j→t} − {r_{i_k}}
                od
            od
        Update U^j with registers used by f_j:
            U^j ← U^j ∪ {r_{i_1}, r_{i_2}, ..., r_{i_m}}
    fi
od
```

Figure 4.15: Register Assignment for Policy A^{s}-lvOpt

Outputs: A set of registers to be saved and/or restored per call ($S_{i \to j}$ and $R_{i \to j}$) and the matrix A that indicates the register assignment per function.

Locals: This algorithm uses the matrix C and the U^j sets as given in Algorithm 1. It also requires the following definition:

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

Method: Apply algorithm given in Figure 4.15. As for Algorithm 1, we assume that local scalar variables assigned to to-be-destroyed registers for leaf functions have already been removed from V^j. □

4.2.3 Global B-lf

The Global B Policy with Leaf Functions (**B^g-lf**) eliminates the register saving/restoring instructions performed on 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 and, afterwards, we present an algorithm to perform register assignment.

As before, our approach is based on the fact that functions that 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 interprocedural 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 such that its ancestors have already their registers assigned and assign registers for it. Let us illustrate the problems of performing an assignment by the example given in Figure 4.16.

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 3 possibilities:

1. 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.
2. Assign registers from the subset of registers not used by its callers. In this case, register saving/restoring is eliminated since the registers have not been used previously. Let us refer to these registers as *free* registers since the RSR generated is zero.
3. An intermediate approach on which we select registers from both subsets.

---

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 provisional assignment made by the compiler.

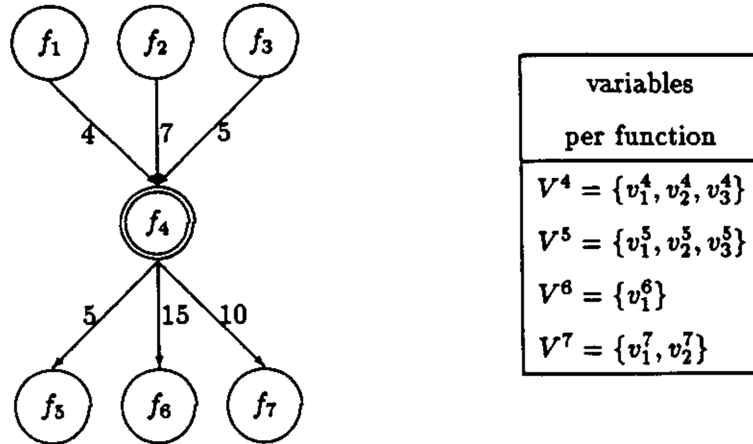


Figure 4.16: Example of Register Assignment for Policy B*-lf

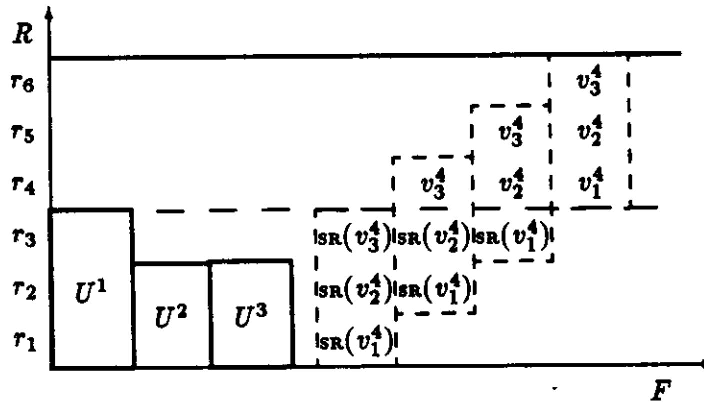


Figure 4.17: Alternative Register Assignments for Function f_4

These alternatives are reflected in Figure 4.17. The U^j sets now indicate the registers that 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 functions executed after f_4.
- The number of registers required by f_4.
- The number of registers required by the descendants of f_4.

For instance, if f_4 is executed only a few times, then the first alternative will be more attractive because more free registers will be left for the f_4 descendants and, therefore, more RSR traffic will be eliminated. On the other hand, if f_4 was a heavy used function, then the second would be. 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 descendants.

If we just consider the *immediate* descendants, 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

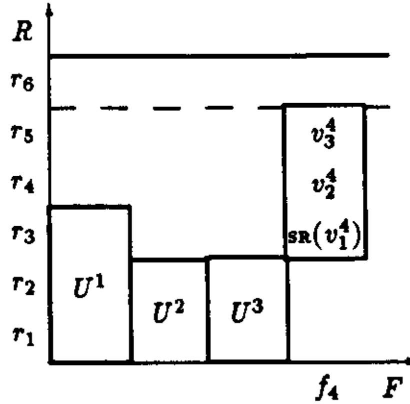


Figure 4.18: Optimal Variable-to-Register Assignment for Function f_4

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 one is left for the functions' descendants, $30 + 16 \times 2 = 62$ savings are eliminated; when only 1 are taken and 2 are left, $45 + 1 \times 16 = 61$ savings are eliminated; finally, when none is taken, 50 savings are eliminated. Thus, the optimal solution for this specific case is the one shown in Figure 4.18.

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 descendants. Thus, to preserve some free registers for the function's descendants we not only consider the savings to each immediate descendant, but also K savings as a fraction of the number of executed functions that will follow through each descendant. As for Policies A^S-live and A^S-lvOpt, we expect to try several values of K and report on them upon completion of this work.

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 Section 4.2.1. Moreover, the ER traffic can also be reduced using a similar approach to the one discussed for Policy A^S-live. In this case, if the register with the return address is not used by any of the function antecedents, it does not need to be saved/restored.

Algorithm 3 Register Assignment for Policy B^S-lf.

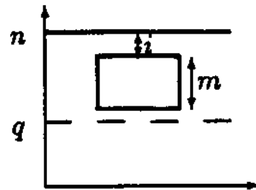
Inputs: An acyclic call graph G, the set of defined functions (F), the sets of selected variables for allocation per function (V^j), the static or dynamic frequency of each defined or executed function ($Y^j = \sum_{\forall t, \exists t \rightarrow j} Q_{t \rightarrow j}$), and a p-element array O with an acyclic numbering of the nodes such that in increasing order, O always contains the caller before the callee.

Outputs: The set of registers to be saved and restored per function (M^j) and the matrix A that indicates the register assignment per function.

Locals: A set U^j per function to indicate the registers defined by the function f_j and all its ancestors

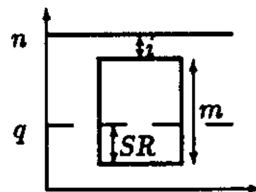
```
for f_j = O[p], O[p-1], ..., O[1] do
    if f_j is a leaf function then W^j ← 0
    else W^j ← Σ_{∀t, j→t}(W^t + Q_{j→t}) fi
od
for f_j = O[1], O[2], ..., O[p] do
    U^j ← ∪_{∀t, t→j} U^t
    q ←| U^j |
    if f_j is a pseudo-function then
        U^j ← U^j ∪ {registers required by functions in cycle}
    else if q = n then
        (* all registers already used by callers ⇒ no free registers *)
        Let variables v_1^j, v_2^j, ..., v_m^j be assigned to registers r_1, r_2, ..., r_m
        a_{1j} ← v_1^j, ..., a_{mj} ← v_m^j
        M^j ← {r_1, r_2, ..., r_m}
    else (* (n - q)-free registers available *)
        (* X[i] ≡ expected savings to be eliminated when exactly i registers
            are kept for descendants *)
        for i = 0, ..., n do X[i] ← 0 od
        for each f_t such that j → t do
            for i = 1, ..., | V^t | do X[i] ← X[i] + KW^t + Q_{j→t} od
            for i =| V^t | +1, ..., n do X[i] ← X[i] + KW^t od
        od
        (* X[i] ≡ expected savings to be eliminated when i regs. or less are kept *)
        for i = 1, ..., n do X[i] ← X[i-1] + X[i] od
        (* add saving to be eliminated for f_j when m variables have to be assigned,
             there are (n - q)-free registers,
             and i registers (or less) are kept for the descendants *)
        if m ≤ n - q then
            for i = 0, ..., m do
                if m + i ≤ n - q then X[i] ← X[i] + mY^j     (* see Fig. 4.20.(a) *)
                else X[i] ← X[i] + (n - q - i)Y^j fi         (* see Fig. 4.20.(b) *)
            od
        else
            for i = 0, ..., n - q do X[i] ← X[i] + (n - q - i)Y^j od (*see Fig. 4.20.(c)*)
        fi
        Find s such that X[s] ≥ X[i] ∀i, i ≠ s
        Update column j of A and compute M^j depending on
          the values of s, m, and q as shown in Figure 4.21
    fi
od
```

Figure 4.19: Register Assignment for Policy B^s-lf



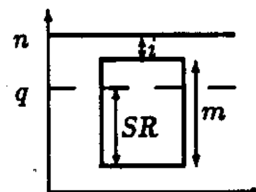
$$\left.\begin{array}{c} m \leq n - q \\ \textbf{and} \\ m + i \leq n - q \end{array}\right\} \Rightarrow mY^j$$

(a)



$$\left.\begin{array}{c} m \leq n - q \\ \textbf{and} \\ m + i > n - q \end{array}\right\} \Rightarrow (n - q - i)Y^j$$

(b)

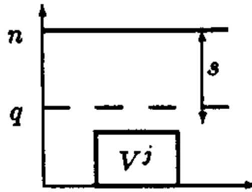


$$m > n - q \Rightarrow (n - q - i)Y^j$$

(c)

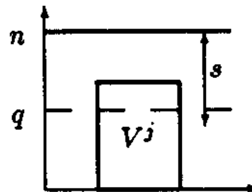
Figure 4.20: Savings to Be Eliminated for Several Values of i, q, and m

(I) **if** $s + q \geq n$ **and** $m \leq q$ **then**



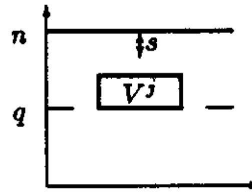
$a_{1j} \leftarrow v_1^j, \ldots, a_{mj} \leftarrow v_m^j$
$M^j \leftarrow \{r_1, \ldots, r_m\}$
U^j is already OK

(II) **if** $s + q \geq n$ **and** $m > q$ **then**



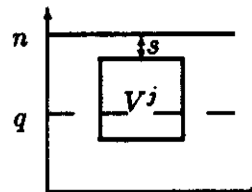
$a_{1j} \leftarrow v_1^j, \ldots, a_{mj} \leftarrow v_m^j$
$M^j \leftarrow \{r_1, \ldots, r_q\}$
$U^j \leftarrow U^j \cup \{r_1, \ldots, r_m\}$

(III) **if** $s + q < n$ **and** $m \leq n - q - s$ **then**



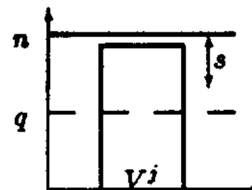
$a_{q+1,j} \leftarrow v_1^j, \ldots, a_{q+m,j} \leftarrow v_m^j$
$M^j \leftarrow \emptyset$
$U^j \leftarrow U^j \cup \{r_{q+1}, \ldots, r_{q+m}\}$

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



$a_{n-s-m+1,j} \leftarrow v_1^j, \ldots, a_{n-s,j} \leftarrow v_m^j$
$M^j \leftarrow \{r_{n-s-m+1}, \ldots, r_q\}$
$U^j \leftarrow U^j \cup \{r_{n-s-m+1}, \ldots, r_{n-s}\}$

(V) **if** $s + q < n$ **and** $m > n - s$ **then**



$a_{1j} \leftarrow v_1^j, \ldots, a_{mj} \leftarrow v_m^j$
$M^j \leftarrow \{r_1, \ldots, r_q\}$
$U^j \leftarrow U^j \cup \{r_1, \ldots, r_m\}$

Figure 4.21: Alternative Register Assignments Depending on s, q, and m

in the call graph; a set W^j per function that gives the number of calls to all the function's descendants; and an array $X = \{X[0], X[1], \ldots, X[i], \ldots, X[n]\}$ used to estimate the savings that could be eliminated when i registers are left free for the function's descendants.

Method: Apply algorithm given in Figure 4.19. As for Algorithm 1, we assume that local scalar variables assigned to to-be-destroyed registers for leaf functions have already been removed from V^j. □

4.2.4 Global G-lf

The Global Policy G with Leaf Functions (renamed **G^s-lf**) differs from the previous interprocedural optimizations in that no RSR instruction has to be eliminated. This is 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. 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 which this one might call. In this case, the probability of having to save a register is reduced. Moreover, if call and return instructions are provided such that the return address can be saved in any of the general-purpose registers as discussed in Section 4.1.1, a disjoint register assignment can be made for the registers with the return address so that the ER traffic will also be reduced.

In this section we discuss first the benefits that are obtained from disjoint register assignment among caller-callee pairs; second, we present the register assignment algorithm; and finally, we mention the limitations of the static interprocedural allocators which do not apply to the dynamic one.

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. Therefore, we expect that the RSR traffic for a large register set will be insignificant.

We now present the algorithm to perform register assignment. If we compare this algorithm with the previous algorithms for Policies A^s-live, A^s-lvOpt, and B^s-lf, we can observe its simplicity. Also, notice that the algorithm only tries to have disjoint register assignment for the caller-callee pairs. It does not optimize the register assignment for more than 2-consecutively-called functions. We expect that the RSR traffic reduction obtained will be satisfactory. Otherwise, we might have to look for a more complex algorithm.

```
for $f_j = O[1], O[2], \ldots, O[p]$ do
    $U^j \leftarrow \bigcup_{\forall t, t \to j} M^t$
    $q \leftarrow | U^j |$
    $m \leftarrow | V^j |$
    if $m \leq n - q$ then
        (* at least $m$ registers not used by immediate predecessors *)
        Let $v_1^j, v_2^j, \ldots, v_m^j$ be assigned to $r_{i_1}, r_{i_2}, \ldots, r_{i_m}$ such that $r_{i_k} \notin U^j$
        $a_{i_1,j} \leftarrow v_1^j, \ldots, a_{i_m,j} \leftarrow v_m^j$
        $M^j \leftarrow \{r_{i_1}, r_{i_2}, \ldots, r_{i_m}\}$
    else (* $(n - q)$ registers not used by mmediate predecessors *)
        for $i = 1, \ldots, n$ do
            $X[i] \leftarrow \sum_{\forall t, t \to j}$ REG_USED$(r_i, M^t)\, Q_{t \to j}$
        od
        Let $v_1^j, \ldots, v_{n-q}^j$ be assigned to $r_{i_1}, \ldots, r_{i_{n-q}}$ such that $\forall k = 1, \ldots, n - q$, $r_{i_k} \notin U^j$
          and $v_{n-q+1}^j, \ldots, v_m^j$ be assigned to $r_{i_{n-q+1}}, \ldots, r_{i_{m-n+q}}$ such that
          $\forall s \neq 1, \ldots, n - q, n - q + 1, \ldots, m - n + q, X[i_{n-q+1}] \leq \ldots \leq X[i_{m-n+q}] \leq X[i_s]$
        $a_{i_1,j} \leftarrow v_1^j, \ldots, a_{i_{m-n+q},j} \leftarrow v_{m-n+q}^j$
        $M^j \leftarrow \{r_{i_1}, \ldots, r_{i_{m-n+q}}\}$
    fi
od
```

Figure 4.22: Register Assignment for Policy G^{s}-lf

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 static or dynamic frequency of each call ($Q_{i \to j}$), 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 call graph G may have cycles. The recursive functions are not removed from the graph as it was done for the previous algorithms. 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.

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

Method: Apply algorithm given in Figure 4.19. As for Algorithm 1, we assume that local scalar

variables assigned to to-be-destroyed registers for leaf functions have already been removed from V^j. □

The RSR traffic reduction that the static interprocedural allocators might obtain is closely related to the program structure. This is not the case for the dynamic Policy G^g-lf. Let us discuss how this happens for Policies A^g-live and A^g-lvOpt. Analogous reasons could be given for Policy B^g-lf.

The number of register saving/restoring instructions which can be eliminated for a specific function f_j depends on the number of registers that are not used for the functions that might be called. If these functions already have all the registers assigned (because, for instance, one of them needs all of them[10]), 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 any RSR instruction for these functions. Therefore, the RSR traffic reduction depends on the program structure.

4.2.5 Summary

Subsection 4.2 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. We expect that the complexity added to the compilation process would be justified by the benefits obtained.

Three of these algorithms are for the static policies. The 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. The Policy B^g-lf eliminates the register saving/restoring instructions performed on 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 look for a disjoint register assignment such that the maximum number of register saving/restoring instructions can be eliminated.

We expect to show that for larger register sets, these optimizations will actually reduce the RSR traffic. This is not the case when no compiler optimizations are performed or when they are only performed at function level (see Sections 3.4 and 4.1.5).

One additional advantage of performing interprocedural optimization is that depending on the program characteristics, the most appropriate RSR policy can be selected (either Policy B^g-lf or the best between Policy A^g-live and A^g-lvOpt). In Section 4.1.5 we mentioned that depending on the program characteristics Policy A-lvOpt was generating more RSR traffic than Policy B-lf. Thus, the optimizer (before starting to perform the global optimization) can estimate the RSR traffic generated by each static policy because the frequency of each call and the number of variables

[10] Notice 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.

selected for allocation per function are known. In this case, it can select the policy that generates the least RSR traffic and apply the corresponding interprocedural optimization.

The fourth algorithm is for the dynamic Policy G. Policy G^s-lf assigns disjoint registers for the functions that may call each other, 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. We expect to show that Policy G^s-lf generates less RSR traffic than any of the static optimizations as it has been the case for the measurements that we have obtained so far. Moreover, Policy G^s-lf has three more advantages with respect to the static policies:

1. It eliminates RSR traffic in both directions: from the root to the leaf functions when the registers have not been previous used and from the leaf functions to the root when the registers are not defined in the functions being called. The static optimizations only eliminate RSR traffic in one direction: from the root to the leaf functions for Policy B^s-lf and vice versa for Policies A^s-live and A^s-lvOpt.

2. It does not depend on the program structure as much as the static policies do. For instance, even when one of the functions (f_j) in the program requires all the registers available, it is unlikely that all of them are written between the calls that f_j generates. Thus, the descendants of f_j will find some registers that have not been used in the exterior levels and, therefore, they will not need to be saved/restored.

3. It simplifies the implementation of the compiler.

Chapter 5

Conclusions

In this research proposal, we have discussed the support that the architecture can provide to implement efficient function calls for single-window architectures and the support that can be provided by the compiler. In this chapter, we comment on the most important contributions obtained or expected.

We have presented six new architectural policies to reduce the register saving and restoring (RSR) 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% (when 24 to-be-preserved registers are available to the register allocator) and 31% (for 6) of the 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.

Moreover, when the register set size is increased, the static Policies A and B generate more RSR traffic. However, this is not the case for the dynamic Policy G. When there are 32 to-be-preserved registers, the RSR traffic generated by Policy G is 54% of the traffic generated when there are 6 while for the Policies A and B, the RSR traffic increases to 176% and 124%, respectively. Consequently, the overall data memory traffic might not be reduced for a larger register set when the conventional static RSR policies are used.

We have also introduced six new compiler optimizations to reduce the RSR overhead. Two of these optimizations are *intraprocedural* and based on live-variable analysis: Policy A-lvOpt and Policy G-live. We have evaluated the RSR traffic generated by Policy A-lvOpt and compared it with the traffic generated by two already-existing compiler optimizations: the standard Policy A with live-variable analysis (A-live) and the Policies B and G with leaf functions (B-lf and G-lf). Policy A-lvOpt has between 59% and 64% of the RSR traffic generated by the standard A-live and between 53% and 61% of Policy B-lf. The optimized dynamic Policy G-lf generates the least traffic: it has between 22% and 47% of the RSR traffic generated by Policy A-lvOpt. No measurements are available for Policy G-live at the moment of writing this proposal. However, since this optimization requires the addition of a machine instruction per function call, we do not expect that the RSR

traffic reduction can justify the increase in instruction memory traffic. Measurements will be taken to justify or deny this hypothesis.

Four of the optimizations are *interprocedural*: A^g-live, A^g-lvOpt, B^g-lf, and G^g-lf. These optimizations require the call graph of the program and their goal is to find a disjoint *register assignment* for the variables selected for allocation (in a previous phase). No measurements are available at the moment of writing this proposal for these optimizations. We expect that Policy G^g-lf will be generating less traffic than the static ones as it has always been the case for the previous measurements. However, Policy G will not be any more the only one that might benefit from a larger register set. We expect that for larger register sets, the interprocedural optimized static policies will get a reduction in their RSR traffic. This was not the case without interprocedural optimization. Moreover, we also expect to show that the benefits that we can obtain from the interprocedural optimized static policies are closely related to the program structure while this might not be the case for Policy G^g-lf.

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 this type of measurements. BKGEN obtains a version of the program being measured that is directly executable on the existing machine (EM) and associates with each block of PM (proposed machine) 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 (compilers, word processors, assemblers, ...) have been measured to obtain the above mentioned results.

We have also commented on two more research topics that we plan to complete for this work: register allocation and the implementation of Policy G in a RISC II-like processor. Since the RSR traffic has always been considered the main drawback to allocate variables to registers, we want to evaluate several *register allocation* schemes and compare the data memory traffic reduction obtained by these schemes when they are used in conjunction with the static and dynamic policies and with the compiler optimizations. We expect to show that to be able to use efficiently a large register set (more than 16 registers), the system has to provide some architectural support (i.e., Policy G for single-window architectures) and/or the interprocedural optimizations. Moreover, we also expect to evaluate the overall data traffic generated by three compilers for VAX-11 (PCC, ACK, and GNU).

We have also to consider the *implementation of Policy G* because it could be the case that although the total number of data memory references becomes smaller, the total execution time to execute a program is larger. This could be a consequence of an increase either in the processor cycle time (because of the extra hardware required for Policy G) or in the number of processor cycles necessary to perform the saving/restoring operations associated to Policy G. We expect to show that (1) Policy G does not have any negative effect on the processor cycle time (as multiple-window architectures do) and (2) although more cycles are required to perform each restoring operation, the total number of cycles has also been reduced due to the reduction in the number of restoring operations to be performed.

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. Published in *Computer Architecture News*, vol. 15, no. 5.

[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. Published in *SIGPLAN Notices*, vol. 17, no. 6.

[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 *Proc. Symp. on Architectural Support for Programming Languages and Operating Systems*, March 1982, pp. 30–38. Published in *Computer Architecture News*, vol. 10, no. 2.

[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.

[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. Cherenson, 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. Published in *Computer Architecture News*, vol. 15, no. 5.

[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. Published in *Computer Architecture News*, vol. 15, no. 5.

[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. Published in *SIGPLAN Notices*, vol. 21, no. 7.

[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. Published in *SIGPLAN Notices*, vol. 17, no. 6.

[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. Published in *SIGPLAN Notices*, vol. 19, no. 6.

[Chow86] F. Chow, M. Himelstein, E. Killian, and L. Weber, "Engineering a RISC Compiler System," in *COMPCON '86*, 1986, pp. 132–137.

[Chow87a] F. Chow, S. Correll, M. Himelstein, 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)*, Palo Alto, California, 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. Published in *Computer Architecture News*, vol. 15, no. 2.

[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 *Proc. 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 Recompilation," in *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, June 1986, pp. 58–67. Published in *SIGPLAN Notices*, vol. 21, no. 7.

[Cort87a] J. Cortadella, *Mecanismos para la Ejecución Eficiente de los Saltos en Arquitecturas RISC*, PhD dissertation, Facultat d'Informática, 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," to be published 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. Published in *Computer Architecture News*, vol. 15, no. 5.

[Day70] W.H.E. Day, "Compiler Assignment of Data Items to Registers," *IBM Systemss 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 *Proc. Symp. on Architectural Support for Programming Languages and Operating Systems*, March 1982, pp. 48–56. Published in *Computer Architecture News*, vol. 10, no. 2.

[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. Published in *Computer Architecture News*, vol. 15, no. 2.

[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. Published in *Computer Architecture News*, vol. 15, no. 5.

[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. Published in *Computer Architecture News*, vol. 15, no. 2.

[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. Published in *Computer Architecture News*, vol. 15, no. 2.

[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. Published in *SIGPLAN Notices*, vol. 19, no. 6.

[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.

[Flyn87] 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.

[Folg83] 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 *Proc. 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.

[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. Published in *Computer Architecture News*, vol. 13, no. 3.

[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.

[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. Published in *Computer Architecture News*, vol. 13, no. 3.

[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. Published in *SIGPLAN Notices*, vol. 22, no. 7.

[John73] R.K. Johnsson, *A Survey of Register Allocation*, Technical Report, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA 15213, May 1973.

[John75] R.K. Johnsson, *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. Johnsson and J. Wick, "An Overview of the Mesa Processor Architecture," in *Proc. Symp. on Architectural Support for Programming Languages and Operating Systems*, March 1982, pp. 20–29. Published in *Computer Architecture News*, vol. 10, no. 2.

[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. Published in *SIGPLAN Notices*, vol. 21, no. 7.

[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. Kraley, 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 *Proc. Symp. on Architectural Support for Programming Languages and Operating Systems*, March 1982, pp. 66–76. Published in *Computer Architecture News*, vol. 10, no. 2.

[Lang85] T. Lang and M. Huguet, *Reduced Register Saving/Restoring in Single-Window Register Files*, Technical Report CSD-850040, UCLA Computer Science Department, Los Angeles, CA 90024, December 1985.

[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. Published in *SIGPLAN Notices*, vol. 21, no. 7.

[Leve83] B.W. Leverett, *Register Allocation in Optimizing Compilers*, UMI Research Press, 1983. Ph. D. Thesis.

[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. Published in *SIGPLAN Notices*, vol. 19, no. 6.

[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. Published in *Computer Architecture News*, vol. 15, no. 5.

[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. Published in *SIGPLAN Notices*, vol. 22, no. 7.

[McDa82] G. McDaniel, "An Analysis of a Mesa Instruction Set Using Dynamic Instruction Frequencies," in *Proc. Symp. on Architectural Support for Programming Languages and Operating Systems*, March 1982, pp. 167–176. Published in *Computer Architecture News*, vol. 10, no. 2.

[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. Published in *Computer Architecture News*, vol. 11, no. 3.

[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. Published in *Computer Architecture News*, vol. 10, no. 3.

[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. Published in *Computer Architecture News*, vol. 11, no. 3.

[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. Published in *SIGPLAN Notices*, vol. 14, no. 8.

[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 Califonia, 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. Published in *SIGPLAN Notices*, vol. 19, no. 6.

[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 90x*, Pyramid Technology Corporation, San Diego, CA 94039, September 1983.

[Radi82] G. Radin, "The 801 Minicomputer," in *Proc. Symp. on Architectural Support for Programming Languages and Operating Systems*, March 1982, pp. 39–47. Published in *Computer Architecture News*, vol. 10, no. 2.

[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.

[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.

[Séqu82] 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. Published in *SIGPLAN Notices*, vol. 21, no. 11.

[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, pp. 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. Published in *SIGPLAN Notices*, vol. 14, no. 8.

[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. Published in *Computer Architecture News*, vol. 15, no. 2.

[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 *Proc. Symp. on Architectural Support for Programming Languages and Operating Systems*, March 1982, pp. 158–166. Published in *Computer Architecture News*, vol. 10, no. 2.

[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. Published in *Computer Architecture News*, vol. 14, no. 2.

[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. Published in *Computer Architecture News*, vol. 12, no. 3.

[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. Published in *SIGPLAN Notices*, vol. 21, no. 7.

[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. Published in *Computer Architecture News*, vol. 15, no. 5.

[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 *Proc. Symp. on Architectural Support for Programming Languages and Operating Systems*, March 1982, pp. 177–184. Published in *Computer Architecture News*, vol. 10, no. 2.

[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.

[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. Published in *Computer Architecture News*, vol. 14, no. 2.

[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.