Strategies for Managing the Register File in RISC

YUVAL TAMIR, STUDENT MEMBER, IEEE, AND CARLO H. SÉQUIN, FELLOW, IEEE

Abstract—**The RISC (reduced instruction set computer) architecture attempts to achieve high performance without resorting to complex instructions and irregular pipelining schemes. One of the novel features of this architecture is a large register file which is used to minimize the overhead involved in procedure calls and returns. This paper investigates several strategies for managing this register file. The costs of practical strategies are compared with a lower bound on this management overhead, obtained from a theoretical *optimal strategy*, for several register file sizes.**

While the results concern specifically the RISC processor recently built at U.C. Berkeley, they are generally applicable to other processors with multiple register banks.

Index Terms—**Cache fetch strategies, computer architecture, procedure calls, register file management, RISC, VLSI processor.**

I. Introduction

INVESTIGATIONS of the use of high-level languages show that procedure call/return is the most time-consuming operation in typical high-level language programs [8], [9] due to the related overhead of passing parameters and saving and restoring of registers. The RISC architecture [8], [9] includes a novel scheme that results in highly efficient execution of this operation.

In conventional, register-oriented computers, the procedure call/return mechanism is based on a LIFO stack of variable size *invocation frames* (activation records). When a procedure is called, an area on top of the stack is used for storing the input arguments, saving the return address and register values, allocating local variables and temporaries, and, if the procedure calls another procedure, storing output arguments. A procedure's invocation frame denotes this area on the stack. At any point in time, the number of invocation frames in the stack is the current *nesting depth*. The invocation frame of the calling procedure overlaps that of the called procedure so that the memory locations containing the parameters passed from the calling procedure to the called procedure are part of both frames.

In most computers, register/register operations can be performed faster than the corresponding memory/memory operations. Therefore, the most heavily used local variables and temporaries are placed in registers. When a procedure is called, it must save the value of all the registers it will use and restore these values before returning control to the calling procedure. Analysis of the dynamic behavior of Pascal and C programs, executing on a VAX 11/780, has shown [8], [9] that saving and restoring register values and writing and reading of parameters from the common area of the caller and the callee are responsible for more than 40 percent of the data memory references.

In RISC, the call/return mechanism is based on *two* LIFO stacks. One of the stacks (henceforth "STACK1") contains *fixed size* frames which hold scalar quantities of the invocation frame (i.e., scalar input arguments, the return address, scalar output parameters, and scalar local variables and temporaries). The second stack (henceforth "STACK2") contains variable size frames, some of which may be empty (i.e., their size is zero). This stack is used for all nonscalar variables which are normally placed on the single stack in conventional computers. It is also used for scalars in case there is not enough space in the fixed size frame on STACK1.

The size of the STACK1 frame in RISC was determined based on a study by Halbert and Kessler [5]. The dynamic behavior of nine noninteractive UNIX™ C programs was analyzed. These programs included the main part of the C compiler *ccom*, the Pascal interpreter *pi*, the UNIX copy command *cp*, the *troff* text formatter, and the UNIX *sort* program. This study showed that a fixed frame size of 22 "words" (22 registers), with an overlap of six "words" between adjacent frames, is sufficient for all the scalar variables and arguments in over 95 percent of the procedure calls.

The implementation of STACK2 in RISC is identical to the implementation of the single LIFO stack in conventional computers: the stack itself resides in memory, there is a processor register that serves as a stack pointer, and there is another register that serves as the frame pointer [4]. There is no special hardware support for operations on STACK2 but, due to STACK1, such operations are far less frequent than operations on the LIFO stack of conventional computers. Since the implementation and operation of STACK2 is identical to those of the stack in conventional computers, STACK2 will not be discussed any further in this paper.

In conventional computers, registers are used for storing part of the invocation frame of the currently executing procedure (i.e., the top frame on the stack). In RISC, there is a large register file that is divided into several fixed size "register banks," each of which can hold one STACK1 frame. Since each STACK1 frame partially overlaps the previous STACK1 frame and the next STACK1 frame, each register bank shares

Manuscript received July 14, 1982; revised January 3, 1983. This work was supported by the Defense Advanced Research Projects Agency under ARPA Order 3803, and monitored by Naval Electronic System Command under Contract N00039-81-K-0251.

The authors are with the Computer Science Division, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720.

™ UNIX is a trademark of Bell Laboratories.

0018-9340/83/1100-0977$01.00

some of its registers with the two neighboring register banks.

The STACK1 frame used by the currently executing procedure, is always in one of the register banks. At each point in time, the contents of one of the register banks are addressable as registers, thus providing a "window" into the register file. This register bank is always the one containing the STACK1 frame of the currently executing procedure. A procedure call modifies a hardware pointer and "moves" the window to the next register bank in the register file, where the STACK1 frame of the called procedure resides. Thus, for example, register 15 (R15) in the calling procedure is in a different physical position in the register file from R15 in the called procedure, although the operand specifier for R15 is identical in the two procedures.

A return instruction restores the previous value of the above mentioned hardware pointer so the previous values of all the registers are "restored" without any data movement. Furthermore, no memory references are required for passing arguments since they are passed in registers which are in the region of overlap between the register banks containing the STACK1 frames of the caller and the callee.

By using this scheme, a procedure call in RISC can be made as fast as a jump and with fewer accesses to data memory than are required in conventional computers.

Since the size of the register file is limited, there is a need for a mechanism which will handle the case when the procedure nesting depth exceeds the number of STACK1 frames which fit in the register file. When a procedure call is executed, a new "empty" register bank is needed. If all the register banks in the register file are in use, an "overflow" occurs. This overflow causes a trap which is handled by operating system software. The operating system must free one or more register banks to make room for the new frame. Since the STACK1 frames in the register banks which are "freed" must be preserved, the software copies the frames to a conventional LIFO stack which is kept in memory and contains only STACK1 frames.

When a return instruction is executed, the window must be moved to a register bank containing the previous frame (i.e., the frame of the calling procedure). If all the register banks are free (i.e., the calling frame is not resident), an "underflow" occurs. This underflow causes a trap, upon which the operating system software loads one or more frames from memory where they were stored when an overflow occurred.

The register file is simply a write-back cache of STACK1. The cache blocks are the STACK1 frames. The top few frames of STACK1 are in the register file while the rest are in memory. When an underflow occurs, one or more occupied STACK1 frames are *fetched* from memory. When an overflow occurs, one or more register banks are "freed." This can be interpreted as "fetching" empty STACK1 frames from memory. Since in both cases the "fetching" is done by software, there is great flexibility in defining the cache *fetch strategy* (algorithm) [10]. This strategy determines the number of frames to be moved to/from memory when an overflow/underflow occurs.

In this paper, several fetch strategies are considered. A theoretical "optimal strategy" is developed and is used as a reference point for evaluating the performance of several practical strategies. In addition, the effect of register file size on the performance of different strategies is investigated.

II. The Optimal Strategy

In this section an *optimal strategy* for managing the register file will be discussed. This strategy requires unbounded look-ahead (possibly to the end of the call/return trace) and is therefore only useful as a lower bound on the cost of practical strategies. A proof that the proposed strategy is, in fact, "optimal" is presented.

A. Definitions

In order to facilitate further discussion, some formal definitions are required.

When a program is executing, its nesting depth constantly changes: every procedure call increases the nesting depth by one and every return decreases the nesting depth by one. Hence, for every execution of a program, there is a corresponding sequence of nesting depths. This sequence will be called a *procedure nesting depth sequence* (PNDS).

Definition 1: A *procedure nesting depth sequence* (PNDS) is a sequence of integers $D = (d_1, d_2, \cdots, d_n)$ where $d_1 = 1$, $d_i > 0$ for $1 \le i \le n$ and $|d_i - d_{i-1}| = 1$ for $2 \le i \le n$.

The integer i is an index into the PNDS; d_1 is the nesting depth at the beginning of the program. For each i, $2 \le i \le n$, d_i is the nesting depth after $i - 1$ calls and returns are executed (i.e., after $i - 1$ changes in the nesting depth). Henceforth, an index into the PNDS will be called a *location*. An example of a PNDS is shown in Fig. 1.

The frames of STACK1 are numbered from 1 to m (with m being the current nesting depth, i.e., the number of the frame of the currently executing procedure). The top (i.e., highest numbered) few frames of the stack are always in the register file while the rest are in memory.

Definition 2: The *register file position* (RFP) is the number of the lowest numbered frame which is in the register file.

When an overflow occurs, the lowest number frame(s) in the register file are copied to memory and the register banks they occupy in the register file are "freed." This increases the register file position. Similarly, when an underflow occurs the RFP is decreased. Thus, the number of times the RFP is changed during the execution of the program is equal to the sum of the number of overflows and the number of underflows which occur.

Definition 3: A *register file move* (RFM) denotes an increase or decrease in the register file position.

Definition 4: The *size* of the register file move is the absolute value of the difference between the RFP before the move and the RFP after the move.

If the current nesting depth is d, the STACK1 frame being used by the currently executing procedure, is the one labeled d. The register file position must be such that this frame is contained in the register file. Hence, if the register file can hold w frames and if the RFP is p, then $p \le d < p + w$. Before execution begins, the RFP is some positive integer p_0. During the execution of a program with a PNDS $D = (d_1, d_2, \cdots, d_n)$, for each nesting depth d_i, the corresponding RFP p_i must be such that the above condition is satisfied, i.e., $p_i \le d_i < p_i + w$.

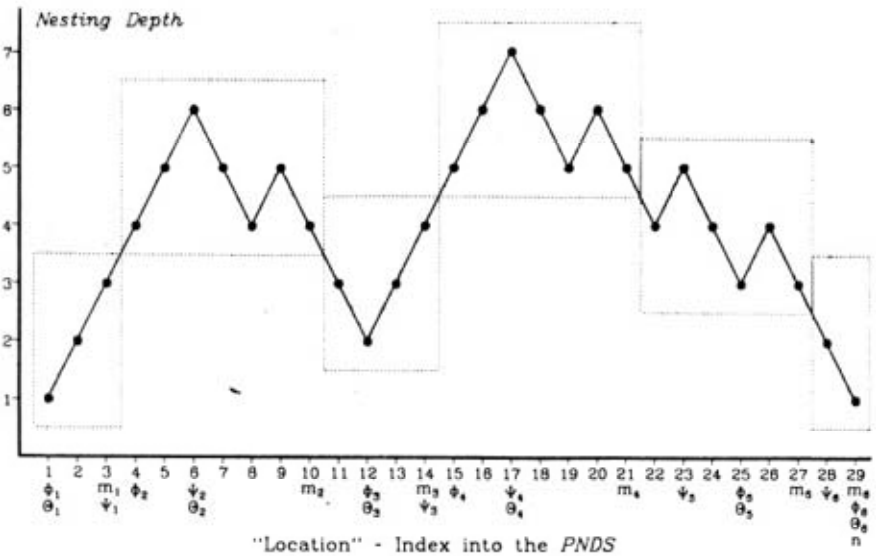

Fig. 1. PNDS and optimal RFP's.

Definition 5: Given a PNDS $D = (d_1, d_2, \cdots, d_n)$ and a register file that can hold w frames, a valid *register file position sequence* (RFPS) is a sequence of RFP's: $P = (p_0, p_1, p_2, \cdots, p_n)$ such that the p_i's are positive integers and for all i, $1 \le i \le n$, $p_i \le d_i < p_i + w$.

There is a one-to-one correspondence between nesting depths in the PNDS and RFP's in the RFPS. Successive RFP's, p_{j-1} and p_j, in the RFPS may be unequal or equal depending on whether the register file position is modified between the $j-2$ and $j-1$ change in the nesting depth.

Definition 6: If $P = (p_0, p_1, p_2, \cdots, p_n)$ is an RFPS, an RFM is said to occur in *location* j ($1 \le j \le n$) of P, if and only if $p_j \neq p_{j-1}$.

The number of RFM's which occur during some interval in which the program is executing is of interest in this paper. The interval is defined as a subsequence of the RFPS (which corresponds to a subsequence of the PNDS).

Definition 7: If $P = (p_0, p_1, p_2, \cdots, p_n)$ is an RFPS, the number of RFM's occurring in location range $[i, j]$ of P, where $1 \le i \le j \le n$, is the number of unique integers k, such that $i \le k \le j$ and $p_k \neq p_{k-1}$. This number will be denoted by $\mathrm{RFM}_P[i, j]$.

Definition 8: If $P = (p_0, p_1, p_2, \cdots, pn)$, is an RFPS, the *memory traffic* occurring in location range $[i, j]$ of P, where $1 \le i \le j \le n$, is the total number of STACK1 frames moved to/from memory as the RFP is set to $p_i, p_{i+1}, \cdots, p_j$ successively. This number is denoted by $\mathrm{MT}_P[i, j]$:

$$\mathrm{MT}_P[i, j] = \sum_{k=i}^{j} \left| p_k - p_{k-1} \right|.$$

B. What is an "Optimal Strategy"?

There is some overhead involved in handling overflow/underflow traps: saving the current state, determining the cause of the trap, activating the proper trap handling routine, restoring state, and returning to normal execution. Hence, it is desirable to minimize the number of register file overflows and underflows. In addition, there is the direct *cost* involved in actually moving the data to/from memory. For each register file move, this cost is proportional to the number of frames moved (i.e., to the size of the register file move). Hence, it is desirable to minimize the number of frames moved for each overflow/underflow, i.e., the memory traffic which is the result of overflows and underflows.

The problem of finding the "best" RFPS is similar to finding optimal strategies for handling page faults in virtual memory systems. In virtual memory systems it is also desirable to minimize both the number of page faults (since there is overhead involved in handling such faults) and the I/O involved in moving memory pages to/from disk or drum. For the virtual memory problem, Belady [1] developed an "optimal" page replacement algorithm which causes the fewest possible page faults for a program which executes in a fixed number of main memory page frames. Belady's algorithm is not realizable since it requires knowledge of the future portion of the page trace.

In the next sections it is shown that if the entire call return trace of the program (i.e., the PNDS) is known, there exists a RFPS which achieves *both* the minimum number of overflow/underflow traps and the minimum memory traffic resulting from register file moves. It is further shown that knowledge of the entire PNDS is *necessary* for achieving an optimum RFPS.

C. The Existence of an Optimal RFPS

In order to prove the existence of an optimal RFPS, an algorithm for deriving such an RFPS from a given PNDS, is presented. The optimality of the RFPS produced by the algorithm is shown by proving that no other valid RFPS can have fewer register file moves or result in less memory traffic.

An optimal RFPS can be obtained as follows. We start with the RFP at 1 and keep it there until the nesting depth exceeds the number (w) of register banks in the register file. Now the RFP must be changed, i.e., an RFM must occur. In order to determine the *optimal size* of the RFM, we must look ahead in the call/return trace (i.e., in the PNDS). Starting from the current location, we determine the longest subsequence of the PNDS for which a constant RFP is possible (i.e., in which the

difference between the maximum nesting depth and the minimum nesting depth does not exceed $w - 1$). The new RFP is chosen so that it is valid for this entire subsequence. From the end of this subsequence we repeat the procedure until the entire PNDS is covered.

Special handling is required when determining the RFP for the last subsequence in the PNDS. In this case the difference between the maximum and minimum nesting depth within the subsequence may be less than $w - 1$. Hence, there is some freedom in setting the RFP. In order to minimize the memory traffic, the new RFP is chosen so that it is valid for the entire subsequence *and* the absolute value of the difference between the new RFP and the previous RFP is minimized. An example of a PNDS and the corresponding optimal RFPS is shown in Fig. 1.

More formally, the procedure can be stated as follows. Given an arbitrary PNDS $D = (d_1, d_2, \cdots, d_n)$, an optimal RFPS $P = (p_0, p_1, p_2, \cdots, p_n)$, for a register file that can hold w frames, can be obtained as follows:

```
[1] let i = 1, p0 = 1
[2] repeat
[3]     let E = (di, di+1, ..., dm)
            where m is the maximum integer such that
            i ≤ m ≤ n and max (E) − min (E) < w
[4]     if (m < n) or (pi−1 > min (E)) then
[5]         for j = i to m
              let pj = min (E)
[6]     else
[7]         for j = i to m
              let pj = max (E) − w + 1
[8]     let i = m + 1
[9] until i > n
```

First, it must be shown that the algorithm generates a valid RFPS for the given PNDS. Proof of the validity of the algorithm and of the generated RFPS requires proving the following lemmas.

Lemma 1: The **repeat** and **for** loops terminate after a finite number of iterations, i.e., the algorithm always terminates.

Proof: Since n is finite and i is incremented by at least 1 during each iteration of the **repeat** loop, n is an upper bound on the number of iterations through the **repeat** loop.

It is always true that $i \geq 1$ and $m \leq n$. Hence, n is an upper bound on the number of iteration through the **for** loop (either one) each time it is entered. ■

Lemma 2: For all i, $1 \leq i \leq n$, $p_i \leq d_i < p_i + w$, i.e., the RFP's generated by the algorithm are valid.

Proof: From the algorithm, if p_i is set in step 5, then $p_i \leq d_i$ [since $p_i = \min(E)$] and $d_i - p_i < w$ (since $\max(E) - \min(E) < w$). Hence, $p_i \leq d_i < p_i + w$.

If p_i is set in step 7, then $p_i \geq d_i - w + 1$ (since $p_i = \max(E) - w + 1$) and $p_i + w - 1 - d_i < w$ (since $\max(E) = p_i + w - 1$ and $\max(E) - \min(E) < w$). From the first inequality, $d_i \leq p_i + w - 1$ and from the second inequality $p_i < d_i + 1$. Hence, $p_i \leq d_i < p_i + w$. ■

The proof of the optimality of the generated RFPS requires some additional notation. The subsequence E which is defined during the kth iteration of the **repeat** loop will be denoted E_k (it corresponds to the kth setting of the RFP). The corresponding integer m will be denoted m_k. For convenience in notation, we define $m_0 = 0$. The number of iterations that the **repeat** loop executes before terminating will be denoted by K (it corresponds to the number of times that the RFP is adjusted). Note that $1 \leq m_1 < m_2 < \cdots < m_K = n$.

For each location range, $[m_{k-1} + 1, m_k]$, the RFP's in the RFPS generated by the algorithm are constant. Within this location range, Φ_k and Ψ_k are the locations of the first occurrence of the minimum and maximum nesting depths, respectively. More formally, see the following.

Definition 9: Φ_k and Ψ_k are the smallest integers, such that for each k ($1 \leq k \leq K$), both are in the location range $[m_{k-1} + 1, m_k]$, where $d_{\Phi_k} = \min(E_k)$ and $d_{\Psi_k} = \max(E_k)$.

In order to prove the optimality of the RFPS generated by the algorithm, it must be shown that this RFPS results in the lowest possible memory traffic. This will be done by using induction on the K boundaries of $K - 1$ location ranges. These boundaries are defined below and are denoted by Θ_k, for all k such that $1 \leq k \leq K$. The boundary point Θ_k is the location of the first minimum or maximum nesting depth within the location range $[m_{k-1} + 1, m_k]$. If the RFP in the RFPS generated by the algorithm for location range $[m_{k-1} + 1, m_k]$ is less than the RFP for location range $[m_{k-2} + 1, m_{k-1}]$, then $\Theta_k = \Phi_k$, otherwise $\Theta_k = \Psi_k$. More formally:

Definition 10: Θ_k is an integer such that for each k, $2 \leq k \leq K$, $\Theta_k = \Phi_k$ if $d_{\Phi_k} < d_{\Phi_{k-1}}$, and $\Theta_k = \Psi_k$ otherwise. For convenience in notation, we define $\Theta_1 = 1$.

Fig. 1 shows the PNDS from the execution of Ackerman's function with arguments (2, 1). The dotted squares show the "optimal" RFP's for a register file that can hold three frames. In this example, five RFM's are necessary ($\mathrm{RFM}_P[1, 29] = 5$, $K = 6$) and the memory traffic resulting from those RFM's is 12 frames ($\mathrm{MT}_P[1, 29] = 12$).

Let $Q = (q_0, q_1, q_2, \cdots, q_n)$ be an arbitrary valid RFPS for D.

The rest of this section contains a formal proof that the number of RFM's in P and the memory traffic resulting from those RFM's are at most equal to the number of RFM's in Q and the memory traffic resulting from those RFM's, respectively.

Lemma 3: If $K > 1$, then for all k, $1 \leq k < K$, $\mathrm{RFM}_Q[1, m_k + 1] \geq k$.

Proof: See the Appendix.

From the algorithm, for all k, $1 \leq k \leq K$, $d_{\Psi_k} - d_{\Phi_k} \leq w - 1$. It is now shown that for $1 \leq k \leq K - 1$, $d_{\Psi_k} - d_{\Phi_k} = w - 1$.

Lemma 4: If $K > 1$, then for all k, $1 \leq k \leq K - 1$, $d_{\Psi_k} - d_{\Phi_k} = w - 1$.

Proof: See the Appendix.

It should be noted that Lemma 4 makes no claims about the value of $(d_{\Psi_k} - d_{\Phi_k})$, i.e., it makes no claims about the case $k = K$. From the algorithm it is clear that $d_{\Psi_K} - d_{\Phi_K} < w$. So it is quite possible that $d_{\Psi_K} - d_{\Phi_K} < w - 1$.

Lemma 5: If $K > 1$, for all k, $1 \leq k \leq K - 1$, for all i, $m_{k-1} + 1 \leq i \leq m_k$, $p_i = d_{\Phi_k} = d_{\Psi_k} - w + 1$. For the last subsequence, i.e., $k = K$: if $\Theta_k = \Phi_k$, then $p_i = d_{\Phi_k}$, else $p_i = d_{\Psi_k} - w + 1$.

Proof: See the Appendix.

Lemma 6: If $K > 1$, then for all k, $1 \le k \le K$, $MT_Q[1, \Theta_k] \ge MT_P[1, \Theta_k] + |q_{\Theta_k} - p_{\Theta_k}|$.

Proof: See the Appendix.

Using the above lemmas, we can formally prove the "optimality" of the RFPS generated by the algorithm.

Theorem 1: The RFPS P generated by the algorithm for the PNDS D is an *optimal* RFPS for D, i.e., if Q is an arbitrary valid RFPS for D, then

$$RFM_P[1, n] \le RFM_Q[1, n]$$

and

$$MT_P[1, n] \le MT_Q[1, n].$$

Proof: If $K = 1$, then there are no RFM's in P so $RFM_P[1, n] = 0$, $MT_P[1, n] = 0$, and the theorem holds.

Assume $K > 1$. In the algorithm, all the RFP's, corresponding to the same subsequence, are set to the same value (step 5 or step 7). Hence, the only way that $p_i \ne p_{i+1}$ can occur is if $i = m_k$ for some k, $1 \le k \le K - 1$. Thus, $RFM_P[1, n] \le K - 1$.

From Lemma 3, $K - 1 \le RFM_Q[1, m_{K-1} + 1]$. Since $m_{K-1} + 1 \le n$, $RFM_Q[1, m_{K-1} + 1] \le RFM_Q[1, n]$. Hence, $K - 1 \le RFM_Q[1, n]$. Thus, $RFM_P[1, n] \le RFM_Q[1, n]$.

From Lemma 6, $MT_Q[1, \Theta_K] \ge MT_P[1, \Theta_K] + |q_{\Theta_K} - p_{\Theta_K}|$. Since $|q_{\Theta_k} - p_{\Theta_k}| \ge 0$, $MT_Q[1, \Theta_K] \ge MT_P[1, \Theta_K]$. Since $\Theta_K \le n$, $MT_Q[1, n] \ge MT_Q[1, \Theta_K]$. Since $d_{\Theta_K} \in E_K$ and $d_n \in E_K$, $p_{\Theta_K} = p_{\Theta_K+1} = \cdots = p_n$. Thus, $MT_P[\Theta_K + 1, n] = 0$, so $MT_P[1, \Theta_K] = MT_P[1, n]$. Hence, $MT_P[1, n] \le MT_Q[1, n]$.

Q.E.D.

D. The Unrealizability of an Optimal Strategy

When a computer is executing a program, the entire call/return trace is not known ahead of time. In fact, it is unlikely that there is any look-ahead possible. In this section it is shown that knowledge of the entire PNDS is necessary for finding an optimal RFPS.

First, it should be noted that no simplifying assumptions about the properties of the call/return trace of "real" programs can be made. In other words, for every given sequence of integers which satisfies the definition of a PNDS (Definition 1), it is possible to construct a real program whose sequence of nesting depths is the given sequence. This is demonstrated by the program in Fig. 2 (which is written in the C language [7]). When this program is executed, its sequence of nesting depths is identical to the sequence of integers in the array *depthlist* (assuming that the sequence of integers in *depthlist* is a valid PNDS).

To show that unbounded look-ahead on the call/return trace is necessary for achieving an optimal RFPS, consider a system where there is a bounded (or nonexistent) look-ahead; more specifically, a system where at each point in time only the next t calls and returns are known in advance. (Note that in most systems $t = 0$.) Assume that the register file of the system can hold w frames and that there are two programs to be executed: PROG1 and PROG2. These programs have identical call/return traces for the first s calls and returns, where $w + t < s$. At some point, before $s - t$ calls/returns are executed, the nesting depth (in both programs) reaches $w + 1$. The nesting depth stays between 2 and $w + 1$ until a total of s calls/returns are executed. After that, in PROG1 the nesting depth decreases and the program terminates at nesting depth 1. In PROG2, on the other hand, the nesting depth increases to $w + 2$ and then decreases until the program terminates at nesting depth 1.

```
int depthlist[] = {  /* This is the PNDS, 0 terminated */
  1, 2, 3, 2, 3, 4, 3, 2, 1, 0 };
int depthind = 1;
main()
{
  while (depthlist[depthind] > 1) {
    deeper(2);
    depthind = depthind + 1;
  }
}
deeper(curdep)
  int curdep;  /* The current nesting depth */
{
  depthind = depthind + 1;
  while (depthlist[depthind] > curdep) {
    deeper(curdep+1);
    depthind = depthind + 1;
  }
  if (depthlist[depthind] == 0)
    exit(0);
}
```

Fig. 2. A program whose "behavior" follows an arbitrary PNDS.

In both programs, when the nesting depth first reaches $w + 1$, the same information about the call/return trace is available, and therefore any strategy for managing the register file will result in the same action being taken for both programs. This action is clearly *not* optimal for at least one of the programs. For PROG1, the optimal action is to move one frame to memory. This action is not optimal for PROG2 since another overflow will occur when a nesting depth of $w + 2$ is reached. The optimal action for PROG2 is to move two frames to memory so that only one overflow will occur during the execution of the program. Moving two frames to memory is *not* the optimal action for PROG1 since it results in unnecessary memory traffic: moving two frames to and from memory instead of one.

The fact that an optimal strategy is not realizable does not imply that all practical strategies for managing the register file are equally bad. As seen in the next section, simple changes in the strategy for managing the register file may significantly affect the cost of handling calls and returns.

III. Practical Strategies for Managing the Register File

In most real systems, no look-ahead at the call/return trace is possible. Thus, the decision as to how many frames should be moved to/from memory when an overflow/underflow occurs must be based on the previous behavior of the executing program or be completely independent of the PNDS of the executing program.

As indicated above, two factors contribute to the *cost* (execution time) of handling register file overflows and underflows: the handling of the interrupt/trap that is initiated by the overflow/underflow and the actual transfer of the STACK1 frames to/from memory. If the number of frames which are moved when an interrupt occurs is not fixed, some computation may be required in order to calculate this number. The cost of this calculation is included in the cost of handling the interrupt. In order to evaluate the effectiveness of different

strategies for managing the register file, these strategies can be tried out on the call/return trace of benchmark programs. The number of overflows/underflows and transfers of STACK1 frames which result from each strategy can thus be determined. These numbers can then be related to the *cost* of the overflow/underflow handler using the following formula:

$$\text{cost} = \alpha \times (\text{number overflows} + \text{number underflows}) + \beta \times (\text{number frames moved})$$

where α and β are constants: α is the cost of responding to the interrupt and calculating the number of frames to be moved, and β is the cost of moving one STACK1 frame to or from memory.

A. Measurement Technique

The method used for obtaining the call/return trace of the benchmark programs used in this paper relies on the fact that the call/return trace of a program executing on a RISC computer is identical to the call/return trace of the same program executing on any similar computer. In this case, the benchmark programs are all written in the C language [7], and their call/return trace is obtained from their execution on a VAX 11/780. The assembly code produced by the C compiler is processed by an editor script which inserts calls to special procedures before and after each procedure call instruction. When the program is executed, in addition to producing its normal output, it creates a file containing a string of bits. The ith bit in the string corresponds to the ith call/return executed by the program. This bit is 1 if a call was executed, 0 if a return was executed. The bit string is the call/return trace of the program. Routines which simulate different strategies for managing the register file use this string to obtain the number of overflows/underflows and the resulting memory traffic which will occur if the benchmark program is executed using the simulated strategy.

For this study, three benchmark programs were used:

rcc — The RISC C compiler [2] which is based on Johnson's portable C compiler [6]. The call/return trace used was generated by the compiler compiling the UNIX file concatenation utility *cat*. 88 606 calls and returns were executed and a nesting depth of 26 was reached.

puzzle — This is a bin-packing program which solves a three-dimensional puzzle. It was developed by Forest Baskett. During the execution of the program, 42 710 calls and returns were executed and a nesting depth of 20 was reached.

tower — This is a Tower of Hanoi program. The call/return trace used, was obtained for the program moving 18 disks. 1 048 574 calls and returns were executed and a nesting depth of 20 was reached.

In this paper, the cost of handling register file overflows and underflows is assumed to be directly proportional to the number of RISC instructions they require. If no calculation is needed in order to determine the number of frames to be moved, the cost of responding to the interrupt is approximately 30 instructions ($\alpha = 30$ in the above discussion). The cost of moving one STACK1 frame is 16 instructions ($\beta = 16$ in the above discussion).

B. The Cost of "Fixed" Strategies

The simplest strategy for managing the register file is to always move the same number of frames (say i) *to* memory, when an overflow occurs, and always move the same number of frames (say j) *from* memory, when an underflow occurs. For a register file that can hold w frames, such a strategy will be denoted *fixed*(i, j) where i and j are integers such that $1 \leq i \leq w$ and $1 \leq j \leq w$.

When a *fixed* strategy is used, no computation is required in order to determine the number of frames to be moved. Hence, the equation

$$\text{cost} = 30 \times (\text{number overflows} + \text{number underflows}) + 16 \times (\text{number frames moved})$$

is used to evaluate the cost of managing the register file. This equation is also used in evaluating the cost of the optimal strategy, which serves as a lower bound on the cost of other strategies.

1) *Measurement Results:* The actual "performance" of the optimal strategy and *fixed* strategies is presented in this section. All possible fixed strategies for register files containing 3, 5, 7, 9, 13, and 17 register banks have been tried with the three benchmark programs.

Tables I–III summarize the results for each one of the three benchmark programs with six different register file sizes and for seven different strategies. The results include the number of overflows, number of underflows, memory traffic, and cost. For the optimal strategy, the "raw" numbers are presented. For the other six strategies, the figures shown are normalized with respect to the corresponding entries for the optimal strategy with the same register file size. In the three tables w denotes the number of register banks in the register file.

The *fixed* strategies included in the tables are: the best of all *fixed* strategies (i.e., the strategy resulting in the least cost) for the particular program and register file size, the worst of all *fixed* strategies (i.e., the strategy resulting in the greatest cost) for the particular program and register file size, *fixed*$(w, 1)$ which guarantees the minimum number of overflows, *fixed*$(1, w)$ which guarantees the minimum number of underflows, *fixed*$(1, 1)$ which guarantees the minimum memory traffic, and *fixed*$(\lceil w/2 \rceil, \lceil w/2 \rceil)$ which is "symmetrical."

2) *Discussion of Measurement Results:* Although the three benchmark programs used are quite different, the results show many common characteristics in their behavior, as far as the management of the register file is concerned. In addition, the results for the *fixed*$(w, 1)$, *fixed*$(1, w)$, and *fixed*$(1, 1)$ strategies provide an experimental verification to the fact that the "optimal strategy," presented in Section II, does indeed minimize the number of overflows/underflows and memory traffic simultaneously.

The register file size and the way that the register file is managed can significantly affect the cost of procedure calls. Table IV shows the average number of instructions per procedure call required for managing the register file. For every

TABLE I
FIXED STRATEGIES WITH *rcc*

Reg File Size (w)		3	5	7	9	13	17
Best Fixed Strategy		(1,1)	(1,1)	(2,2)	(2,2)	(2,2)	(2,2)
Worst Fixed Strategy		(3,3)	(5,5)	(7,7)	(9,9)	(13,13)	(17,17)
Optimal Strategy (raw)	#Overflows	5828	1483	548	236	83	16
	#Underflows	6200	1232	474	171	71	14
	Mem. Traffic	19006	5554	2482	1376	458	74
	Cost	664936	170314	70372	34226	11948	2084
Best Fixed Strategy (normalized)	#Overflows	1.63	1.87	1.65	1.87	1.67	1.56
	#Underflows	1.53	2.25	1.91	2.58	1.96	1.79
	Mem. Traffic	1.00	1.00	1.46	1.28	1.21	1.35
	Cost	1.31	1.50	1.59	1.60	1.44	1.49
Worst Fixed Strategy (normalized)	#Overflows	2.30	7.06	9.49	10.64	43.40	79.31
	#Underflows	2.16	8.50	10.97	14.69	50.73	90.64
	Mem. Traffic	4.24	18.85	29.33	32.86	204.48	583.05
	Cost	3.15	13.53	20.99	25.54	143.50	387.79
fixed(w,1) (normalized)	#Overflows	1.00	1.00	1.00	1.00	1.00	1.00
	#Underflows	2.82	6.02	8.09	12.42	15.20	19.43
	Mem. Traffic	1.84	2.67	3.09	3.09	4.71	7.35
	Cost	1.89	2.96	3.61	4.05	5.81	8.32
fixed(1,w) (normalized)	#Overflows	3.19	3.88	5.44	5.66	4.83	3.94
	#Underflows	1.00	1.00	1.00	1.00	1.00	1.00
	Mem. Traffic	1.96	2.07	2.40	1.94	1.75	1.70
	Cost	2.01	2.31	2.83	2.57	2.28	2.08
fixed(1,1) (normalized)	#Overflows	1.63	1.87	2.28	2.92	2.76	2.31
	#Underflows	1.53	2.25	2.62	4.02	3.23	2.64
	Mem. Traffic	1.00	1.00	1.00	1.00	1.00	1.00
	Cost	1.31	1.50	1.62	1.85	1.76	1.63
$f_{used}\left(\left\lfloor\frac{w}{2}\right\rfloor,\left\lfloor\frac{w}{2}\right\rfloor\right)$ (normalized)	#Overflows	1.65	2.73	2.38	2.51	1.48	4.19
	#Underflows	1.55	3.29	2.76	3.46	1.70	4.79
	Mem. Traffic	2.02	4.38	4.21	4.30	3.70	16.30
	Cost	1.79	3.71	3.49	3.81	2.88	11.19

TABLE II
FIXED STRATEGIES WITH *puzzle*

Reg File Size (w)		3	5	7	9	13	17
Best Fixed Strategy		(1,1)	(1,1)	(1,1)	(2,2)	(7,7)	(3,3)
Worst Fixed Strategy		(3,3)	(5,5)	(7,7)	(9,9)	(13,13)	(17,17)
Optimal Strategy (raw)	#Overflows	736	159	26	6	1	1
	#Underflows	747	159	30	5	1	1
	Mem. Traffic	2056	514	94	30	14	6
	Cost	77386	17784	3184	810	284	156
Best Fixed Strategy (normalized)	#Overflows	1.40	1.62	1.81	1.67	1.00	1.00
	#Underflows	1.38	1.62	1.57	2.00	1.00	1.00
	Mem. Traffic	1.00	1.00	1.00	1.33	1.00	1.00
	Cost	1.22	1.33	1.38	1.53	1.00	1.00
Worst Fixed Strategy (normalized)	#Overflows	9.80	39.50	130.58	70.83	1462.00	2791.00
	#Underflows	9.65	39.50	113.17	85.00	1462.00	2791.00
	Mem. Traffic	21.05	122.20	505.64	255.00	2715.14	15815.67
	Cost	14.54	77.79	302.82	182.59	2450.39	10806.18
fixed(w,1) (normalized)	#Overflows	1.00	1.00	1.00	1.00	1.00	1.00
	#Underflows	2.96	5.00	6.07	10.80	13.00	17.00
	Mem. Traffic	2.15	3.09	3.87	3.80	1.86	5.67
	Cost	2.05	3.04	3.79	4.36	2.94	6.95
fixed(1,w) (normalized)	#Overflows	3.04	4.99	7.85	6.50	7.00	3.00
	#Underflows	1.00	1.00	1.00	1.00	1.00	1.00
	Mem. Traffic	2.18	3.09	4.34	2.60	1.00	1.00
	Cost	2.08	3.04	4.28	3.17	1.83	1.38
fixed(1,1) (normalized)	#Overflows	1.40	1.62	1.81	2.50	7.00	3.00
	#Underflows	1.38	1.62	1.57	3.00	7.00	3.00
	Mem. Traffic	1.00	1.00	1.00	1.00	1.00	1.00
	Cost	1.22	1.33	1.38	1.70	2.27	1.77
$f_{used}\left(\left\lfloor\frac{w}{2}\right\rfloor,\left\lfloor\frac{w}{2}\right\rfloor\right)$ (normalized)	#Overflows	1.30	2.38	2.38	1.17	1.00	2.00
	#Underflows	1.28	2.38	2.07	1.40	1.00	2.00
	Mem. Traffic	1.86	4.42	5.28	2.33	1.00	6.00
	Cost	1.53	3.33	3.66	1.90	1.00	4.46

call there is a corresponding *return*. Hence, in this context, "procedure call" includes returning from the procedure as well as invoking it.

The data indicate that, even with the optimal strategy, the

TABLE III
FIXED STRATEGIES WITH *tower*

Reg File Size (w)		3	5	7	9	13	17
Best Fixed Strategy		(1,1)	(3,3)	(1,1)	(1,1)	(1,1)	(3,3)
Worst Fixed Strategy		(3,3)	(4,5)	(6,6)	(9,9)	(13,13)	(17,17)
Optimal Strategy (raw)	#Overflows	74898	16912	4128	1026	64	4
	#Underflows	74898	16912	4128	1026	64	4
	Mem. Traffic	262142	65534	16382	4094	254	14
	Cost	8888152	2063284	509792	127064	7904	464
Best Fixed Strategy (normalized)	#Overflows	1.75	1.11	1.98	2.00	1.98	1.00
	#Underflows	1.75	1.11	1.98	2.00	1.98	1.00
	Mem. Traffic	1.00	1.71	1.00	1.00	1.00	1.71
	Cost	1.39	1.42	1.48	1.48	1.48	1.34
Worst Fixed Strategy (normalized)	#Overflows	2.00	4.84	32.26	128.00	64.00	16384.00
	#Underflows	2.00	3.88	32.26	128.00	64.00	16384.00
	Mem. Traffic	3.43	10.00	97.54	577.41	419.28	159158.88
	Cost	2.69	7.23	65.82	359.88	246.87	85309.80
fixed(w,1) (normalized)	#Overflows	1.00	1.00	1.00	1.00	1.00	1.00
	#Underflows	3.00	5.00	7.00	9.00	13.00	17.00
	Mem. Traffic	1.71	2.58	3.53	4.51	6.55	9.71
	Cost	1.86	2.79	3.78	4.75	6.77	9.34
fixed(1,w) (normalized)	#Overflows	3.00	5.00	6.99	8.99	7.00	3.00
	#Underflows	1.00	1.00	1.00	1.00	1.00	1.00
	Mem. Traffic	1.71	2.58	3.52	4.50	3.53	1.71
	Cost	1.86	2.79	3.75	4.74	3.78	1.86
fixed(1,1) (normalized)	#Overflows	1.75	1.94	1.98	2.00	1.98	1.75
	#Underflows	1.75	1.94	1.98	2.00	1.98	1.75
	Mem. Traffic	1.00	1.00	1.00	1.00	1.00	1.00
	Cost	1.39	1.48	1.48	1.48	1.48	1.39
$f_{used}\left(\left\lfloor\frac{w}{2}\right\rfloor,\left\lfloor\frac{w}{2}\right\rfloor\right)$ (normalized)	#Overflows	2.33	1.11	8.47	16.48	1.00	64.00
	#Underflows	2.33	1.11	8.47	16.48	1.00	64.00
	Mem. Traffic	2.87	1.71	17.07	41.31	3.53	329.14
	Cost	2.49	1.42	12.89	29.28	2.30	192.00

TABLE IV
COST OF REGISTER FILE MANAGEMENT PER PROCEDURE CALL

Reg File Size (w)		3	5	7	9	13	17
Optimal Strategy	*rcc*	15.01	3.84	1.59	0.77	0.27	0.05
	puzzle	3.82	0.83	0.15	0.04	0.01	0.01
	tower	16.57	3.94	0.97	0.24	0.02	0.00*
Best Fixed Strategy	*rcc*	19.73	5.77	2.53	1.23	0.39	0.07
	puzzle	4.43	1.11	0.20	0.06	0.01	0.01
	tower	23.00	5.57	1.44	0.36	0.02	0.00*
Worst Fixed Strategy	*rcc*	47.24	52.00	33.33	19.73	38.70	17.30
	puzzle	52.68	84.71	45.15	6.93	32.59	78.94
	tower	44.57	28.44	84.00	87.17	3.72	75.50

cost of managing the register file may become prohibitive if the register file is too small (three register banks). In this case, for two out of the three programs (*rcc* and *tower*), it is likely that a conventional stack mechanism for handling procedure calls would have resulted in better performance. If a larger register file is used, the cost of managing the register file drops sharply. The results indicate that, for a register file of five or more register banks, this scheme compares favorably with the conventional stack mechanism.

Invoking a high-level language procedure and returning from it requires several RISC instructions in addition to those used for managing the register file. Specifically, arguments have to be copied to the area of overlap between the current STACK1 frame and the next STACK1 frame; if the procedure returns a value, it may have to be copied from this overlap area; the stack pointer and frame pointer for STACK2 may need to be updated; the actual RISC *call* and *ret* instructions must be executed. *C* procedures typically have less than four arguments [5]. Hence, in addition to the RISC instructions that manage the register file, between three and seven instructions will be executed for each procedure call/return pair.

If an efficient strategy (such as the "best fixed strategy") is used, the cost of managing the register file decreases as the number of register banks in the register file increases. Once this cost reaches approximately one RISC instruction per procedure call/return pair (e.g., using the "best fixed strategy" with a register file containing nine register banks), it no longer dominates the total number of instructions required for each procedure call/return. In a single chip VLSI microprocessor, chip area is a precious resource. Rather than adding more register banks (e.g., beyond nine), the limited chip area can be used more effectively for other purposes, such as an on-chip cache or hardware support for multiply, that are likely to make a greater contribution to overall processor performance. Even for the benchmarks used here, which reach a relatively high nesting depth [5], a register file with between five and nine register banks seems optimal.

Choosing a "good" strategy is critical to the success of the register file scheme. Tables II and III show that choosing the "wrong" strategy can result in more than four orders of magnitude increase in the cost of managing the register file. Furthermore, if an inefficient strategy is used, an increase in the register file size can result in an *increase* in the cost of managing the register file (since there is an opportunity to generate more useless memory traffic). In most cases, the best fixed strategy is to minimize the memory traffic (i.e., use the *fixed*(1, 1) strategy). This can be explained by the fact that the cost of moving one frame to memory and then from memory back to the register file is about the same as the cost of handling the trap when an overflow or underflow occurs. Hence, the immediate cost of unnecessarily moving a frame (which results in one frame's worth of traffic to memory and later back to the register file) is about equal to the cost of not moving a frame when it should have been moved (an extra overflow or underflow trap). In addition, if an unnecessary move is made, the cost may include the cost of an extra overflow or underflow which will occur later. Hence, the "penalty" for moving one more frame than necessary, when an overflow or underflow occurs, is greater than the "penalty" for moving one fewer frame than necessary. Thus, if the call/return sequence is random, the best fixed strategies are likely to be those that require the movement of only one or two frames when an overflow or underflow occurs. The use of such strategies is further supported by the fact that with the optimal strategy, in cases where there are more than ten overflows/underflows throughout the execution of the program, the average number of frames moved when an overflow or underflow occurs is between 1.4 and 3 and in most cases is approximately 2.

C. Taking the Past into Account

The *fixed* strategies do not attempt to take into account the previous behavior of the executing program. It is conceivable that a strategy that does take past behavior into account would result in a lower cost, closer to that of the optimal strategy.

One way of "taking the past into account" involves keeping track of which register banks have been used since the last overflow or underflow. If two or more STACK1 frames are moved whenever an overflow or underflow occurs, it is clear that, in some cases, it will turn out that too many frames will be moved, resulting in unnecessary memory traffic. When an overflow occurs, register banks are "freed" by copying their contents to memory. If some of the freed register banks remain unused until the next underflow, their contents remain intact and need not be copied from memory to the register file. Similarly, if too many register banks are loaded when an underflow occurs, the contents of those that are unused until the next overflow need not be copied to memory since their contents are already in the appropriate memory locations.

Many practical strategies result in unnecessary memory traffic, i.e., more memory traffic than is required by the optimal strategy. The above technique reduces the memory traffic resulting from any such strategy. Our measurements indicate that with the useless "worst fixed strategy," which produces an exorbitant number of unnecessary moves of STACK1 frames, keeping track of which register banks are used can reduce this memory traffic by up to an order of magnitude. However, with "reasonable" strategies, the gains are less impressive. If the "best fixed strategy" is *fixed*(1, 1) then clearly no gain is possible. With the *fixed*(2, 2) strategy, the decrease in memory traffic is less than ten percent. The above technique requires some extra hardware and a few more instructions in the trap handling routine. When the overhead of these extra instructions is taken into account, the total cost of managing the register file for the *fixed*(2, 2) strategy is about the same as without this extra mechanism. For the *fixed*(1, 1) strategy, the extra instructions will simply add to the cost of managing the register file without any saving in memory traffic.

We have investigated two other methods for "taking the past into account." They both involve determining the number of frames to be moved when an overflow or underflow occurs based on the previous behavior of the program. The first method (henceforth denoted C/R) is to use the call/return trace immediately preceding the overflow or underflow. The second method (henceforth denoted O/U) is to use the trace of overflows and underflows which preceded the trap being handled.

The C/R method can be implemented by adding a special shift register to the processor. Every call instruction shifts a 1 into the register and every return shifts a 0. The routine which handles the overflow/underflow trap examines the contents of this register and determines the immediately preceding call/return trace of the program. This pattern is used to access a table containing the "optimal" number of frames that should be moved, given a particular call/return pattern. This scheme adds very few instructions to the cost of handling the overflow/underflow trap.

The O/U method does not require any additional hardware. The "overflow/underflow trace" is kept in a fixed memory location and is updated each time an overflow or underflow occurs by the routine that handles these traps. The pattern in this memory location is used in the same way as the contents of the shift register for the C/R method.

Both the C/R method and O/U method require finding a mapping between "call/return patterns" or "overflow/underflow patterns" and "number of frames to be moved" so that the total cost is reduced. In order to find such a mapping (for either one of the methods) we tabulated the optimal number of frames to be moved (which can be found given unbounded look-ahead) following various call/return or overflow/un-

derflow patterns for the three benchmark programs. We attempted to use these tables to determine which patterns indicate that a single frame should be moved and in which cases moving more than one frame would be preferable. However, we could not find a single mapping which worked better than the *fixed*(1, 1) strategy for all three programs!

For the three benchmark programs used in this work, it appears that the optimal number of frames to be moved is, for all practical purposes, independent of the immediately preceding call/return pattern of length ten or less. The O/U method shows more promise but the results are inconclusive. Following a suggestion by Denning [3], we tested an O/U method which involved moving two frames after two consecutive overflows or underflows and moving one frame otherwise. For register file sizes of interest (between five and nine frames), the cost of managing the register file using this method was compared to the cost using the *fixed*(1, 1) strategy. Reductions of up to 28 percent in the number of overflows and underflows and increases of up to 59 percent in the memory traffic were measured. When the extra instructions in the trap handling routines are taken into account, the overall cost was either equal to or greater than the cost of the *fixed*(1, 1) strategy in all but one case.

IV. Conclusions

The success of the RISC architecture is due, in part, to the reduction in the number of memory accesses which is possible through the use of the register file [11]. We have shown that the effectiveness of the register file is dependent on choosing the "right" size for the register file and an efficient strategy for deciding how many frames should be moved to/from memory when an overflow/underflow occurs.

Our measurements indicate that with the simple *fixed* strategy, *fixed*(1, 1), the cost of managing the register file is within a factor of two of the cost of the optimal strategy (which requires unbounded look-ahead). For a register file containing more than eight register banks, the *fixed*(2, 2) strategy yields slightly better performance.

If a "reasonable" strategy is used, the cost of managing the register file is inversely proportional to its size. If the register file is too small, the number of overflows and underflows becomes prohibitively large. Since the STACK1 frames have a fixed size, the large number of overflows and underflows results in a lot of memory traffic even when the number of registers actually used (for arguments and local variables) is small. Hence, if the register file is too small, the overall cost of procedure calls may be greater than if a conventional stack mechanism is used. Our measurements indicate that if the register file contains five or more frames, the use of the register file scheme rather than a conventional stack mechanism is worthwhile.

We have attempted to use past behavior of the program in order to predict the future behavior and reduce the cost of managing the register file. So far, our attempts have not succeeded.

The first method (keeping track of which register banks have been used since the last overflow or underflow), reduces the cost of managing the register file only for inefficient strategies. For efficient strategies, such as *fixed*(1, 1) or *fixed*(2, 2), the extra overhead in the trap handling routine was greater than the savings from the reduced memory traffic.

The two other methods attempt to determine the "optimal" number of frames to be moved from the immediately preceding pattern of calls/returns or overflows/underflows. These methods appear ineffective since we could not find a single mapping between either type of patterns and number frames to be moved, which reduces the cost for all three programs. These results, while preliminary, raise serious doubts that a mapping which reduces the cost of managing the register file for a majority of programs could be found. Even in this context, the simplest solution appears to also be the best.

Appendix

Proof of Lemmas 3-6

Lemma 3: If $K > 1$, then for all k, $1 \le k < K$,

$$\mathrm{RFM}_Q[1, m_k + 1] \ge k.$$

Proof: By induction on k.

Basis: $k = 1$. It is shown that $\mathrm{RFM}_Q[1, m_1 + 1] \ge 1$. From the algorithm,

$$\max(E_1) - \min(E_1) < w$$

while

$$\max(E_1 \cup \{d_{m_1+1}\}) - \min(E_1 \cup \{d_{m_1+1}\}) \ge w.$$

Hence, either

$$d_{m_1+1} < \min(E_1)$$

or

$$d_{m_1+1} > \max(E_1).$$

By Definition 1, $d_1 = 1$ and $d_i \ge 1$ for all i, $1 \le i \le n$. Hence,

$$d_1 = \min(E_1 \cup \{d_{m_1+1}\}) = \min(E_1)$$

and

$$d_{m_1+1} = \max(E_1 \cup \{d_{m_1+1}\}).$$

Thus,

$$d_{m_1+1} - d_1 \ge w.$$

Since Q is a valid RFPS for D, $q_1 \le d_1 < q_1 + w$ and $q_{m_1+1} \le d_{m_1+1} < q_{m_1+1} + w$. $d_{m_1+1} \ge d_1 + w$ and $d_1 \ge q_1$ imply that $d_{m_1+1} \ge q_1 + w$. But $q_{m_1+1} + w > d_{m_1+1}$. Hence,

$$q_{m_1+1} + w \ge q_1 + w,$$

i.e., $q_{m_1+1} > q_1$. The fact that $q_{m_1+1} \ne q_1$ implies that at least one RFM occurs in the location range $[2, m_1 + 1]$, so $\mathrm{RFM}_Q[1, m_1 + 1] \ge 1$.

Induction Step: Assuming that this lemma holds for $k = \alpha - 1$, where $2 \le \alpha < K$, it is now proven that it holds for $k = \alpha$. In other words, assuming $\mathrm{RFM}_Q[1, m_{\alpha-1} + 1] \ge \alpha - 1$, it is proven that $\mathrm{RFM}_Q[1, m_\alpha + 1] \ge \alpha$:

If $\mathrm{RFM}_Q[1, m_{\alpha-1} + 1] \ge \alpha - 1$, then either

$$\mathrm{RFM}_Q[1, m_{\alpha-1} + 1] \ge \alpha$$

or

$$\mathrm{RFM}_Q[1, m_{\alpha-1} + 1] = \alpha - 1.$$

The former case implies that $\mathrm{RFM}_Q[1, m_\alpha + 1] \geq \alpha$ (since $m_\alpha + 1 \geq m_{\alpha-1} + 1$) and the lemma is proved. Hence, we can assume $\mathrm{RFM}_Q[1, m_{\alpha-1} + 1] = \alpha - 1$.

The rest of the proof is similar to the proof of the *basis*:

From the algorithm,

$$\max(E_\alpha) - \min(E_\alpha) < w$$

while

$$\max(E_\alpha \cup \{d_{m_\alpha+1}\}) - \min(E_\alpha \cup \{d_{m_\alpha+1}\}) \geq w.$$

Hence, either $d_{m_\alpha+1} < \min(E_\alpha)$ or $d_{m_\alpha+1} > \max(E_\alpha)$.

Assume $d_{m_\alpha+1} < \min(E_\alpha)$:

From the algorithm and the definition of Ψ, $d_{\Psi_\alpha} - d_{m_\alpha+1} \geq w$. Since Q is a valid RFPS for D,

$$q_{\Psi_\alpha} \leq d_{\Psi_\alpha} < q_{\Psi_\alpha} + w$$

and

$$q_{m_\alpha+1} \leq d_{m_\alpha+1} < q_{m_\alpha+1} + w.$$

Hence,

$$q_{\Psi_\alpha} + w > d_{\Psi_\alpha} \geq d_{m_\alpha+1} + w \geq q_{m_\alpha+1} + w,$$

i.e., $q_{\Psi_\alpha} > q_{m_\alpha+1}$.

Assume $d_{m_\alpha+1} > \max(E_\alpha)$:

From the algorithm and the definition of Φ,

$$d_{m_\alpha+1} - d_{\Phi_\alpha} \geq w.$$

Since Q is a valid RFPS for D,

$$q_{\Phi_\alpha} \leq d_{\Phi_\alpha} < q_{\Phi_\alpha} + w$$

and

$$q_{m_\alpha+1} \leq d_{m_\alpha+1} < q_{m_\alpha+1} + w.$$

Hence,

$$q_{m_\alpha+1} + w > d_{m_\alpha+1} \geq d_{\Phi_\alpha} + w \geq q_{\Phi_\alpha} + w,$$

i.e., $q_{m_\alpha+1} > q_{\Phi_\alpha}$.

The fact that $q_{\Psi_\alpha} \neq q_{m_\alpha+1}$ ($q_{\Phi_\alpha} \neq q_{m_\alpha+1}$) implies that there is at least one RFM in the location range $[\Psi_\alpha + 1, m_\alpha + 1]$ ($[\Phi_\alpha + 1, m_\alpha + 1]$). But $\Psi_\alpha \geq m_{\alpha-1} + 1$ ($\Phi_\alpha \geq m_{\alpha-1} + 1$). Hence, there is at least one RFM in the location range $[m_{\alpha-1} + 2, m_\alpha + 1]$, i.e., $\mathrm{RFM}_Q[m_{\alpha-1} + 2, m_\alpha + 1] \geq 1$. But by assumption $\mathrm{RFM}_Q[1, m_{\alpha-1} + 1] = \alpha - 1$. Hence, $\mathrm{RFM}_Q[1, m_\alpha + 1] \geq \alpha$.

■

Lemma 4: If $K > 1$, then for all k, $1 \leq k \leq K - 1$,

$$d_{\Psi_k} - d_{\Phi_k} = w - 1.$$

Proof: From the algorithm,

$$\max(E_k) - \min(E_k) < w$$

while

$$\max(E_k \cup \{d_{m_k+1}\}) - \min(E_k \cup \{d_{m_k+1}\}) \geq w.$$

Hence, either $d_{m_k+1} < \min(E_k)$ or $d_{m_k+1} > \max(E_k)$.

By Definition 1, $|d_{m_k+1} - d_{m_k}| = 1$. Since $d_{m_k} \in E_k$, either $d_{m_k+1} = \min(E_k) - 1$ or $d_{m_k+1} = \max(E_k) + 1$. Hence,

$$\max(E_k \cup \{d_{m_k+1}\}) - \min(E_k \cup \{d_{m_k+1}\}) = \max(E_k) - \min(E_k) + 1.$$

Thus, $\max(E_k) - \min(E_k) \geq w - 1$. But from the algorithm, $\max(E_k) - \min(E_k) \leq w - 1$. Hence,

$$\max(E_k) - \min(E_k) = w - 1,$$

i.e., $d_{\Psi_k} - d_{\Phi_k} = w - 1$.

■

Lemma 5: If $K > 1$, for all k, $1 \leq k \leq K - 1$, for all i,

$$m_{k-1} + 1 \leq i \leq m_k, p_i = d_{\Phi_k} = d_{\Psi_k} - w + 1.$$

For the last subsequence, i.e., $k = K$: if $\Theta_k = \Phi_k$, then $p_i = d_{\Phi_k}$, else $p_i = d_{\Psi_k} - w + 1$.

Proof: For all i, $1 \leq i \leq n$, the value of p_i is set in step 5 or in step 7 of the algorithm.

If $1 \leq k \leq K - 1$, then by Lemma 4,

$$d_{\Psi_k} - d_{\Phi_k} = w - 1.$$

Hence,

$$d_{\Phi_k} = d_{\Psi_k} - w + 1$$

and the same value (d_{Φ_k}) will be assigned to p_i in step 5 or step 7 of the algorithm.

If $k = K$, then it may be the case that $d_{\Psi_k} - d_{\Phi_k} < w - 1$. Hence, it may make a difference whether the value of p_i is assigned in step 5 or in step 7. This is controlled by the value of Θ_k.

If $\Theta_k = \Phi_k$, then, by the definition of Θ,

$$d_{\Phi_k} < d_{\Phi_{k-1}}.$$

Since $k - 1 < K$,

$$p_{m_{k-1}} = p_{\Phi_{k-1}} = d_{\Phi_{k-1}}.$$

Hence,

$$\min(E_k) < p_{m_{k-1}},$$

and the second clause in step 4 of the algorithm is satisfied. Thus, p_i ($m_{k-1} + 1 \leq i \leq m_k$) is assigned a value in step 5 of the algorithm. So

$$p_i = d_{\Phi_k}.$$

If $\Theta_k = \Psi_k$, then, by the definition of Θ,

$$d_{\Phi_k} \geq d_{\Phi_{k-1}}.$$

Since $k - 1 < K$,

$$p_{m_{k-1}} = p_{\Phi_{k-1}} = d_{\Phi_{k-1}}.$$

Hence,

$$\min(E_k) \geq p_{m_{k-1}},$$

and the second clause in step 4 of the algorithm is *not* satisfied. Since $k = K$, $m_k = n$, and the first clause in step 4 of the algorithm is also *not* satisfied. Thus, p_i ($m_{k-1} + 1 \leq i \leq m_k$) is assigned a value of step 7 of the algorithm. So

$$p_i = d_{\Psi_k} - w + 1.$$

■

Lemma 6: If $K > 1$, then for all k, $1 \le k \le K$,

$$\mathrm{MT}_Q[1, \Theta_k] \ge \mathrm{MT}_P[1, \Theta_k] + |q_{\Theta_k} - p_{\Theta_k}|.$$

Proof: By induction on k.

Basis: $k = 1$. It is shown that $\mathrm{MT}_Q[1, \Theta_1] \ge \mathrm{MT}_P[1, \Theta_1] + |q_{\Theta_1} - p_{\Theta_1}|$.

By the definition of Θ, $\Theta_1 = 1$. Hence,

$$\mathrm{MT}_Q[1, \Theta_1] = \mathrm{MT}_Q[1, 1] = |q_1 - q_0|$$

and

$$\mathrm{MT}_P[1, \Theta_1] + |q_{\Theta_1} - p_{\Theta_1}| = \mathrm{MT}_P[1, 1] + |q_1 - p_1| = |p_1 - p_0| + |q_1 - p_1|.$$

By Definitions 2 and 5, for all i, $1 \le i \le n$,

$$1 \le q_i \le d_i < q_i + w$$

and

$$1 \le p_i \le d_i < p_i + w.$$

By Definition 1, $d_1 = 1$. Hence, $q_1 = p_1 = 1$. From the algorithm, $p_0 = 1$. Hence,

$$\mathrm{MT}_P[1, \Theta_1] + |q_{\Theta_1} - p_{\Theta_1}| = |p_1 - p_0| + |q_1 - p_1| = 0.$$

Since $|q_1 - q_0| \ge 0$,

$$\mathrm{MT}_Q[1, \Theta_1] \ge 0.$$

Thus,

$$\mathrm{MT}_Q[1, \Theta_1] \ge \mathrm{MT}_P[1, \Theta_1] + |q_{\Theta_1} - p_{\Theta_1}|.$$

Induction Step: Assuming that this lemma holds for $k = \alpha - 1$, where $2 \le \alpha \le K$, it is now proven that it holds for $k = \alpha$. In other words, assuming

$$\mathrm{MT}_Q[1, \Theta_{\alpha-1}] \ge \mathrm{MT}_P[1, \Theta_{\alpha-1}] + |q_{\Theta_{\alpha-1}} - p_{\Theta_{\alpha-1}}|,$$

it is proven that

$$\mathrm{MT}_Q[1, \Theta_\alpha] \ge \mathrm{MT}_P[1, \Theta_\alpha] + |q_{\Theta_\alpha} - p_{\Theta_\alpha}|.$$

From Definition 8,

$$\mathrm{MT}_Q[1, \Theta_\alpha] = \mathrm{MT}_Q[1, \Theta_{\alpha-1}] + \mathrm{MT}_Q[\Theta_{\alpha-1} + 1, \Theta_\alpha]$$

and

$$\mathrm{MT}_P[1, \Theta_\alpha] = \mathrm{MT}_P[1, \Theta_{\alpha-1}] + \mathrm{MT}_P[\Theta_{\alpha-1} + 1, \Theta_\alpha].$$

Using the induction hypothesis,

$$\mathrm{MT}_Q[1, \Theta_\alpha] \ge \mathrm{MT}_P[1, \Theta_{\alpha-1}] + |q_{\Theta_{\alpha-1}} - p_{\Theta_{\alpha-1}}| + \mathrm{MT}_Q[\Theta_{\alpha-1} + 1, \Theta_\alpha].$$

$\mathrm{MT}_Q[\Theta_{\alpha-1} + 1, \Theta_\alpha]$ is the number of STACK1 frames transferred to/from memory in location range $[\Theta_{\alpha-1} + 1, \Theta_\alpha]$. A change by one in the RFP indicates that one STACK1 frame is transferred to or from memory. Hence, the memory traffic in location range $[\Theta_{\alpha-1} + 1, \Theta_\alpha]$ is at least the difference between the RFP at the beginning of the range and the RFP at the end of the range, i.e.,

$$\mathrm{MT}_Q[\Theta_{\alpha-1} + 1, \Theta_\alpha] \ge |q_{\Theta_\alpha} - q_{\Theta_{\alpha-1}}|.$$

Hence,

$$\mathrm{MT}_Q[1, \Theta_\alpha] \ge \mathrm{MT}_P[1, \Theta_{\alpha-1}] + |q_{\Theta_{\alpha-1}} - p_{\Theta_{\alpha-1}}| + |q_{\Theta_\alpha} - q_{\Theta_{\alpha-1}}|.$$

From Definition 8 and the algorithm,

$$\mathrm{MT}_P[\Theta_{\alpha-1} + 1, \Theta_\alpha] = \sum_{\beta=\Theta_{\alpha-1}+1}^{\Theta_\alpha} |p_\beta - p_{\beta-1}|$$

$$= \sum_{\beta=\Theta_{\alpha-1}+1}^{m_{\alpha-1}} |p_\beta - p_{\beta-1}| + |p_{m_{\alpha-1}+1} - p_{m_{\alpha-1}}| + \sum_{\beta=m_{\alpha-1}+2}^{\Theta_\alpha} |p_\beta - p_{\beta-1}| = |p_{m_{\alpha-1}+1} - p_{m_{\alpha-1}}|.$$

Since $\alpha - 1 < K$, by Lemma 5, $p_{m_{\alpha-1}} = d_{\Phi_{\alpha-1}}$. From the algorithm, $p_{m_{\alpha-1}+1} = p_{m_\alpha}$. Hence,

$$\mathrm{MT}_P[\Theta_{\alpha-1} + 1, \Theta_\alpha] = |p_{m_\alpha} - d_{\Phi_{\alpha-1}}|.$$

Thus,

$$\mathrm{MT}_P[1, \Theta_\alpha] = \mathrm{MT}_P[1, \Theta_{\alpha-1}] + |p_{m_\alpha} - d_{\Phi_{\alpha-1}}|.$$

In the rest of the proof, the following four cases will be handled separately:

Case A: $\Theta_\alpha = \Phi_\alpha$ and $\Theta_{\alpha-1} = \Phi_{\alpha-1}$
Case B: $\Theta_\alpha = \Phi_\alpha$ and $\Theta_{\alpha-1} = \Psi_{\alpha-1}$
Case C: $\Theta_\alpha = \Psi_\alpha$ and $\Theta_{\alpha-1} = \Phi_{\alpha-1}$
Case D: $\Theta_\alpha = \Psi_\alpha$ and $\Theta_{\alpha-1} = \Psi_{\alpha-1}$.

Case A: $\Theta_\alpha = \Phi_\alpha$ and $\Theta_{\alpha-1} = \Phi_{\alpha-1}$:

$$|q_{\Theta_{\alpha-1}} - p_{\Theta_{\alpha-1}}| + |q_{\Theta_\alpha} - q_{\Theta_{\alpha-1}}| = |q_{\Phi_{\alpha-1}} - p_{\Phi_{\alpha-1}}| + |q_{\Phi_\alpha} - q_{\Phi_{\alpha-1}}| = |p_{\Phi_{\alpha-1}} - q_{\Phi_{\alpha-1}}| + |q_{\Phi_{\alpha-1}} - q_{\Phi_\alpha}| \ge p_{\Phi_{\alpha-1}} - q_{\Phi_{\alpha-1}} + q_{\Phi_{\alpha-1}} - q_{\Phi_\alpha} = p_{\Phi_{\alpha-1}} - q_{\Phi_\alpha} = (p_{\Phi_{\alpha-1}} - p_{\Phi_\alpha}) + (p_{\Phi_\alpha} - q_{\Phi_\alpha})$$

By Lemma 5, since $\Theta_\alpha = \Phi_\alpha$ and $\alpha - 1 < K$, $p_{\Phi_\alpha} = d_{\Phi_\alpha}$ and $p_{\Phi_{\alpha-1}} = d_{\Phi_{\alpha-1}}$. Hence,

$$|q_{\Theta_{\alpha-1}} - p_{\Theta_{\alpha-1}}| + |q_{\Theta_\alpha} - q_{\Theta_{\alpha-1}}| \ge (d_{\Phi_{\alpha-1}} - d_{\Phi_\alpha}) + (d_{\Phi_\alpha} - q_{\Phi_\alpha})$$

Since Q is a valid RFPS for D,

$$q_{\Phi_\alpha} \le d_{\Phi_\alpha} < q_{\Phi_\alpha} + w.$$

Hence, $(d_{\Phi_\alpha} - q_{\Phi_\alpha}) \ge 0$. Thus,

$$|q_{\Theta_{\alpha-1}} - p_{\Theta_{\alpha-1}}| + |q_{\Theta_\alpha} - q_{\Theta_{\alpha-1}}| \ge d_{\Phi_{\alpha-1}} - d_{\Phi_\alpha}.$$

From the definition of Θ, since $\Theta_\alpha = \Phi_\alpha$, $d_{\Phi_{\alpha-1}} > d_{\Phi_\alpha}$. Hence,

$$d_{\Phi_{\alpha-1}} - d_{\Phi_\alpha} = |d_{\Phi_{\alpha-1}} - d_{\Phi_\alpha}|.$$

Thus,

$$|q_{\Theta_{\alpha-1}} - p_{\Theta_{\alpha-1}}| + |q_{\Theta_\alpha} - q_{\Theta_{\alpha-1}}| \ge |d_{\Phi_{\alpha-1}} - d_{\Phi_\alpha}|$$

Therefore,

$$\mathrm{MT}_Q[1, \Theta_\alpha] \ge \mathrm{MT}_P[1, \Theta_{\alpha-1}] + |d_{\Phi_{\alpha-1}} - d_{\Phi_\alpha}|$$

By Lemma 5, since $\Theta_\alpha = \Phi_\alpha$, $p_{m_\alpha} = d_{\Phi_\alpha}$. Hence,

$$\mathrm{MT}_P[1, \Theta_\alpha] = \mathrm{MT}_P[1, \Theta_{\alpha-1}] + |d_{\Phi_\alpha} - d_{\Phi_{\alpha-1}}|$$

Thus,

$$\mathrm{MT}_Q[1, \Theta_\alpha] \geq \mathrm{MT}_P[1, \Theta_\alpha].$$

Case B: $\Theta_\alpha = \Phi_\alpha$ and $\Theta_{\alpha-1} = \Psi_{\alpha-1}$:

$$\begin{aligned}|q_{\Theta_{\alpha-1}} - p_{\Theta_{\alpha-1}}| + |q_{\Theta_\alpha} - q_{\Theta_{\alpha-1}}| &= |q_{\Psi_{\alpha-1}} - p_{\Psi_{\alpha-1}}| + |q_{\Phi_\alpha} - q_{\Psi_{\alpha-1}}| \\ &= |p_{\Psi_{\alpha-1}} - q_{\Psi_{\alpha-1}}| + |q_{\Psi_{\alpha-1}} - q_{\Phi_\alpha}| \geq p_{\Psi_{\alpha-1}} - q_{\Psi_{\alpha-1}} + q_{\Psi_{\alpha-1}} - q_{\Phi_\alpha} = p_{\Psi_{\alpha-1}} - q_{\Phi_\alpha}.\end{aligned}$$

From the algorithm, $p_{\Psi_{\alpha-1}} = p_{\Phi_{\alpha-1}}$. Hence,

$$|q_{\Theta_{\alpha-1}} - p_{\Theta_{\alpha-1}}| + |q_{\Theta_\alpha} - q_{\Theta_{\alpha-1}}| \geq p_{\Phi_{\alpha-1}} - q_{\Phi_\alpha}.$$

The rest of the proof for this case is identical to the proof of Case A.

Case C: $\Theta_\alpha = \Psi_\alpha$ *and* $\Theta_{\alpha-1} = \Phi_{\alpha-1}$:

$$\begin{aligned}|q_{\Theta_{\alpha-1}} - p_{\Theta_{\alpha-1}}| + |q_{\Theta_\alpha} - q_{\Theta_{\alpha-1}}| &= |q_{\Phi_{\alpha-1}} - p_{\Phi_{\alpha-1}}| + |q_{\Psi_\alpha} - q_{\Phi_{\alpha-1}}| \\ &\geq q_{\Phi_{\alpha-1}} - p_{\Phi_{\alpha-1}} + q_{\Psi_\alpha} - q_{\Phi_{\alpha-1}} = q_{\Psi_\alpha} - p_{\Phi_{\alpha-1}} \\ &= (q_{\Psi_\alpha} - p_{\Phi_\alpha}) + (p_{\Phi_\alpha} - p_{\Phi_{\alpha-1}}).\end{aligned}$$

By Lemma 5, since $\Theta_\alpha = \Psi_\alpha$ and $\alpha - 1 < K$, $p_{\Phi_\alpha} = d_{\Psi_\alpha} - w + 1$ and $p_{\Phi_{\alpha-1}} = d_{\Phi_{\alpha-1}}$. Hence,

$$|q_{\Theta_{\alpha-1}} - p_{\Theta_{\alpha-1}}| + |q_{\Theta_\alpha} - q_{\Theta_{\alpha-1}}| \geq (q_{\Psi_\alpha} - d_{\Psi_\alpha} + w - 1) + (d_{\Psi_\alpha} - w + 1 - d_{\Phi_{\alpha-1}}).$$

Since Q is a valid RFPS for D,

$$q_{\Psi_\alpha} \leq d_{\Psi_\alpha} < q_{\Psi_\alpha} + w.$$

Hence, $q_{\Psi_\alpha} - d_{\Psi_\alpha} + w > 0$, so

$$q_{\Psi_\alpha} - d_{\Psi_\alpha} + w - 1 \geq 0.$$

Thus,

$$|q_{\Theta_{\alpha-1}} - p_{\Theta_{\alpha-1}}| + |q_{\Theta_\alpha} - q_{\Theta_{\alpha-1}}| \geq d_{\Psi_\alpha} - w + 1 - d_{\Phi_{\alpha-1}}.$$

From the definition of Θ, since $\Theta_\alpha = \Psi_\alpha$, $d_{\Phi_\alpha} > d_{\Phi_{\alpha-1}}$. From the algorithm,

$$\max(E_{\alpha-1}) - \min(E_{\alpha-1}) < w$$

while

$$\max(E_{\alpha-1} \cup \{d_{m_{\alpha-1}+1}\}) - \min(E_{\alpha-1} \cup \{d_{m_{\alpha-1}+1}\}) \geq w.$$

Hence, either $d_{m_{\alpha-1}+1} < d_{\Phi_{\alpha-1}}$ or $d_{m_{\alpha-1}+1} > d_{\Psi_{\alpha-1}}$. In this case, since $d_{\Phi_\alpha} > d_{\Phi_{\alpha-1}}$ and $d_{m_{\alpha-1}+1} \geq d_{\Phi_\alpha}$, it must be true that $d_{m_{\alpha-1}+1} > d_{\Psi_{\alpha-1}}$. By the definition of Ψ, $d_{\Psi_\alpha} \geq d_{m_{\alpha-1}+1}$. Hence,

$$d_{\Psi_\alpha} > d_{\Psi_{\alpha-1}}.$$

By Lemma 4, since $\alpha - 1 < K$, $d_{\Psi_{\alpha-1}} = d_{\Phi_{\alpha-1}} + w - 1$. Hence, $d_{\Psi_\alpha} > d_{\Phi_{\alpha-1}} + w - 1$, so

$$d_{\Psi_\alpha} - w + 1 - d_{\Phi_{\alpha-1}} > 0.$$

Thus,

$$d_{\Psi_\alpha} - w + 1 - d_{\Phi_{\alpha-1}} = |d_{\Psi_\alpha} - w + 1 - d_{\Phi_{\alpha-1}}|.$$

By Lemma 5, since $\Theta_\alpha = \Psi_\alpha$, $p_{m_\alpha} = d_{\Psi_\alpha} - w + 1$. Hence,

$$|q_{\Theta_{\alpha-1}} - p_{\Theta_{\alpha-1}}| + |q_{\Theta_\alpha} - q_{\Theta_{\alpha-1}}| \geq |p_{m_\alpha} - d_{\Phi_{\alpha-1}}|.$$

Therefore,

$$\mathrm{MT}_Q[1, \Theta_\alpha] \geq \mathrm{MT}_P[1, \Theta_{\alpha-1}] + |p_{m_\alpha} - d_{\Phi_{\alpha-1}}|.$$

It has been shown above that $\mathrm{MT}_P[1, \Theta_\alpha] = \mathrm{MT}_P[1, \Theta_{\alpha-1}] + |p_{m_\alpha} - d_{\Phi_{\alpha-1}}|$. Hence,

$$\mathrm{MT}_Q[1, \Theta_\alpha] \geq \mathrm{MT}_P[1, \Theta_\alpha].$$

Case D: $\Theta_\alpha = \Psi_\alpha$ *and* $\Theta_{\alpha-1} = \Psi_{\alpha-1}$:

$$\begin{aligned}|q_{\Theta_{\alpha-1}} - p_{\Theta_{\alpha-1}}| + |q_{\Theta_\alpha} - q_{\Theta_{\alpha-1}}| &= |q_{\Psi_{\alpha-1}} - p_{\Psi_{\alpha-1}}| + |q_{\Psi_\alpha} - q_{\Psi_{\alpha-1}}| \\ &\geq q_{\Psi_{\alpha-1}} - p_{\Psi_{\alpha-1}} + q_{\Psi_\alpha} - q_{\Psi_{\alpha-1}} = q_{\Psi_\alpha} - p_{\Psi_{\alpha-1}}.\end{aligned}$$

From the algorithm, $p_{\Psi_{\alpha-1}} = p_{\Phi_{\alpha-1}}$. Hence,

$$|q_{\Theta_{\alpha-1}} - p_{\Theta_{\alpha-1}}| + |q_{\Theta_\alpha} - q_{\Theta_{\alpha-1}}| \geq q_{\Psi_\alpha} - p_{\Phi_{\alpha-1}}.$$

The rest of the proof for this case is identical to the proof of Case C.

■

ACKNOWLEDGMENT

We would like to thank P. Denning, D. Ferrari, M. Katevenis, J. Ousterhout, R. Sherburne, and A. Smith for their useful suggestions on improving this paper and D. Patterson for his help with the development of some of the initial RISC analysis tools.

REFERENCES

[1] L. A. Belady, "A study of replacement algorithms for a virtual storage computer," *IBM Syst. J.*, vol. 5, no. 2, pp. 78-101, 1966.
[2] R. Campbell, "A C compiler for RISC," M.S. rep., Univ. California, Berkeley, Dec. 1980.
[3] P. J. Denning, private communication, May 1982.
[4] *VAX11 Architecture Handbook*, Digital Equipment Corp., 1979.
[5] D. Halbert and P. Kessler, "Windows of overlapping register frames," in *CS292R Final Project Reports* (unpublished), Univ. California, Berkeley, June 1980, pp. 82-100.
[6] S. C. Johnson, "A portable compiler: Theory and practice," in *Proc. 5th ACM Symp. Principles of Programming Languages*, Jan. 1978, pp. 97-104.
[7] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall, 1978.
[8] D. A. Patterson and C. H. Séquin, "RISC I: A reduced instruction set VLSI computer," in *Proc. 8th Annu. Symp. Comput. Architecture*, Minneapolis, MN, May 1981, pp. 443-457.
[9] ——, "A VLSI RISC," *Computer*, vol. 15, pp. 8-21, Sept. 1982.
[10] A. J. Smith, "Cache memories," *Comput. Surveys*, vol. 14, pp. 473-530, Sept. 1982.
[11] Y. Tamir, "Simulation and performance evaluation of the RISC architecture," Electron. Res. Lab., Univ. California, Berkeley, Memo. UCB/ERL, M81/17, Mar. 1981.

Yuval Tamir (S'78) received the B.S.E.E. degree ("with highest distinction") from the University of Iowa, Iowa City, in 1979 and the M.S. degree in electrical engineering and computer science from the University of California, Berkeley, in 1981.

Since 1979 he has been a Research Assistant in the Electronics Research Laboratory at U.C. Berkeley where he is currently working on his Ph.D. dissertation. His research interests are fault-tolerant computing, computer architecture, and distributed systems.

Mr. Tamir is a student member of the IEEE Computer Society and the Association for Computing Machinery.

Carlo H. Séquin (M'71-SM'80-F'82) received the Ph.D. degree in experimental physics from the University of Basel, Basel, Switzerland, in 1969.

In 1969-1970 he performed postdoctoral work at the Institute of Applied Physics, University of Basel, which concerned interface physics of MOS transistors and problems of applied electronics in the field of cybernetic models. From 1970 to 1976 he worked at Bell Laboratories, Murray Hill, NJ, in the MOS Integrated Circuit Laboratory on the design and investigation of charge-coupled devices for imaging and signal processing applications. He spent 1976-1977 on leave of absence with the University of California, Berkeley, where he lectured on integrated circuits, logic design, and microprocessors. In 1977 he joined the faculty in the Department of Electrical Engineering and Computer Sciences, where he is Professor of Computer Science. Since 1980 he has headed the CS Division as Associate Chairman for Computer Sciences. His research interests lie in the field of computer architecture and design tools for very large scale integrated systems. In particular, his research concerns multimicroprocessor computer networks, the mutual influence of advanced computer architectures and modern VLSI technology, and the implementation of special functions in silicon. Since 1977 he has been teaching courses in structured MOS-LSI design. He is an author of the first book on charge-transfer devices, and has written many papers in that field.

Dr. Séquin is a member of the Association for Computing Machinery and the Swiss Physical Society.

0018-9340/83/1100-0989$01.00