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

REDUCED REGISTER SAVING/RESTORING FOR SMALL REGISTER FILES

Miquel Huguet
Tomas Lang

August 1988
CSD-880066

Reduced Register Saving/Restoring for Small Register Files

Miquel Huguet* and Tomás Lang

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

Abstract

With the current trend towards VLSI load/store architectures, registers have become one of the critical resources to increase processor performance. Since registers are conventionally saved/restored across function calls, the corresponding saving and restoring (RSR) traffic can almost eliminate the data memory traffic reduction obtained. To reduce the RSR overhead some current processors rely on hardware support, such as multiple-window register files, intra-procedural compiler optimizations, such as live-variable analysis and leaf-function optimization, and/or inter-procedural register allocation. In this paper we present an intra-procedural optimization to reduce the RSR traffic when registers are saved/restored at the caller side (Policy A-lvOpt). We show that this policy generates the least RSR traffic with respect to the conventional policies of saving/restoring the registers at the caller with live-variable analysis and at the callee with leaf-function optimization. Moreover, we present a new RSR policy which makes use of dynamic information to know which registers have been used during program execution so that unnecessary RSR traffic is eliminated. We also show that our dynamic policy with leaf-function optimization (Policy G-lf) generates less RSR traffic than Policy A-lvOpt. Finally, we compare the RSR traffic generated by Policy G-lf to the one generated by the already-existing schemes for multiple-window architectures (fixed-size windows, variable-size windows, and multi-size windows) and show that Policy G-lf generates the least RSR traffic when a small register file (32 registers) is used.

1 Introduction

With the current trend towards VLSI load/store architectures, registers have become one of the critical resources to increase processor performance [Patt82,Radi82,Henn84,Patt85a]. Recent advances in compiler technology, especially in *register allocation* optimization [Chai82,Chow83,Wall86, Stee87], have resulted in better use of the register set because the compiler can allocate more *scalar* variables to registers. The use of registers reduces both the data memory traffic and the instruction memory traffic. However, since registers are conventionally saved/restored across function calls, the corresponding saving and restoring (RSR) traffic can almost eliminate the overall reduction. To reduce the register saving and restoring overhead some current processors rely on

*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.

hardware support, such as multiple-window register files [Patt82,Kate83,Unga84,Atki87,Ditz87b], intra-procedural compiler optimizations [Radi82,Chow84], and/or inter-procedural register allocation [Wall86,Stee87,Chow88].

Although multiple-window architectures produce a large reduction in the RSR traffic [Patt82, Hugu85b,Flyn87], they have three main drawbacks: they use a large chip area in a VLSI implementation or a large number of chips in a MSI/LSI implementation, they increase the amount of processor context to be saved on context switching, and they increase the processor cycle-time due to the long data busses [Henn84].

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], leaf-function optimization (as used in the compilers for MIPS [Chow86] and for HP-Spectrum [Cout86]), and inline expansion or procedure integration [Sche77]. In this case, the *conventional* register saving/restoring policies used by the register allocator are: to save/restore registers at the caller (named **Policy A**), to save/restore them at the callee (named **Policy B**), or a combination of both. These RSR policies are *static* because they save and restore the registers defined in the function (indicated by a *mask* generated by the compiler) independently of the register usage during program execution.

Another approach is to perform register allocation once the *call graph* of the program is known (inter-procedural). Wall [Wall86] eliminates the RSR traffic by allocating variables at link time and allocating only those that fit in the 52 registers available. Steenkiste [Stee87] and Chow [Chow88] still perform intra-procedural allocation and eliminate the RSR instructions for the functions whose descendents in the call graph do not use the registers that are being saved/restored.

We have proposed six architectural policies to reduce the traffic generated by the conventional Policies A and B for a single-window architecture with an intra-procedural register allocator. These new policies make use of *dynamic* information to know which registers have been used during program execution so that unnecessary RSR traffic is eliminated. The drawback with five of these policies is that they must save/restore a mask per function call, which increases the overall traffic. However, this is not the case for the sixth one, called **Policy G**. Thus, only Policy G is described in detail in this paper. The reader is referred to [Lang86,Hugu88] for a detailed description of the other policies.

In this paper we present an intra-procedural optimization to reduce the RSR traffic for Policy A (**Policy A-lvOpt**) and compare it with the standard Policies A with live-variable analysis (**Policy A-live**) and B with leaf-function optimization (**Policy B-lf**). We show that when 24 *to-be-preserved* registers[1]) are available to the register allocator, Policy A-lvOpt generates 60% of the RSR traffic produced by Policy A-live and 57% of Policy B-lf. (The allocation policy used by the compiler and the programs measured are described in Sections 1.1 and 1.2.) We also show that our dynamic Policy G with leaf-function optimization (**Policy G-lf**) generates the least RSR traffic with respect to the optimized static policies. For instance, our measurements show that Policy G-lf generates 22% of the RSR traffic produced by Policy A-lvOpt (for 24 to-be-preserved registers).

We compare the RSR traffic generated by our best optimized dynamic Policy G to the one generated by the already-existing schemes for multiple-window architectures: fixed-size windows

[1]These registers correspond to the registers that must be preserved across function calls. As a contrast, registers for temporary values that can be destroyed across function calls are called *to-be-destroyed* registers.

[Kate83], variable-size windows [Ditz82], and multi-size windows [Hugu85b]. Since the goal is to have a small register file (32 registers) to avoid an increase in the processor cycle time, we show that Policy G-lf generates the least RSR traffic. Moreover, we also sketch the implementation of Policy G in a RISC-like processor to show that its implementation does not affect the processor cycle time because the operations required are performed in parallel with the main CPU activities.

Finally, we estimate the speed-up factor for a RISC-like processor with Policy G-lf with respect to Policy A-lvOpt. Our measurements show a speed up of 4.5% when load/store instructions execute in 2 processor cycles and 10.5% when 3.

1.1 Register Allocation and Assignment

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. Thus, we have selected a register allocation scheme based on the register allocation performed by the Portable C Compiler (PCC) [John79]. This scheme has already been used to evaluate several register saving/restoring schemes by some other authors [Patt82,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. However, this scheme is useful to evaluate the RSR traffic reduction caused by each policy because it implies a heavy register usage.

The PCC standard intra-procedural register allocation only allocates to registers integer and pointer *register variables* [Kern78] explicitly defined by the programmer. 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 1.9 registers assigned per executed function when there are 6 to-be-preserved registers. This number only increases to 2.0 when the number of to-be-preserved registers is 32. With this allocation, 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[2]. The scalar variables considered for allocation not only include integer and pointer variables (as PCC does), but also characters and short integers. The four programs measured (given below) 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 intra-procedural register assignment policy has also been modified. PCC always assigns registers in the same order for each function. Thus, some registers are assigned more frequently than others. Since the dynamic policies require that the intersection of registers defined by the caller and by the callee be as small as 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. However, round-robin register assignment does not guarantee that a caller and a callee have disjoint registers, but it increases the probability for doing so. We are planning to use inter-procedural register assignment to obtain disjoint registers [Hugu88], but this is not considered in this paper.

Although the measurements have only been taken using this register allocation policy, we expect that our conclusions are valid for any intra-procedural allocation policy. We have already taken

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

some measurements using the Amsterdam C Compiler [Tane83] (which uses a register allocation by priority) and have obtained similar results. We are also planning to evaluate the RSR traffic with coloring.

1.2 BKGEN and the Programs Measured

To perform the above mentioned evaluations a new tool has been designed: a *Block-and-Actions Generator* (BKGEN) as described in [Hugu87]. BKGEN reduces drastically the overhead introduced by a conventional simulator (the tool traditionally used to obtain these types of measurements) and makes it possible to measure *large typical* programs (compilers, word processors, assemblers, ...).

We have measured four programs for a UNIX environment: the UNIX VAX-11 assembler (ASM), the nroff word processor (NROFF), the UNIX sort program (SORT), and the Portable C Compiler (VPCC). These programs have almost 900 functions defined and execute more than 700,000 function calls.

We have selected these large programs rather than *small specific programs* (such as the Ackermann's function, the Hanoi Towers, the Erastothenes Sieve, the puzzle program, ...) or *synthetic benchmarks* (such as the CFA benchmarks, Whetstone, Dhrystone, ...) because the first type of programs is 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. Moreover, our first measurements on several of these programs show that the results obtained from benchmarks might not be representative of a real system behavior [Hugu85a]. Some other authors [Levy82,Chow83,Colw85,Patt85b,Hitc86,Laru86,Wong88,Wall88] have also pointed to the limitations of these benchmarks to evaluate certain architecture features.

The above mentioned policies are compared using an *average RSR traffic* per function call for the four programs. This average has been computed as the total RSR traffic for all programs divided by the number of function calls. The reader interested in obtaining the information per program is referred to [Hugu88]. Only the RSR traffic is considered 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. The RSR traffic does not include the traffic generated by the *environment registers*[3] (ER). This is because the ER traffic is *fixed* for each function call and, therefore, some special hardware (multiple PCs) or specific intra- or inter-compiler optimizations can be performed to reduce it [Hugu88].

2 Intra-Procedural Optimizations for the Static Policies

This section presents a new compiler optimization to reduce the register saving and restoring traffic for Policy A and compares it with two existing optimizations: live-variable analysis [Hech77,Aho86]

[3]The ER traffic is caused by the saving/restoring of the *program counter* (i.e., the return address). We assume that no traffic is generated for other environment registers (*frame pointer* and/or *argument pointer*) because the activation record size is fixed during function execution so that locals and parameters are referred with respect to the *stack pointer* [Wulf75].

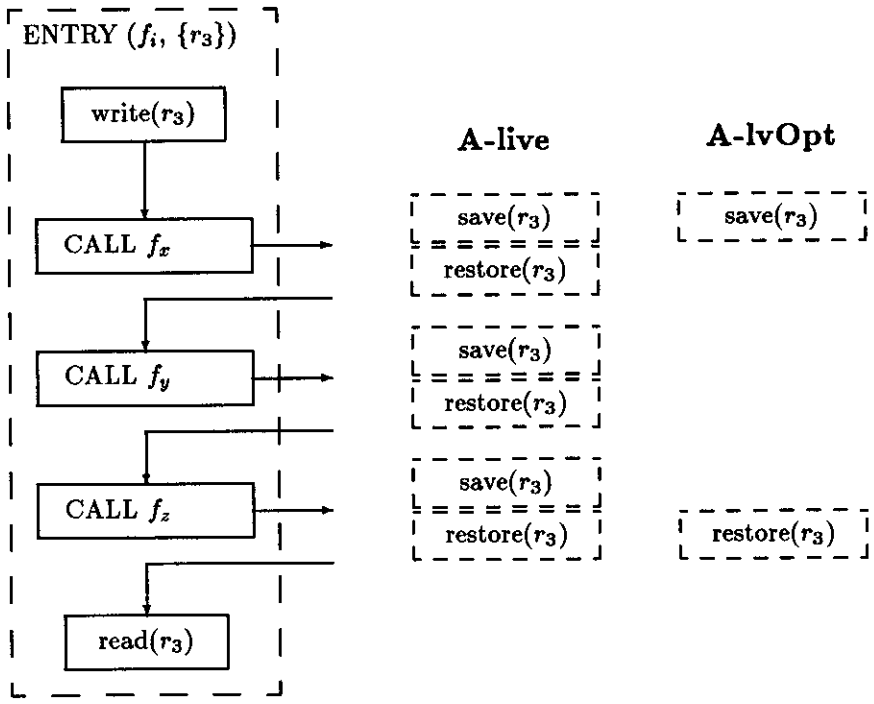

Figure 1: Policy A-lvOpt versus Policy A-live

for Policy A (renamed **A-live**) and register assignment in leaf functions for Policy B (renamed **B-lf**).

Policy A-live is the standard intra-procedural register allocation for Policy A implemented by an optimizing compiler when variables are assigned to registers for the whole scope of the function [Aho86]. While the unoptimized Policy A saves and restores all the registers defined by the function, Policy A-live only saves and restores a subset of them: the ones 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*. 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. Our measurements show that Policy A-live generates from 38% (for 32 registers) to 53% (for 6) of the RSR traffic produced by the unoptimized Policy A.

Policy A-live generates some unnecessary RSR traffic. For instance, let us consider the situation shown in Figure 1. The function currently being parsed has three *consecutive calls*, that is, three calls without unconditional or conditional branches between them. Register r_3 is written before the first call and it is read after the third one. Since it is alive at each call, Policy A-live saves/restores it in every consecutive call. However, it is only necessary to save it at the first call and to restore it upon return from the third one. This is done by Policy A-live Optimized (renamed **A-lvOpt**). This optimization can be applied not only to the consecutive calls, but also to the non-consecutive calls obtained following all possible paths.

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

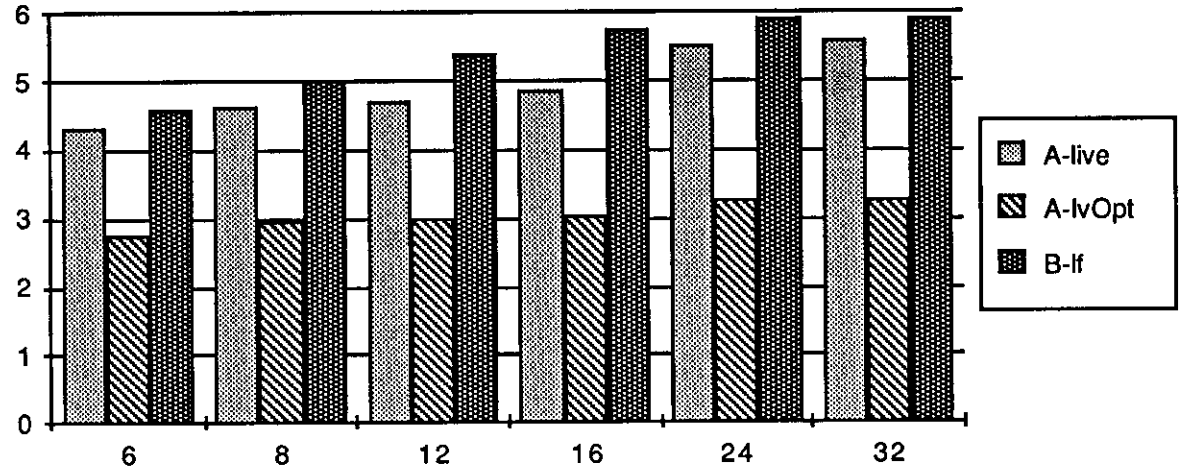

Figure 2: RSR Traffic for the Static Policies

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. For this reason, we 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[4], this additional requirement does not increase the compiler complexity.

Figure 2 shows the RSR traffic for Policies A-live and A-lvOpt. On the average, Policy A-lvOpt generates between 59% and 64% of the RSR traffic produced by the standard Policy A-live.

On the other hand, an optimizing compiler which saves/restores the registers at the callee can reduce the register saving/restoring traffic during function calls by changing the assignment of variables to registers in *leaf functions* [Chow86,Cout86]. A function is said to be a leaf when it does not have any call instruction in its definition body. Since these functions will not 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[5], no RSR traffic is generated.

[4]This is necessary to use the stack pointer as frame pointer and argument pointer (see Section 1.2).

[5]The number of to-be-destroyed registers required is small. We have measured the RSR traffic reduction for both 4 and 6 registers and the results obtained are similar [Hugu88]. For this reason, in this paper we only present the results for 4.

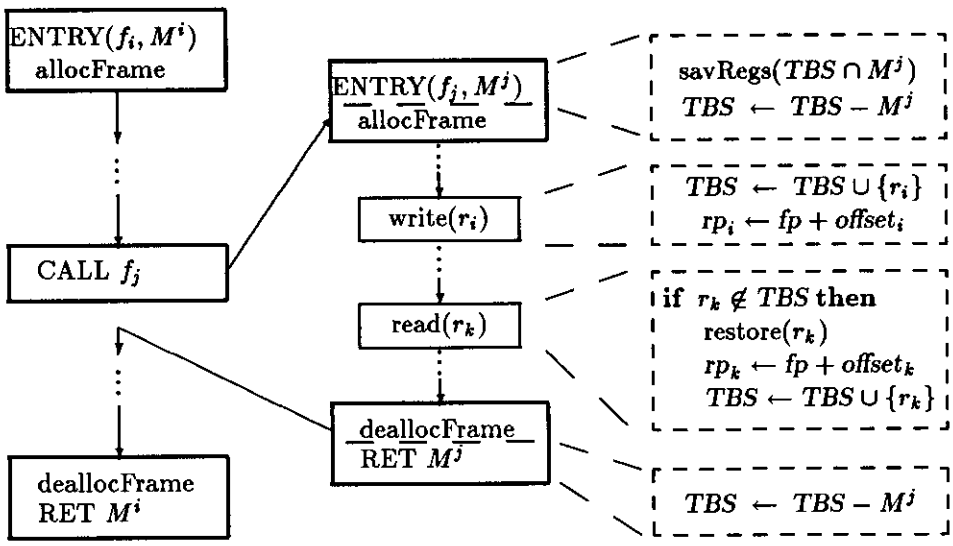

Figure 3: Policy G

Figure 2 compares Policy B-lf with the Policies A-live and A-lvOpt. As we can see, it is better to save/restore registers at the caller rather than at the callee. Policy A-live generates between 85% and 95% of the RSR traffic produced by Policy B-lf and Policy A-lvOpt between 54% and 62%. Since Policy A-lvOpt is the static policy which generates the least RSR traffic, this is the one that we will use as a reference to evaluate the RSR traffic reduction obtained by Policy G.

3 Architectural Support: Policy G

In this section we present the operations performed (and the masks required) by Policy G, we discuss the architectural support necessary to implement them, and we evaluate the RSR traffic reduction obtained when we apply to Policy G the standard leaf-function optimization (renamed G-lf).

3.1 A Description of Policy G

Policy G as well as Policy B perform the register saving/restoring upon function entry. However, while Policy B saves the whole set of registers defined in the function[6], Policy G only saves the registers that have been used (i.e., written) by the functions in the *exterior levels* (i.e., by one of the functions which is currently active). The registers used at the exterior levels are indicated by a global mask ($TBS \equiv$ To Be Saved).

When a register is saved, the bit associated to this register in the TBS mask is marked *unused* so that this register 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 (how to determine the address where the register

[6]The registers defined for each function are indicated by a mask M. This mask is given in function entry and duplicated with each return instruction so that it does not have to be saved across function calls.

has to be saved is discuss below). In this manner, the register does not have to be restored when the function returns (as Policy B does), but only when the function using this register 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). Notice that the register is not restored if the first operation to be performed after return is a write.

To save a register in the activation record of the outer function which last used it, 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 4 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.

3.2 Implementation of Policy G

It could be the case that although the total number of data memory references becomes smaller, the total time to execute the program is larger because either the processor cycle time has increased as a consequence of the extra hardware required for Policy G or more processor cycles are required to execute the operations associated to Policy G. Thus, it is important to consider the implementation of Policy G to show that it has no negative effects.

We are currently studying the implementation for our dynamic policy in a RISC-like processor and we expect to show that the number of cycles required to perform the register saving/restoring operations are equivalent to the ones required by the conventional static policies and that there are no negative effects on the processor cycle time. This is because the operations required are performed in parallel with the main CPU activities. Here we sketch a possible implementation of this scheme to determine its feasibility. Of course, alternative implementations are possible and the most suitable one would depend on many design requirements and constraints.

For our RISC-like processor we assume that the registers to be saved are indicated by a mask M given by the instruction **savRegs** generated by the compiler upon function entry. This mask is duplicated in the return instruction. Similar to RISC II, instructions have 3-register addresses and the processor is implemented with a 3-stage pipeline [Kate83].

Figure 5 shows the data section of the processor. We have encircled those components that have to be added due to the register saving/restoring policy. These are:

1. A set of registers *RP* to contain the pointers. One pointer is required for each to-be-preserved register. Since these registers are addressed together with the main registers, it is possible to share one of the address decoders among both sets.

2. A register to store the *TBS* mask.

3. An adder to compute the address of the memory location where the particular register is saved in (or restored from) the activation record. This address is obtained as the sum of the frame pointer (stored in the specialized register *fp*) and the register number.

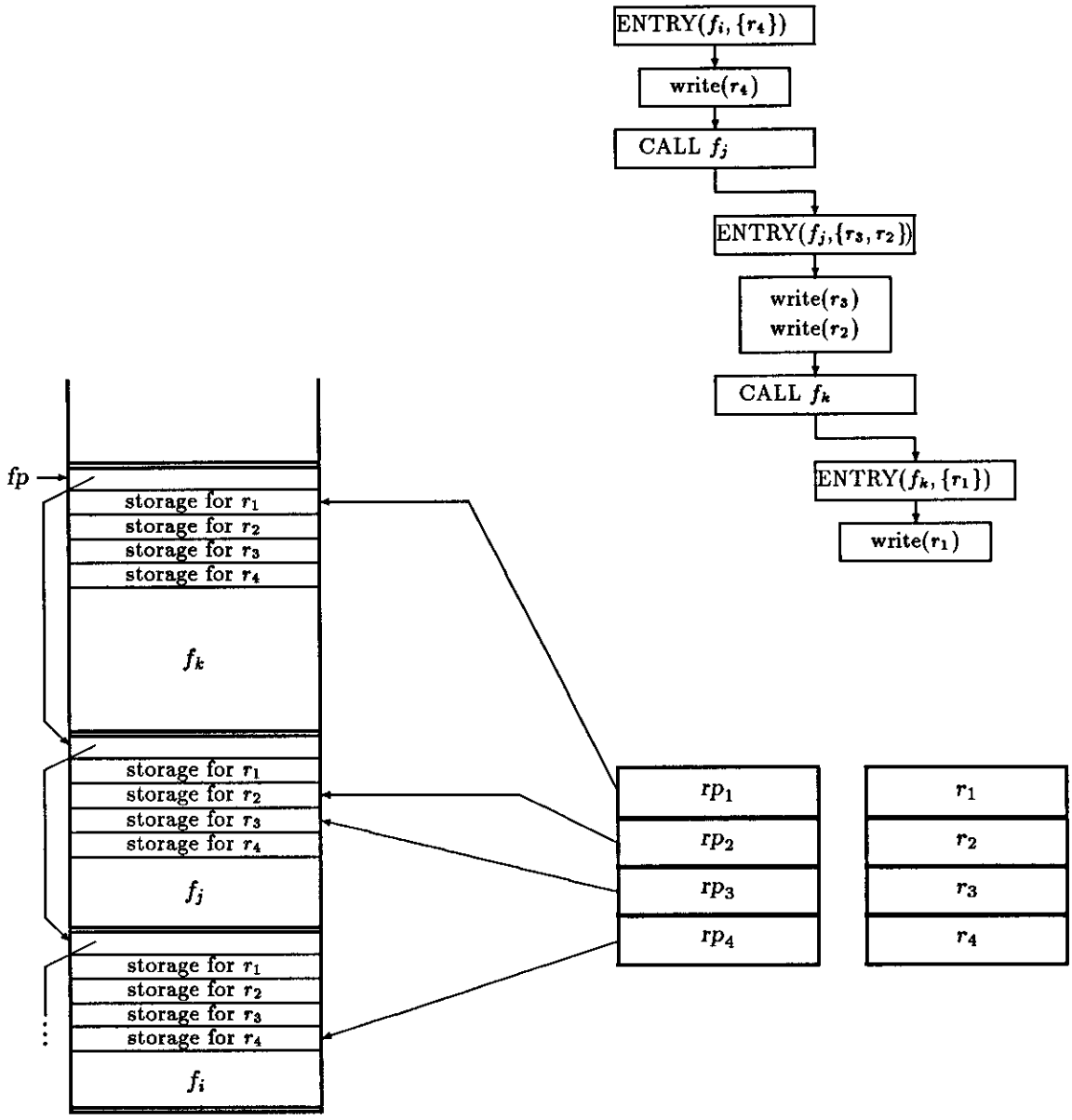

Figure 4: Register Pointers

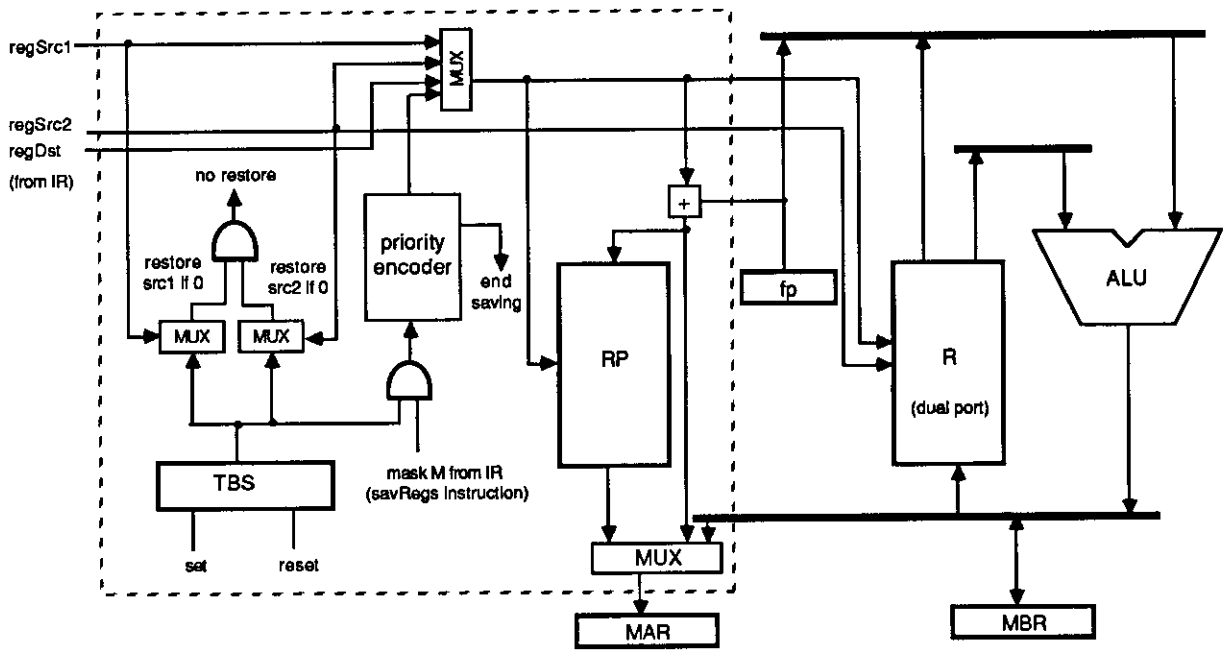

Figure 5: Implementation of Policy G

4. Two multiplexers, one for the memory address register (MAR) and the other for the register number to be used to address both register files (R and RP).

5. A priority encoder of the AND of both masks to determine the next register to save.

6. Logic to determine whether to restore 0, 1, or 2 registers. This consists of two multiplexers and one AND gate.

With this data section the operations related to saving/restoring would be as follows:

i) For the instruction **savRegs**, the priority encoder generates, in sequence, the register numbers of those that have to be saved. For each of them, the corresponding pointer is transferred to the MAR and the values to the memory buffer register (MBR). A memory write is performed and the corresponding bit of the *TBS* cleared. The clearing of this bit makes the priority encoder produce the next register number. The prefetched instruction is not loaded in the *IR* until the end saving condition is reached.

ii) During a *write* to a register, the corresponding pointer is stored in *RP* and the corresponding bit of *TBS* is set.

iii) For *restoring*, the fields of the instruction specifying the source registers are used together with the corresponding bits of the *TBS* to determine whether registers have to be restored. If *no-restore* = *1*, the operation is executed in a normal manner. If *no-restore* = *0*, it is necessary to restore one or two registers. This restoration is done as normal register loads from memory, using as addresses the frame pointer plus the corresponding register numbers. These addresses are also stored in the pointer register file. For each register restored the corresponding bit of TBS is set. After the registers are restored the instruction is executed in the normal way.

iv) During *return* it is necessary to reset the bits of TBS corresponding to the 1's of mask M (given by the return instruction).

v) For a *context switch* the registers are saved in the corresponding activation records so that the registers RP never have to be saved/restored. When the context has to be restored, the TBS is set to zero and no registers are restored. When the register is needed (i.e., it is read), then the register is loaded and its associated RP register is initialized.

The implementation described indicates that the additional hardware required for this saving/restoring policy consists mainly of a register set for the pointers and of an adder. The cost, in additional modules or circuit area in a VLSI implementation, can be justified by the reduction in RSR traffic provided by the policy. The basic processor cycle is not increased by these additions since they are decoupled from the main datapaths and the operations that have to be done during normal register reads and writes are performed overlapped with the normal cycle.

No major impact of this RSR policy is expected in the performance of the pipeline in our RISC-like processor. Since the *TBS* mask is not written until the 3rd stage (at the same time that the register is written), a hazard condition arises when the destination register of the current instruction is used as one of the source registers for the next one. However, the control unit already has to detect this situation to perform forwarding. Thus, the "false" restoring signal (because the corresponding bit has not been written yet) can also be detected and ignored.

Observe that in a architecture with a conventional register saving/restoring scheme (Policies A or B), all the registers saved are restored either during the execution of the return instruction (with a register mask) or right before (Policy B) or after (Policy A) the return (with explicit instructions). Thus, although the number of cycles required for each individual restoring operation might be larger for Policy G than the static policies (because the execution of the instruction has to be aborted), the total delay introduced in the pipeline is less for Policy G than for Policies A or B because only registers required are restored after the return. A more detailed analysis of the implementation is required to compare the effect of these alternatives on the performance of the pipeline.

3.3 A Comparison with Policy A-lvOpt

Policy G can use the standard leaf-function optimization as is used for Policy B. The RSR traffic generated by Policy G-lf is shown in Figure 6. As we can see, our measurements show that Policy G-lf generates between 47% (for 6 to-be-preserved registers) and 22% (for 24) of the RSR traffic produced by Policy A-lvOpt.

Moreover, 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. On the other hand, in the same two situations, the RSR traffic generated by Policy A-lvOpt is 118%. Thus, the overall data memory traffic might not be reduced for a larger register set when the static RSR policies are used. In fact, this occurs for Policy A-live as we will see in Section 4.

In addition to reducing the RSR traffic, Policy G-lf also reduces the number of registers to be saved/restored during context switching. The average number of to-be-preserved registers to be saved ranges from 3.0 when there are 6 to-be-preserved registers to 7.8 when there are 32. None of

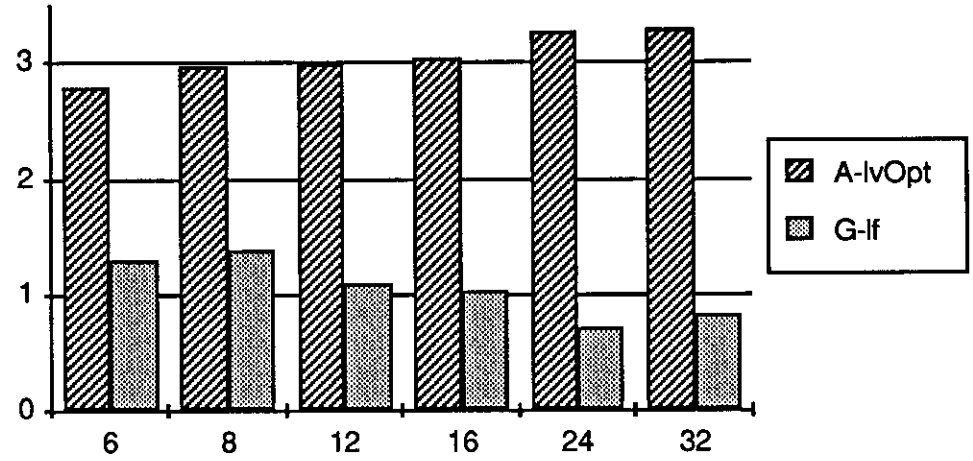

Figure 6: RSR Traffic for Policy G with Leaf-Function Optimization

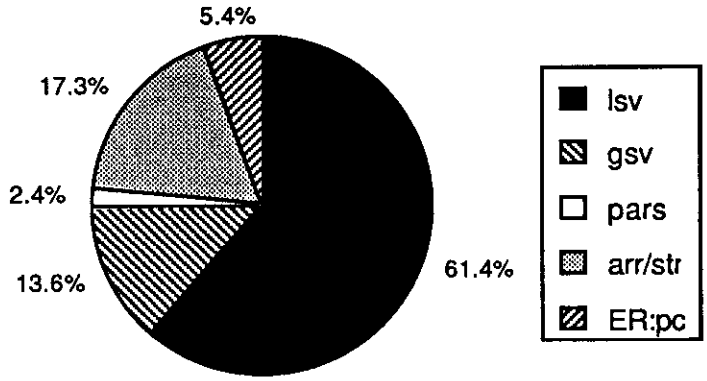

Figure 7: Distribution of Data Memory Traffic

the to-be-preserved registers have to be restored when the process is reshedule to run because they will be restored dynamically when a machine instruction reads them. The to-be-destroyed registers and the environment registers (i.e., *pc*, *sp*, *psw*, ...) have to be saved/restored as usual.

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.

4 Speed-Up

In this section we compare the overall data memory traffic reduction for Policies A-live, A-lvOpt, and G-lf and we evaluate the speed-up that we might obtain for a RISC-like machine.

Figure 7 shows the distribution of data memory references for the four programs measured: local scalar variables (*lsv*), global scalar variables (*gsv*), parameter passing (*pars*), arrays and

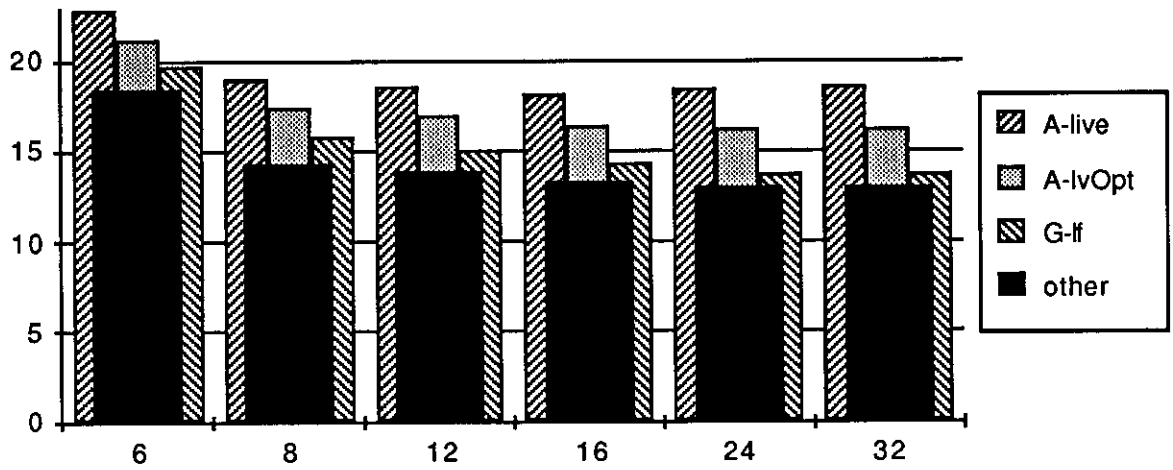

Figure 8: Overall Data Memory Traffic

structures (*arr/str*), and the saving/restoring traffic caused by the return address (*ER:pc*). No traffic is accounted for temporary variables since these operations are performed in a small number of registers[7].

Figure 8 shows the overall traffic generated per function when we eliminate the traffic caused by the pushing of parameters in the stack by passing them through registers and the traffic caused by the return address by providing multiple PCs. The *other* traffic (black boxes) corresponds to the traffic caused by the local and global scalar variables and arrays and structures. Since only local scalar variables are allocated by the PCC (see Section 1.1), the traffic reduction corresponds to these when more of them are assigned to registers. Notice that almost all of them have been allocated when 16 to-be-preserved registers are available (since there is only a slight traffic reduction for the *other* traffic). Policy G-lf generates between 84% (for 24 registers) and 93% (for 6) of the traffic produced by Policy A-lvOpt and between 74% and 87% of the standard Policy A-live. Also, notice that as we mentioned in Section 3.3, there is no data memory traffic reduction for Policy A-live for larger register sets (more than 16 registers) because the reduction in traffic for the non-allocated locals is compensated by the increase in RSR traffic.

To estimate the speed-up obtained by Policy G-lf we have assumed that each register-to-register instruction is executed in one cycle and that load/store instructions require one or two extra cycles to access memory. The reason for considering the two memory speeds is that if the processor cycle keeps getting reduced, then it is possible that 2 extra cycles will be required to access memory because of chip bandwidth limitations. The execution time (T) of a program with N instructions is given by:

$$T = N(1 + c\{\text{lsv} + \text{gsv} + \text{arrays \& structures} + \text{RSR}\})$$

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

As we mentioned above, no traffic is accounted for the parameter passing and the return address. Figure 9 shows the speed up obtained by Policies A-lvOpt and G-lf with respect to Policy A-live

[7]Our previous measurements on 3 of the 4 programs measured (NROFF, SORT, and VPCC) showed that 2 registers are enough to evaluate 95% of the arithmetic expressions [Hugu85a].

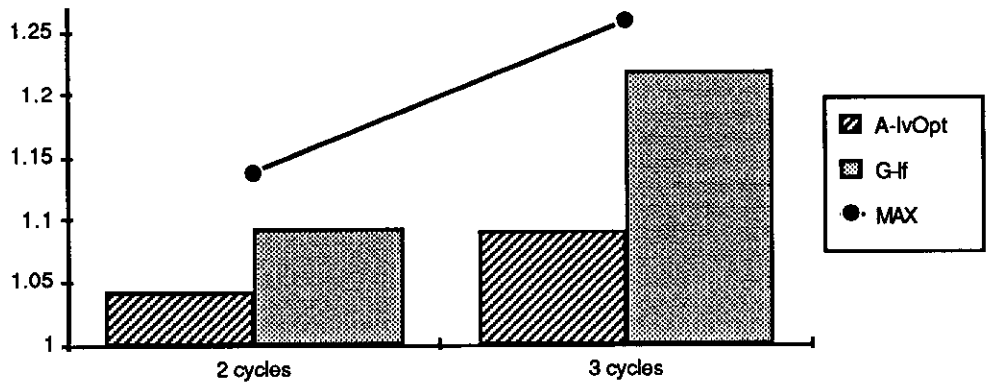

Figure 9: Speed-Up for Intra-Procedural Optimizations

for 24 to-be-preserved registers and the two different memory speeds. The maximum speed up shown is obtained when the RSR traffic has been completely eliminated. In this case, our programs would be executed 9.3% and 20.6% faster, respectively. When the RSR traffic is accounted, the programs would be executed 3.8% and 8.3% faster with Policy A-lvOpt and 8.1% and 17.3% faster with Policy G-lf. The speed up obtained with Policy G-lf with respect to Policy A-lvOpt is 4.5% and 10.5%.

5 A Comparison with Multiple-Window Register Files

In this section we compare the RSR traffic generated by our best policy for single-window architectures to the three existing schemes for multiple-window architectures: fixed-size windows [Kate83], variable-size windows [Ditz82], and multi-size windows [Hugu85b]. The RSR traffic for these schemes is presented for three different register file sizes: 32, 64, and 128. As we can in Figure 10, to be able to have a small RSR traffic for any of the mentioned schemes, at least a 64-register file is necessary. However, this file will have a negative effect on the processor cycle time. For instance, Sherburne [Sher84] claims that the datapath cycle time increases with the square root of the register-file size and Ditzel et al. [Ditz87a] say that the access time for register files larger than 16 or 32 increases about 30% when the register-file size is doubled. Therefore, a 32-register file might be more convenient and, in this case, our Policy G-lf generates the least traffic.

Although most register-file designs use fixed-size windows due to its implementation simplicity (C/70, RISC, PYRAMID 90X, CELERITY C1200, SOAR, SPUR), fixed-size windows do not use the register file efficiently because the register allocator cannot use all the registers in the window for every function. For instance, the average number of registers unused for a window size of 8 registers is 4.15; of 12, 7.91; of 16, 11.74; and of 24, 19.62. These registers must be saved/restored on overflow/underflow and during context switching even though they do not have any significant value.

To be able to use every register in the file, some designs (CRISP, Dragon) use variable-size windows. In this case, the exact number of registers needed by the register allocator are used.

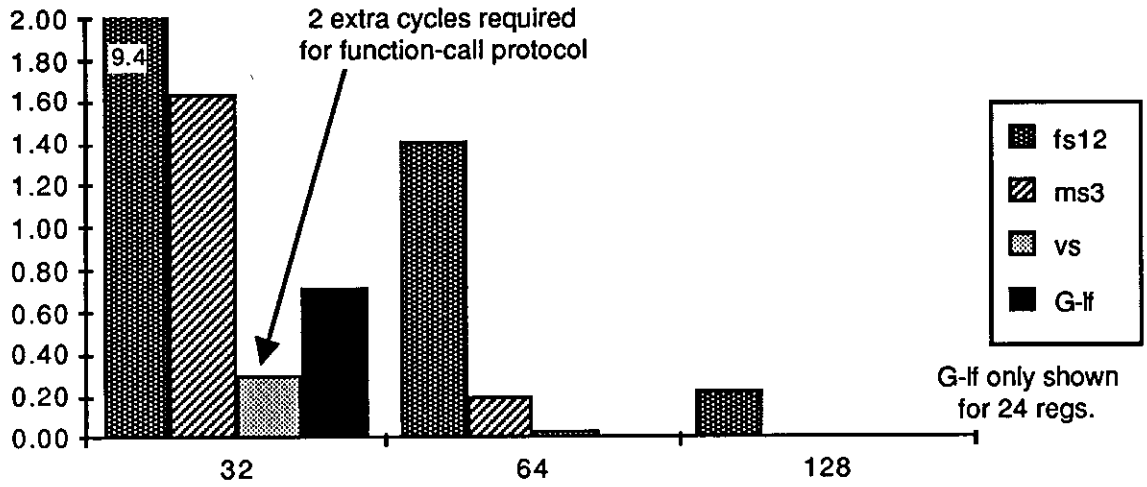

Figure 10: RSR Traffic for Multiple Windows

Thus, the register file has no unused registers. However, two more instructions are necessary to implement the function-call protocol: one used upon entry at the callee to allocate the window and a second one at the caller executed after return to detect underflow [Ditz82]. Only the exact number of registers are transferred on overflow and underflow.

Figure 10 shows the RSR traffic generated by a register file with fixed-size windows of 12 registers (fs12) and variable-size windows (vs) of maximum size 12. As we can see, the RSR traffic generated by variable-size windows for a 32-register file is small (0.31) and it is insignificant for larger register files. However, if we consider that each function requires up to two extra memory cycles due to the two extra instructions[8], then the overall number of cycles required might be worse than the ones required by Policy G-lf.

To combine the advantages of both approaches it is possible to use multi-size windows. By default, the smallest window is allocated (4 registers). If the callee requires more registers, an instruction is executed upon entry to expand the window. When the function returns, we check if the largest possible window is in the file. If not, an underflow condition is generated. In this case, more registers are transferred (the largest window) rather than the exact number of registers being needed (as for variable-size windows). Thus, only two instructions are necessary (as fixed-size windows) to implement the function-call protocol when the allocator does not require more registers for a function. On the average, our measurements show that only 26% of the executed functions require more than 4 registers (including one for the return address).

Figure 10 also shows the RSR traffic generated by 3-size windows (4, 8, and 12; labeled *ms3*). As we can see, 3-size windows generate 18% of the traffic produced by fixed-size windows for a 32-register file and 14% for a 64-register file. However, the RSR traffic generated is still larger than the one produced by a single-window architecture with Policy G-lf: this generates 47% of the RSR traffic produced by 3-size windows.

[8] We said "up to" because some optimizations have been proposed to eliminate the instruction to check for underflow when the registers are not needed any longer by the caller (for instance, when it is followed by a return) [Band87].

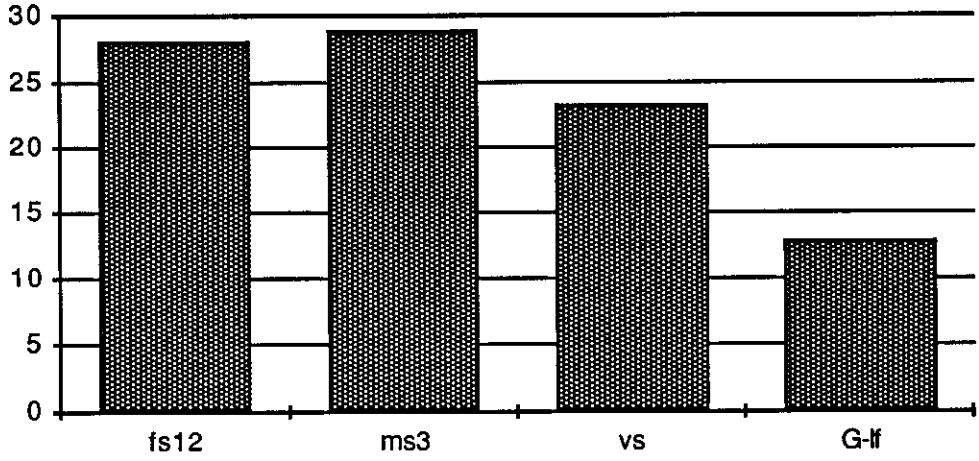

Figure 11: Average Number of Registers to Be Saved During Context Switching

Moreover, a single-window architecture with Policy G-lf also has the smallest average number of registers to be saved during context switching. As we can see Figure 11, for a 32-register file, 28.0 registers need to be saved for 12-register fixed-size windows[9], 28.8 for 3-size windows, 23.3 for variable-size windows, and 12.8 for Policy G-lf (including the program-counter register and 4 to-be-destroyed registers).

6 Conclusions

In this paper we have shown that a *dynamic* architectural policy which takes into account the register usage being performed during program execution together with the conventional *intra-procedural leaf-function optimization*, Policy G-lf, generates less register saving/restoring traffic than the conventional static policies used for single-window architectures (to save registers either at the caller or at the callee) and the existing schemes for multiple-window architectures (fixed-size windows, variable-size windows, and multi-size windows). Moreover, we have sketched the implementation of Policy G in a RISC-like processor and shown that its implementation does not affect the processor cycle time because the operations required are performed in parallel with the main CPU activities. Finally, we have also estimated the speed-up factor for a RISC-like processor with Policy G-lf with respect to Policy A-lvOpt, our best static policy. When there are 24 to-be-preserved registers, our measurements have shown a speed up of 4.5% when load/store instructions execute in 2 processor cycles and and 10.5% when they execute in 3 processor cycles.

References

[Aho86] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[9]That is, 2 windows of 12 registers each plus 4 overlapped registers.

[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.

[Band87] S. Bandyopadhyay, V.S. Begwani, and R.B. Murray, "Compiling for the CRISP Microprocessor," in *Spring COMPCON '87*, February 1987, pp. 96–100.

[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.

[Chow83] F.C. Chow, *A Portable Machine-Independent Global Optimizer—Design and Measurements*, PhD dissertation, Computer Systems Laboratory, Stanford University, Stanford, CA 94305-2192, December 1983. Published as Technical Report 83-254.

[Chow84] F. Chow and J.L. Hennessy, "Register Allocation by Priority-based Coloring," in *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, June 1984, pp. 222–232.

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

[Chow88] F.C. Chow, "Minimizing Register Usage Penalty at Procedure Calls," in *SIGPLAN'88 Conference on Programming Language Design and Implementation*, June 1988, pp. 85–94.

[Colw85] R.P. Colwell and et al., "Computers, Complexity, and Controversy," *Computer*, vol. 18, no. 9, September 1985, pp. 8–20.

[Cout86] 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.

[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.

[Ditz87a] 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.

[Ditz87b] D.R. Ditzel, H.R. McLellan, and A.D. Berenbaum, "The Hardware Architecture of the CRISP Microprocessor," in *Proceedings of the 14th Annual International Symposium on Computer Architecture*, June 1987, pp. 309–319.

[Eick87] R.J. Eickemeyer and J.H. Patel, "Performance Evaluation of Multiple Register Sets," in *Proceedings of the 14th Annual International Symposium on Computer Architecture*, June 1987, pp. 264–271.

[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.

[Hech77] M.S. Hecht, *Flow Analysis of Computer Programs*, Elsevier North-Holland, 1977.

[Henn84] J.L. Hennessy, "VLSI Processor Architecture," *IEEE Transactions on Computers*, vol. C-33, no. 12, December 1984, pp. 1221–1246.

[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.

[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.

[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 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.

[Hugu88] M. Huguet, *Architectural and Compiler Support for Efficient Function Calls: A Proposal*, Technical Report CSD-880030, UCLA Computer Science Department, Los Angeles, CA 90024, June 1988.

[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.

[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.

[Kern78] B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.

[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.

[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.

[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.

[Patt82] D.A. Patterson and C.H. Séquin, "A VLSI RISC," *IEEE Computer*, vol. 15, no. 9, September 1982, pp. 8–21.

[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.

[Radi82] G. Radin, "The 801 Minicomputer," in *Proc. Symp. on Architectural Support for Programming Languages and Operating Systems*, March 1982, pp. 39–47.

[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.

[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.

[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.

[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.

[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.

[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.

[Wall88] D.W. Wall, "Register Windows vs. Register Allocation," in *SIGPLAN'88 Conference on Programming Language Design and Implementation*, June 1988, pp. 67–78.

[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.

[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.