https://ntrs.nasa.gov/search.jsp?R=19980008790 2020-06-16T00:31:12+00:00Z

REPORT NO. UIUCDCS-R-87-1377

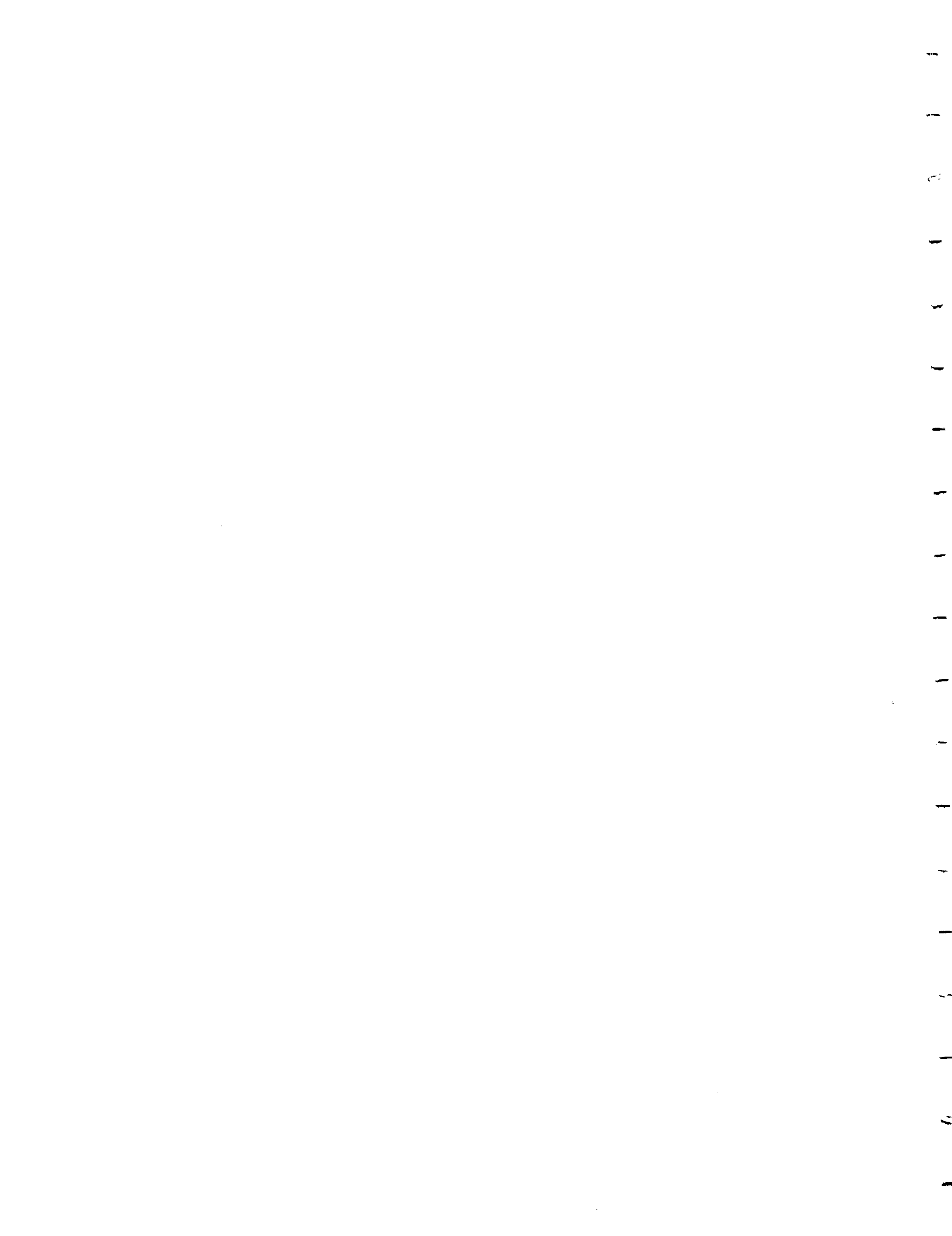
CONTEXT SWITCHING WITH MULTIPLE REGISTER WINDOWS: A RISC PERFORMANCE STUDY

by
Marian B. Konsek
Daniel A. Reed
Wittaya Watcharawittayakul

October 1987

DEPARTMENT OF COMPUTER SCIENCE
1304 W. SPRINGFIELD AVE.
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
URBANA, IL 61801

This work was supported inpart by NSF Grant Number DCR 84-17948.



Context Switching with Multiple Register Windows: A RISC Performance Study

Marian B. Konsek†
Complex Automation Systems Department
Polish Academy of Sciences
Gliwice, Poland

Daniel A. Reed‡
*Wittaya Watcharawittayakul**
Department of Computer Science
University of Illinois
Urbana, Illinois 61801

ABSTRACT

Although previous studies have shown that a large file of overlapping register windows can greatly reduce procedure call/return overhead, the effects of register windows in a multiprogramming environment are poorly understood. This paper investigates the performance of multiprogrammed, reduced instruction set computers (RISCs) as a function of window management strategy. Using an analytic model that reflects context switch and procedure call overheads, we analyze the performance of simple, linearly self-recursive programs. For more complex programs, we present the results of a simulation study. These studies show that a simple strategy that saves all windows prior to a context switch, but restores only a single window following a context switch, performs near optimally.

†Supported by a Senior Fulbright Scholarship while on leave at the University of Illinois at Urbana-Champaign.

‡Supported in part by NSF Grant Number DCR-84-17948.

*On leave from the Faculty of Applied Statistics, National Institute of Development Administration, Bangkok, Thailand

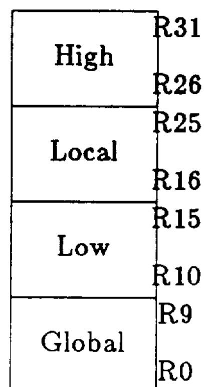
1. Introduction

Although a return to simple instruction sets was first advocated by John Cocke and later successfully realized in the IBM 801 [Radi82], the Stanford MIPS [Henn84], and UC–Berkeley RISC–II [Patt82, Kate84], the source and magnitude of reduced instruction set computer (RISC) performance increases have been surrounded by controversy. One of the major contributing factors to the debate has been the presence of a large register file in the RISC–II design. The portion of RISC–II's performance attributable to its register file has been contested [Hitc85] and has renewed discussions on register file design. Because this paper considers the management of RISC–II register files, we digress to briefly review their organization.

RISC–II Register Design

The UC–Berkeley RISC–II design [Patt82] provides each procedure invocation with a "window" of 32 registers; see Figure 1. The window associated with a called procedure partially overlaps both the window of the calling procedure (the "high" registers) and the window of the next procedure called (the "low" registers). Thus, the "high" registers contain the parameters passed from the caller, and the "low" registers are used to pass parameters to the next callee. On

Figure 1 RISC–II Register Organization



a procedure call, the "low" registers of the current window become the "high" registers of the callee's window. The "local" registers are, as the name implies, available for use by the procedure. Finally, the "global" registers are shared by *all* windows.

This overlapped register scheme reduces memory traffic in two important ways. First, rather than placing parameters on the stack prior to a procedure call, they can remain in registers. Second, by providing a sufficient number of overlapping register windows, the registers of the invoking procedure need not be saved prior to a procedure call. Of course, it is possible for the depth of the dynamic chain of procedure calls to exceed the number of register windows. In this case, some portion of the register file must be saved in memory to provide space for additional procedure invocations. Tamir [Tami83] has investigated strategies for solving this problem.

Overview

Although the RISC-II register file organization does reduce memory traffic due to procedure calls, its value is clouded by several pragmatic issues. First, the performance gains attributable to reduced procedure call overhead are lessened by the longer machine cycle time that results from capacitive loading of longer buses. Second, little is known about the behavior of multiple register windows in a multiprogramming environment, with its associated context switching. Certainly, the context switch overhead in a multiple register window architecture is greater than that in a single register set architecture, but it is not known if the performance gains due to reduced procedure call overhead are offset by larger context switch overheads.

In this paper we evaluate the performance of a RISC-II processor with multiple register windows in a multiprogramming environment. In section 2, three window management strategies are discussed. Section 3 presents an analytic model of register management. Finally, section 4 presents the results of a simulation study that confirms the results obtained from the

analytic models.

2. Register Management Strategies

As mentioned above, Tamir [Tami83] investigated RISC–II register window management strategies for execution of stand–alone programs. In RISC–II, register windows form a last–in–first–out (LIFO) buffer. On a procedure call, the processor allocates an adjacent, overlapping register window, provided one is available in the register file. Otherwise, a register file *overflow* occurs, and one or more windows are pushed to memory, freeing a window for the pending procedure call. On a procedure return, the processor switches to the previously active window. If this window is no longer in the register file, an *underflow* occurs, and one or more windows are restored from memory. Two pointers, a current window pointer *CWP* and a saved window pointer *SWP* are used to manage windows in the LIFO buffer and to recognize window overflows and underflows [Kate84]. Tamir [Tami83] showed that the simplest management strategy, namely saving the oldest window on overflow and restoring one window on underflow was nearly optimal. Therefore in the remainder of this paper, we assume this strategy is used.

Context Switching

When a processor is multiprogrammed, the process associated with each program is suspended and resumed many times before completion. The operating system must preserve the state of the process at the end of each time slice. If the processor contains a single register set, this preservation typically entails copying the contents of all registers to memory. If the processor has many register windows, the context switch overhead includes, in principle, saving *all* active register windows. For a machine like RISC–II, this cost can be large. Fortunately, there are several alternative register management schemes, and some avoid saving all registers.

Suppose a process occupies n register windows at the time of a context switch. In general, let

$$\textit{Strategy}\ (k,\ j) \qquad 1 \leq k \leq n \quad 1 \leq j \leq N$$

denote saving k windows and restoring j windows, where N is the current depth of procedure calls. Note that N can be greater than n if windows have been saved on the window stack in memory. Several strategies of this form are possible. In this paper, we consider three: *Strategy (n, n)*, *Strategy (n, 1)*, and *Strategy (0, 1)*.

Strategy (n, n)

The obvious extension of context switching to a multiple window register file simply saves all active windows of the current process prior to context switching and restores those same register windows when the process receives its next time slice. Because the complete state of each process is restored prior to its time slice, the probability of register window underflow or overflow is independent of the multiprogramming mix.

Empirical data suggest that most programs exhibit nesting depth locality. Specifically, the dynamic depth of procedure calls changes only a small amount over long periods of time, even if the maximal chain of dynamic calls is high.[1] Indeed, this is the primary reason a small set of register windows on RISC–II can cache most sub–sequences of calls [Patt85]. However, because nesting depth shows only a small variation with time, the register file is likely to contain many register windows that will not be used by the process until far in the future. By analogy with virtual memory, the register file contains more windows than those constituting the "working set" of the process. As a result, *Strategy (n, n)* will often restore windows that will not be used before the next context switch.

[1]This does not mean that there are few procedure calls, merely that the depth of calls changes little.

Strategy (n, 1)

Rather than restoring all windows, we might restore only the window corresponding to the currently active procedure. This reduces the cost of a context switch and, because each process resumes with more free windows, also reduces the probability of register file overflow. However, because only one window is restored, more register file underflows will occur than if the process ran by itself. Suppose that a process needs all windows saved during the previous context switch. Underflows will cause these windows to be restored singly, and the total cost will be greater than if they were restored enmasse. Certainly, the size and number of windows will determine whether this difference is important. Table 2 shows that the context switching cost for saving n RISC–II windows takes the form $an + b$ and that bulk restores *are* more efficient.

Unlike *Strategy (n, n)*, the length of the time slice interacts with *Strategy (n, 1)* to *change* the overflow and underflow probabilities. *Strategy (n, 1)* has lower context switching cost, but potentially higher procedure underflow costs. Because the efficacy of the two strategies depends on the number of windows in the register file, the dynamic chain of procedure calls, and the number and size of registers windows, it is difficult to predict *a priori* their relative performance.

Strategy (0, 1)

The two strategies proposed above save *all* active register windows. Clearly, context switching overhead is minimized if *no* windows are automatically saved: Instead, windows can be saved as needed. This approach is similar to that used with caches. That is, register windows remain in the register file until their space is needed. If no intervening process needs the space, a process may find that some of its register windows are still in the register file at the beginning of its next time slice.

With this strategy, a window overflow or underflow trap procedure must be able to determine the owner of each register window. Therefore, a process identifier register and an

occupancy flag must be associated with each window, and procedure calls must load these registers appropriately. Finally, the order of a process' windows in memory must be preserved. If the youngest window belonging to a process is saved before older windows, space in the memory stack must be reserved for those intervening windows.

When a process resumes execution after a context switch, its register windows may appear in several possible states.

(1) No windows belonging to the resumed process are in the register file and either

(a) no free windows exist, or

(b) at least one free window exists.

(2) At least one window belonging to the resumed process is in the register file and either

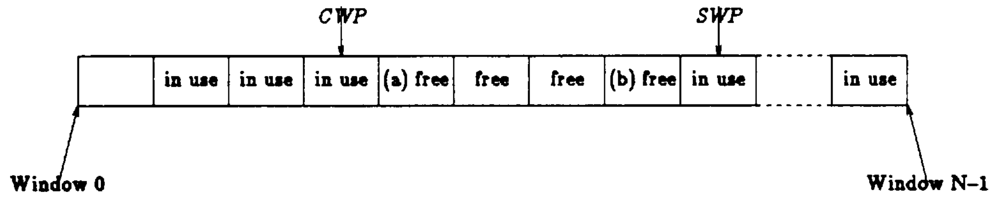
(a) the process' most recently active window is in the register file, or

(b) the process' most recently active window is *not* in the register file.

The necessary action differs in each case.

In cases (1a) and (1b), the window belonging to the active procedure of the process must be restored from the top of the corresponding memory stack. There are several possibilities for its placement in the register file. If the process that just relinquished the processor left free windows (i.e., the process could have executed another procedure call without window overflow) one of these can be allocated. However, even this poses alternatives. As Figure 2 shows, it is possible to restore the process window to the free window following the one pointed to by the *CWP* (current window pointer) of the previous process, window (a). This permits the maximum number of calls before a window must be written to memory. Alternatively, a free window just before the one pointed to by the *SWP* of the previous process can be allocated, window (b) in Figure 2, giving preference to returns. As a compromise, a free window between the two pointers would give equal preference to both calls and returns. These choices will determine how windows in the

Figure 2 *Strategy (0, 1)* Window Management



Call:
$CWP := (CWP + 1) \bmod N$
if $CWP = SWP$ then ***overflow***

Return:
$CWP := (CWP - 1) \bmod N$
if $CWP = SWP$ then *underflow*

register file are populated.

If the process active during the previous time slice did not leave any free windows (i.e., another call would have caused an overflow), two possibilities exist. Either the entire register file is full, or there exists at least one free window in the register file that is not in the contiguous region between *CWP* and *SWP* used by the previous process. In the first case, one or more windows should be saved in memory. In the second case, a free window must be located. The "placement" of the resumed process window in the register file is analogous to the cache replacement strategies [Smit82]. Like those strategies, it must be fast and efficient.

The preceding discussion concerned only cases (1a) and (1b), when no windows belonging to the process remained in the register file at the beginning of its time slice. If the register file is large, or the multiprogramming level is low, some windows belonging to the process may remain in the register file. This is analogous to a "warm start" in a cache [Smit82]. However, window restoration can be completely avoided only if the topmost portion of the window stack belonging to the process still resides in the register file. Otherwise, either the portion of the window stack still in the register file must be augmented with those windows in memory; *or* the portion of the

window stack in the register file must be saved in memory, and the topmost window of the stack loaded from memory. These alternatives are necessary if the stack structure of the windows is to be maintained in the register file.

Clearly, there are many possible implementations of *Strategy (0, 1)*. The similarities with caches are obvious, although subtle differences exist, primarily because the *order* of windows in the register file reflects the call/return sequences of processes. Window replacement policies must maintain this order. Finally, additional hardware is needed for *Strategy (0, 1)* implementation; for details on the hardware requirements, see [Watc87].

Space precludes a complete analysis of *Strategy (0, 1)*. Hence, in this paper, we assume that *Strategy (0, 1)* must maintain a contiguous group of register windows in both the register file and associated memory stack of each process. Moreover, if any windows belonging to a process remain in the register file, we require that the most recently used window also remain in the register file. Saving this window in memory forces the saving of all other windows belonging to the process. Thus, the most recently used segment of windows belonging to a process remains in the register file. If a process regains control of the processor and finds that its current window is missing, the first group of free windows, beginning at window zero, is used to allocate a window for the process. This window is located at the middle of the free group, giving equal preference to procedure calls and returns. Finally, if no free windows exist, the least recently used window of the process relinquishing the processor is replaced. This maximizes the time until the replaced window is needed.

In the next section, we formalize the interdependency of context switching and window underflow/overflow as an optimization problem and show how it can be analyzed for simple programs. Following that, we compare the performance of the context switching strategies just described using trace driven simulations.

3. Analytic Models of Register Management

As we have just seen, the window management strategy used for context switching can affect the register file overflow and underflow probabilities. Moreover, increasing (decreasing) the size of the register file will decrease (increase) overflow and underflow probabilities while increasing (decreasing) the context switching cost. Abstractly, however, the execution time of a program, measured in machine cycles,[2] depends on the number of program instructions executed, the window management cost for procedure calls, and context switching overhead. Selecting a window management strategy is then reduced to the following optimization problem,

$$\underset{\forall P \in MPSet}{minimize}\ ExecutionTime(P,\ W,\ Ts) \tag{1}$$

$$subject\ to \quad 1 \leq W \leq W_{\max}$$

$$1 \leq Ts \leq Ts_{\max}$$

where P is a program in the multiprogramming set $MPSet$, W is the number of windows in the register file, and Ts is the time slice. The execution time, in turn, is given by

$$ExecutionTime(P,\ W,\ Ts) = Instructions(P) + Context(P,\ W,\ Ts) + Overflow(P,\ W,\ Ts) \tag{2}$$

where *Instructions (P)* is the execution time of a program without procedure and context switch overhead, $Context(P,\ W,\ Ts)$ is the cost of context switching, and $Overflow(P,\ W,\ Ts)$ is the cost of window management during procedure call and return.

Because this optimization problem depends on the multiprogramming mix and the interaction of programs with the context switching strategy, there is little prospect of solving the

[2]Because all RISC-II instructions other than load or store execute in a single cycle [Kate84], modeling program execution time is straightforward.

general case. However, for many interesting cases, closed form solutions are possible. Although these solutions might initially appear to be of marginal value, they provide insight into the interaction of the parameters. The following section analyzes the performance of self–recursive programs as a function of time slice; see [Watc87] for an extension to other program classes.

Self–Recursive Programs

Consider a class of programs of self–recursive programs where, at any depth, the probability of an additional call is p. Then the distribution of procedure depths is binomial, and the expected depth for any execution of the program is

$$\overline{D} = \frac{1}{1-p}. \tag{3}$$

Let T be the execution time of each procedure, and, for simplicity's sake, let each procedure call occur at the point $\frac{T}{2}$. That is, each procedure invocation executes for $\frac{T}{2}$ time units, recursively invokes itself, and following the return of the recursive call, executes for an additional $\frac{T}{2}$ time units. Then the mean program execution time, exclusive of procedure call and context switching overhead, is

$$\mathit{Instructions}(P) = T\overline{D}. \tag{4}$$

Now consider *Strategy (n, n)* that saves and restores the complete context of each process. Because the program state is unchanged after each context switch, the procedure call overhead is independent of the time slice. If the depth of procedure calls $\overline{D}$ is less than the number of register windows W, there is no procedure call overhead. Otherwise, each call of depth greater than $\overline{D}$ causes both a window overflow and underflow. Hence, the overflow cost is

$$\mathit{Overflow}(P,\ W,\ Ts) = \begin{cases} 0 & \overline{D} \le W \\ S\left[\overline{D} - W\right] & \overline{D} > W, \end{cases} \tag{5}$$

where S is the cost to save and restore a single register window.

Finally, the time slice Ts can be either smaller

$$Ts = \frac{T}{2k} \qquad k = 1, 2, 3, \cdots$$

or larger

$$Ts = \frac{Tk}{2} \qquad k = 1, 2, 3, \cdots$$

than $\frac{T}{2}$. In the first case, each procedure call suffers multiple context switches. Conversely, there are several procedure calls per time slice in the second case. We consider the two cases separately.

Case $Ts = \frac{T}{2k}$:

As Figure 3a shows, the procedure at depth d suffers context switches with d windows in the register file, both *before* and *after* executing its recursive call. Thus, the context switching cost for *Strategy (n, n)* is

$$Context(P,\ W,\ Ts) \ = \ 2kS\sum_{d=1}^{\overline{D}} d \ = \ kS\overline{D}\left[\overline{D} + 1\right] \tag{6}$$

if $\overline{D} \leq W$. Recall that k is the number of context switches per procedure invocation, and S is the cost to save and restore one window.

Similarly, if the mean depth of calls $\overline{D}$ exceeds the number of windows W, Figure 3b shows that the $\overline{D} - W$ procedure invocations that overflow the register file suffer context switching cost $kWS(\overline{D} - W)$ *before* their recursive calls and cost $kS(\overline{D} - W)$ *after* their recursive calls return. Why? On the downward chain of calls, the register file fills, and each context switch must save the entire register file of W windows. On the upward chain of

returns, the register file empties, and each context switch saves only a single window. Thus, the context switching cost is

$$Context(P,\ W,\ Ts) \ = \ 2kS\sum_{d=1}^{W} d \ + \ kWS\left[\overline{D} - W\right] \ + \ kS\left[\overline{D} - W\right] \tag{7}$$

$$= \ kS\overline{D}\left[W + 1\right]$$

if $\overline{D} > W$.

Case $Ts = \frac{Tk}{2}$:

This case is similar to the previous one except that the context switching interval exceeds T, the procedure execution time. Thus, successive context switches see the number of allocated register windows grow by increments of k. The number of context switches on the downward chain of calls is $\left\lfloor \frac{\overline{D}}{k} \right\rfloor$ if $\overline{D} \le W$, and the context switching cost is

$$Context(P,\ W,\ Ts) \ = \ S\overline{D}\left[1 + \left\lfloor \frac{\overline{D}}{k} \right\rfloor\right]. \tag{8}$$

Similarly, if $\overline{D} > W$, the context switching cost is

$$Context(P,\ W,\ Ts) \ = \ S\left[W\left[\left\lfloor \frac{W}{k} \right\rfloor + \left\lfloor \frac{\overline{D} - W}{k} \right\rfloor + 1\right] + \left\lfloor \frac{\overline{D} - W}{k} \right\rfloor\right]; \tag{9}$$

see [Watc87] for a complete derivation of these formulae.

Inspecting these equations shows that the overflow cost (5) is a linearly decreasing function of the number of windows W in the register file. Similarly, the context switching cost equations (7) and (9) are linearly *increasing* functions of W. If $aW + b$ denotes the overflow cost, and $cW + d$ denotes the context switching cost[3], the linear combination $(a + b)W + (c + d)$ can

have either positive or negative slope. If the slope is positive[3] the total overhead will *increase* with the number of register windows. Conversely, with negative slope, overhead is minimized with a small number of windows in the register file. The optimal choice depends on the time slice, cost of register saves and restores, and the depth of procedure calls.

Figure 4 illustrates one combination of values based on the derivation just presented. Similarly, Figure 5 shows the interaction of context switching cost and procedure overflow cost on an actual, linearly self-recursive program, *factorial*, when time sliced on RISC-II with varying sized register files and the register management costs given in Table 2. The critical dependence of *Strategy (n, n)* on so many parameters suggests that it is inappropriate for a multiprogramming environment.

Analysis of program behavior is *not* restricted to linearly, self-recursive programs nor to just *Strategy (n, n)*. The technique has been applied to programs whose call probability is a function of depth and to programs with richer patterns of call behavior (e.g., trees). Moreover, other window management strategies, including *Strategy (n, 1)* and variations of *Strategy (0, 1)* are amenable to this technique [Watc87].

4. A Simulation Study of Register Management

Although the analysis in the previous section does provide insight into the behavior of certain program classes, it cannot be used to precisely predict the performance of real program mixes. For this, trace driven simulation is needed.

Selection of benchmarks for trace driven evaluation is always difficult. The desire for generality must be balanced against the cost of simulating many program traces. Reducing the number of benchmarks to reduce simulation costs means that the remaining benchmarks must

[3]Table 2 shows just such cost functions for RISC-II.

reflect "typical" behavior. Moreover, continuity with other studies [Patt82, Hitc85] is necessary to maintain a standard of reference.

The simulation experiments reported below were based on nine benchmarks. The first six are those used during the RISC–II evaluation and permit comparison with previously reported studies [Patt82, Tami83, Hitc85]. Because many of these programs have been criticized as procedure–call intensive, the set was augmented with three other programs: the Dhrystone synthetic benchmark [Weic84], the UC–Berkeley RISC–II simulator (*Rsim*) executing the Fibonacci program, and the *sed* editor editing a 500–line UNIX manual.

Table 1 shows the characteristics of these benchmarks when executed stand–alone on RISC–II with 8 register windows. The call/return instruction frequencies and call/return memory traffic shown in Table 1 include instructions for saving and restoring both local registers and environment registers (e.g., program counter and stack pointer). The maximal procedure

Table 1 Benchmark characteristics

Benchmark	*Call/return frequency[%]*	*Call/return memory traffic[%]*	*MND*[1]	*ACS*[2]	*WRR*[3]	*ARM*[4]
Ackerman	17.4	49.2	512	2.01	6	3.5±1.5
Fibonacci	21.9	42.9	21	2.00	4	2.3±0.8
Hanoi	16.7	48.7	20	2.00	10	3.5±2.5
Puzpnt	0.8	6.6	19	1.21	11	4.5±1.2
Puzsub	0.7	3.3	19	1.10	10	2.3±1.0
Qsort	9.7	27.3	10	1.01	15	2.6±2.3
Dhrystone	8.6	22.4	5	1.25	12	3.8±1.5
Rsim	0.8	2.6	6	1.06	12	2.2±0.7
Sed	1.4	5.9	7	1.69	11	4.7±1.4

[1]*MND* – maximal nesting depth
[2]*ACS* – average length of call sequences
[3]*WRR* – number of registers in a window referenced in the benchmark
[4]*ARM* – average number of registers modified in a procedure

nesting depth *MND* is the minimum number of register windows sufficient to avoid both register window overflow and underflow. Similarly, *ACS* is the average number of sequential procedure calls before a return.

Comparison of the benchmark set characteristics with those for other workloads [Clar82, Wiec82, Emer84] shows that at least three program classes were included in the set:

- *procedure intensive* with a greater than normal frequency of procedure calls (*Ackerman, Fibonacci, and Hanoi*),
- *procedure typical* with average procedure call frequency (*Dhrystone, Qsort*), and
- *procedure parsimonious* with minimal procedure call frequency (*Puzpnt, Puzsub, Rsim, Sed*).

With exception of the *Ackerman* benchmark, the dynamic pattern of procedure nesting depth confirms the locality of procedure nesting.

Simulated Multiprogramming

Like cache performance studies [Smit82], the performance of RISC–II context switching strategies depends on the multiprogramming mix and the process scheduling algorithm. The experiments presented below were based on a simple, round–robin scheduling algorithm with a fixed time slice. As defined, *Strategy (n, n)* and *Strategy (n, 1)* are independent of the *mix* of programs, only the length of the time slice is important. For these two strategies, it suffices to simulate programs singly and calculate the context switching cost at fixed multiples of the time slice.

The performance of *Strategy (0, 1)* does depend on the mix of programs. Thus, it was necessary to capture program instruction traces and simulate context switches among the traces [Kons86]. In all cases, a multiprogramming level of three was used. Among the three program classes discussed earlier, three mixes were created.

Mix 1 is a combination of programs from each of the three classes, procedure intensive, typical, and parsimonious. Fibonacci, Dhrystone and Puzpnt are used in this mix.

Mix 2 is a group of homogeneous programs drawn from the same class. Three combinations are possible here. First, Fibonacci (two copies) and Hanoi constitute a mix of procedure intensive programs. Second, Dhrystone (two copies) and Qsort are a "typical" mix. Finally, Puzpnt (two copies) and Sed constitute a mix of procedure parsimonious programs.

Mix 3 is similar to the first mix, except that Fibonacci was replaced by the Ackerman program. Ackerman's absence of procedure nesting locality can degrade the performance of the *entire* multiprogramming mix [Watc87].

The choice of appropriate time slices for simulations is a difficult problem, because it depends on the hardware/software environment. For the VAX-11/780 with VMS, the average time slice has been measured to vary between 1,812 and 9,729 instructions [Emer84, Clar85]. To cover a range of possibilities, we repeated all experiments for the following time slices, measured in cycles: 500, 1000, 1500, 5000, 10000, and 20,000.

Performance Measurement

Procedure call and return overhead was calculated using the product of the number of window overflows and underflows and the execution time of the trap procedure servicing these events. Similarly, the context switch overhead was assumed to be the product of the number of context switches and the execution time of context switching algorithm. Table 2 shows these costs, obtained from an analysis of the RISC-II assembly code for each operation.

As stated before, the execution times of the benchmarks were used as a measure of the RISC performance. The procedure and context switch overheads, $Overflow(P, W, Ts)$ and $Context(P, W, Ts)$ in equation (2), were monitored separately to study behavior in two

Table 2 Procedure and context switch overhead (cycles)

Strategy	*Window Overflow*	*Window Underflow*	*Context Save*	*Context Restore*
(n,n)	54	57	37 + n×43[1]	27 + n×45[1]
(n,1)	54	57	37 + n×43[1]	67
(0,1)	54	57 or 94 + n×43[2]	41	100,137 or 187[2]

[1]n – number of active windows.
[2]implementation strategy dependent

execution environments: stand–alone mode and in multiprogramming mode. The ratios of these overheads to the program time (i.e., *Context/Instructions*, *Overflow/Instructions*, and *(Overflow + Context)/Instructions)* were used as performance metrics. Note that *Instructions* is the optimal performance, the execution time without procedure and context switch overhead (i.e., for the infinite number of windows and no context switching).

4.1. Stand–alone Program Execution

Figures 6 and 7 show the procedure overhead for selected members of the benchmark set as a function of the number of windows in stand–alone mode. For most benchmarks, the procedure overhead becomes negligible long before the number of windows approaches the benchmark's maximal nesting depth. For those benchmarks with parsimonious or typical procedure call frequencies, four windows suffice to reduce the procedure overhead to less than 2 percent of the program execution time. For highly recursive benchmarks such as Fibonacci and Hanoi, the procedure overhead is less than 6 percent when the the number of windows exceeds 10. Only the Ackerman benchmark shows anomalous behavior. With a maximum procedure nesting depth of 512 and little locality in the pattern of procedure calls, the Ackerman benchmark benefits little from multiple register windows. This produces the very high procedure call overhead.

4.2. Execution in a Multiprogramming System

Space precludes a complete presentation of the simulation results; see [Watc87] for details. Hence, we concentrate on two of the three program classes, *procedure typical* and *procedure intensive* using the Dhrystone and Fibonacci benchmarks as representatives; other benchmarks yield similar results. In all cases, we show the overhead for procedure calls and context switching as a function of the program execution time in stand–alone mode with an infinite number of register windows.

Strategy (n, n)

Figures 8 and 9 show that, for each context switching interval, there is an optimal number of windows. As the number of windows in the register file increases, the probability of window overflow decreases. Simultaneously, the cost of each context switch increases. These two trends, one increasing cost, the other decreasing cost, yield an optimal number of windows for a given context switching interval. This is in apparent contrast to the analytic results obtain earlier. Recall, however, that the Fibonacci benchmark is *not* linearly recursive. Instead, its pattern of calls (i.e., Fibonacci (n) = Fibonacci (n–1) + Fibonacci (n–2)) form a tree of procedure call depths. This is illustrated in Figure 10. This behavior yields a quadratic cost function for overhead; whence the minima in Figure 9.

Table 3 shows that the optimal number of windows for each benchmark is *not* a constant; it depends on the time slice. For small time slices, it is more important to minimize the number of register windows because these windows must be saved frequently. As the time slice increases, procedure overflow and underflow overheads dominate, favoring use of additional windows.

Two final points about Figures 8 and 9 should be noted. First, the optimal number of register windows depends on the program type. For programs with modest procedure call depth,

Table 3 Optimal number of windows for *Strategy (n, n)*

Benchmark	*Time slice (cycles)*					
	0.5K	*1K*	*1.5K*	*5K*	*10K*	*20K*
Fibonacci	**9**	**9**	**11**	**13**	**13**	**15**
Hanoi	**7**	**7**	**9**	**9**	**11**	**11**
Puzpnt	**2**	**2**	**3**	**3**	**4**	**5**
Puzsub	**2**	**2**	**2**	**2**	**3**	**4**
Qsort	**2**	**2**	**2**	**4**	**4**	**5**
Dhrystone	**4**	**4**	**4**	**4**	**4**	**4**
Rsim	**2**	**3**	**3**	**3**	**3**	**3**
Sed	3	3	3	5	5	5

a small number of register windows is best. Using too many windows retains portions of the dynamic call chain that are not in the "working set" of windows, resulting in excessive context switching overhead. Likewise, using too few windows causes "window thrashing." The sensitivity of programs to the number of windows is striking, as the Dhrystone benchmark illustrates. In contrast, highly recursive programs like Fibonacci have a large window "working set" and need more windows. Second, the absence of a single register set size that minimizes execution time for all programs suggests that *Strategy (n, n)* is a poor candidate for register window management in a multiprogramming environment.

Strategy (n, 1)

As Figures 11 and 12 show, restoring a single window following a context switch greatly reduces the overhead, compared to *Strategy (n, n)*. For all classes of benchmarks, the overhead approaches an asymptote as the number of register windows grows. Table 4 shows the number of register windows that yields execution time within 1 percent of the minimal execution time achievable with a infinite number of windows. A comparison with Table 3 shows that the values

Table 4. Optimal number of register windows under *Strategy(n,1)*

Benchmark	*Time slice values (cycles)*					
	0.5K	*1K*	*1.5K*	*5K*	*10K*	*20K*
Fibonacci	11	13	11	13	13	13
Hanoi	11	13	11	9	11	11
Puzpnt	3	3	3	3	3	3
Puzsub	2	2	2	2	2	2
Qsort	2	3	3	3	3	3
Dhrystone	4	4	4	4	4	4
Rsim	2	3	3	3	3	3
Sed	4	4	4	4	4	5

in Table 4 are slightly larger. Recall that *Strategy (n, 1)* restores only a single register after a context switch. Thus, the mean number of windows a process can maintain in the register file is, for a fixed size register file, smaller for *Strategy (n, 1)* than for *Strategy (n, n)*. This favors a slightly larger register file for *Strategy (n, 1)*.

Because the performance of *Strategy (n, 1)* is monotonic in the size of the register file, it is a promising candidate for a multiprogramming environment. A register file large enough to accommodate highly recursive programs is also optimal for procedure parsimonious programs.

Strategy (0, 1)

Figure 13 shows the procedure and context switch overhead under *Strategy(0,1)* for both mix 1, a mixture of program types (Fibonacci, Dhrystone, and Puzpnt), and mix 2, a homogeneous program group (Dhrystone, Dhrystone, and Qsort). Comparing Figure 13 to Figure 11 shows that, within the range of 2 to 16 windows, *Strategy (0, 1)* is generally inferior to *Strategy (n, 1)*. There are two principal reasons for this performance gap.

First, recall that *Strategy(0,1)* can potentially find the most recently used window of the process *still* in the register file, a "window hit." However, detailed examination of the simulations

showed that, in mix 1, the hit ratio for one program was less than 10 percent and did not exceed 60 percent for any program in the mix. Similarly, the hit ratio ranged from 10 to 80 percent for the programs in mix 2. When the most recently used window is not in the register file, the window underflow trap procedure must search for free windows in the register file. In most cases the register file was full, leading to large overheads.

Second, because the register file utilization is so high, processes compete for free windows. In other words, the overall performance is degraded by interference among processes. This competition for windows can result in anomalies for *certain* processes in jobs mixes (i.e., a larger register file can actually increase the overhead); see Figure 13a. The effects of competition are most pronounced for small time slices. Each process spends a large portion of its time slice fetching register windows from memory.

To overcome the window management overhead and the interference effect, *Strategy (0, 1)* requires a larger register file. This will increase the hit ratio and increase the window allocation for each process. Figure 14 shows the performance of *Strategy(0,1)* for 16 to 80 windows on the Dhrystone benchmark. For a a large enough register file, the hit ratio approaches 100 percent. Table 5 shows the overhead ratios for *Strategy (0, 1)* and *Strategy (n, 1)* with an infinite number of windows. For programs with typical procedure call patterns (e.g., Dhrystone), *Strategy (n, 1)*

Table 5 Ratio of overheads *Strategy (n, 1):Strategy (0, 1)*

Slice	*Mix 1*			*Mix 2*		
	Dhrystone	*Puzpnt*	*Fibonacci*	*Dhrystone*	*Puzpnt*	*Fibonacci*
1.5K	6.67	6.92	20.21	7.04	6.92	20.21
5K	4.41	7.38	17.96	6.17	7.76	17.96
10K	4.15	7.94	14.97	5.96	8.35	14.97
20K	3.17	8.57	13.37	4.86	9.04	13.37

has roughly 5 times the overhead of *Strategy (0, 1)*. For heavily recursive programs, the asymptotic overhead ratio approaches 20. This is tempered by the knowledge that the absolute overhead of both schemes is relatively small for a sufficient number of windows. In this light, Table 6 shows the number of windows necessary for *Strategy (0, 1)* to yield lower overhead than *Strategy (n, 1)*. As can be seen, this depends heavily on the program mix.

The effects of the program mix on the performance of *Strategy (0, 1)* and the variation in size of the register file necessary to optimize performance suggest that *Strategy (n, 1)* is likely preferable. However, *Strategy (0, 1)* should be investigated further.

5. Conclusions

We have presented three window management strategies for a multiprogrammed RISC–II processor. The simplest strategy saves all active windows belonging to a process at the end of its time slice. Upon resumption, all windows are restored. Although this technique, *Strategy (n, n)*, requires no modification to the existing RISC–II hardware, we showed via analytic models that the optimal size of the register file depends on the context switch interval and the pattern of procedure calls. This was confirmed via trace driven simulation. This suggests that this strategy is inappropriate for a multiprogrammed environment.

Table 6
Number of windows where
Strategy (0, 1) is preferable to *Strategy (n, 1)*

Slice	*Mix 1*			*Mix 2*		
	Dhrystone	*Puzpnt*	*Fibonacci*	*Dhrystone*	*Puzpnt*	*Fibonacci*
1.5K	20	20	12	10	10	28
5K	24	24	12	10	12	32
10K	28	24	14	12	14	32
20K	28	28	14	12	16	40

The second approach, *Strategy (n, 1)*, saves all active windows upon a context switch, but *restores* only one. Simulations showed that it is uniformly superior to the first strategy. Moreover, the context switching overhead decreased asymptotically with larger register files. As before, no modification to existing hardware is necessary. This suggests that a single, large register file can provide good performance in a multiprogrammed environment.

The final technique, *Strategy (0, 1)*, treats the register file as a cache, saving windows only when their space is needed. The performance of this strategy is sensitive to the mix of programs, unlike either of the other strategies. Although a larger register file is necessary to achieve good performance, this strategy is asymptotically superior to either of the two strategies that save the entire context each time slice. As we noted at the outset, there are many variations of *Strategy (0, 1)*, based on the window replacement algorithms used. Further experimentation is needed to determine if the hardware costs of this approach are offset by increased performance.

References

[Clar82] D. W. Clark and H. M. Levy, "Measurement and Analysis of Instruction Use in the VAX-11/780," *Proceedings of the Ninth Annual Symposium on Computer Architecture,* ACM SIGARCH Computer Architecture News (April 1982), Vol. 10, No. 3, pp. 9–17.

[Clar85] D. W. Clark and J. S. Emer, "Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement," *ACM Transactions on Computer Systems* (February 1985), Vol. 3, No. 1, pp. 31–62.

[Emer84] J. S. Emer and D. W. Clark, "A Characterization of Processor Performance in the VAX-11/780," *Proceedings of the Eleventh International Symposium on Computer Architecture,* ACM SIGARCH Computer Architecture News (June 1984), Vol. 12, No. 3, pp. 301–310.

[Henn84] J. L. Hennessy, "VLSI Processor Architecture," *IEEE Transactions on Computers* (December 1984), Vol. C–33, No. 12, pp. 1221–1246.

[Hitc85] C. Y. Hitchcock III and H. M. Brinkley Sprunt, "Analyzing Multiple Register Sets," *Proceedings of the Twelfth International Symposium on Computer Architecture,* ACM SIGARCH Computer Architecture News (June 1985), Vol. 13, No. 3, pp. 55–63.

[Kate84] M. G. Katevenis, *Reduced Instruction Set Computer Architecture for VLSI,* The MIT Press, 1984.

[Kons86] M. B. Konsek and D. A. Reed, "ISpy: An Instruction Set Analysis Tool," Department of Computer Science, University of Illinois at Urbana–Champaign, Technical Report No. UIUCDCS–R–86–1276, (May 1986).

[Patt82] D. A. Patterson and C. H. Sequin, "A VLSI RISC," *IEEE Computer* (September 1982), Vol. 15, No. 9, pp. 8–21.

[Patt85] D. A. Patterson, "Reduced Instruction Set Computers," *Communications of the ACM,* (January 1985), Vol. 28, No. 1, pp. 8–21.

[Radi82] G. Radin, "The 801 Minicomputer," *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems,* ACM SIGARCH Computer Architecture News (March 1982), Vol. 10, No. 2, pp. 39–47.

[Smit82] A. J. Smith, "Cache Memories," *Computing Surveys* (September 1982), Vol. 14, No. 3, pp. 473–530.

[Tami83] Y. Tamir and C. H. Sequin, "Strategies for Managing the Register File in RISC," *IEEE Transactions on Computers* (November 1983), Vol. C–32, No. 11, pp. 977–988.

[Wiec82] C. A. Wiecek, "A Case Study of VAX-11 Instruction Set Usage for Compiler Execution," *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems,* ACM SIGARCH Computer Architecture News (March 1982), Vol. 10, No. 2, pp. 177–184.

[Weic84] R. P. Weicker, "Dhrystone: A Synthetic Systems Programming Benchmark," *Communications of the ACM* (October 1984), Vol. 27, No. 10, pp. 1013–1030.

[Watc87] W. Watcharawittayakul, "Managing Large Register Files," *Ph.D. Dissertation,* Department of Computer Science, University of Illinois at Urbana–Champaign, *in preparation.*

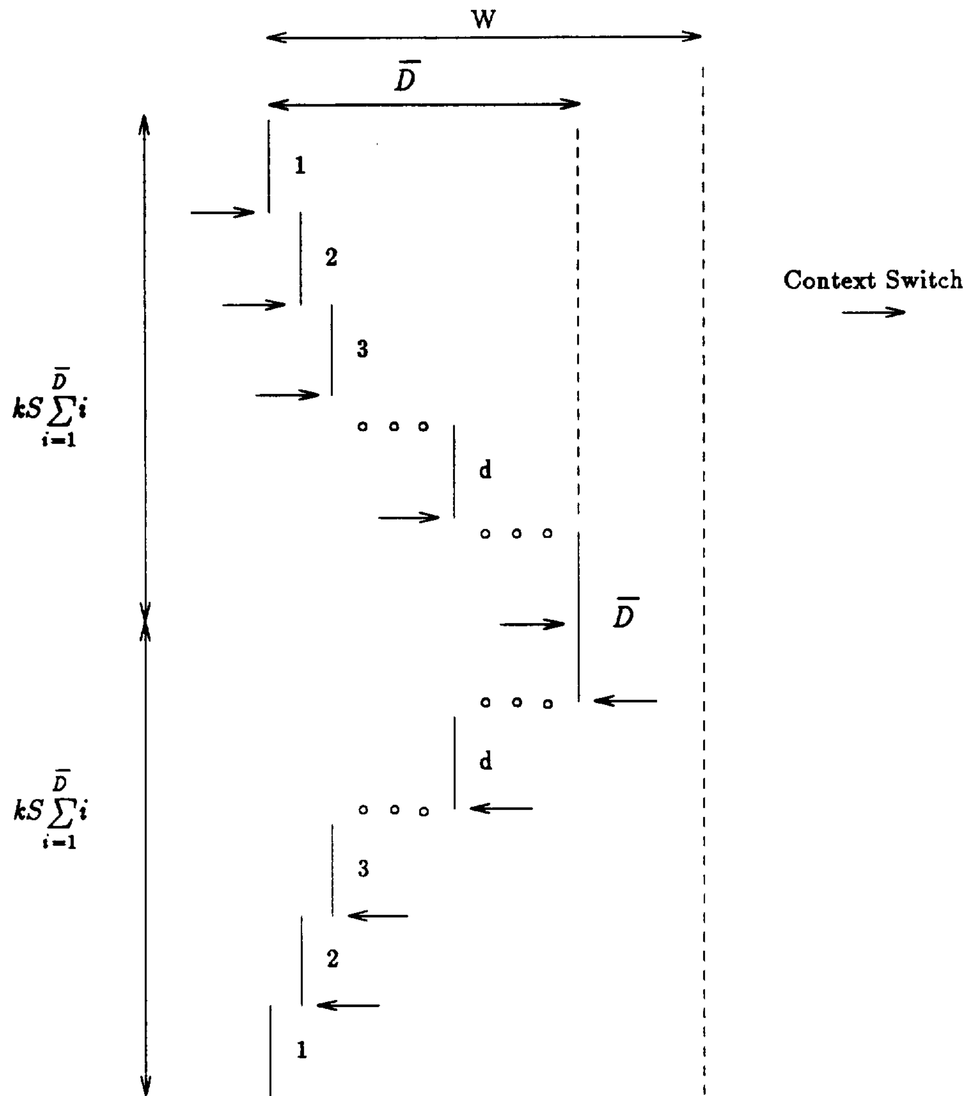


Figure 3(a) $W > \overline{D}$, $k = 1$
Context switching and procedure call overheads

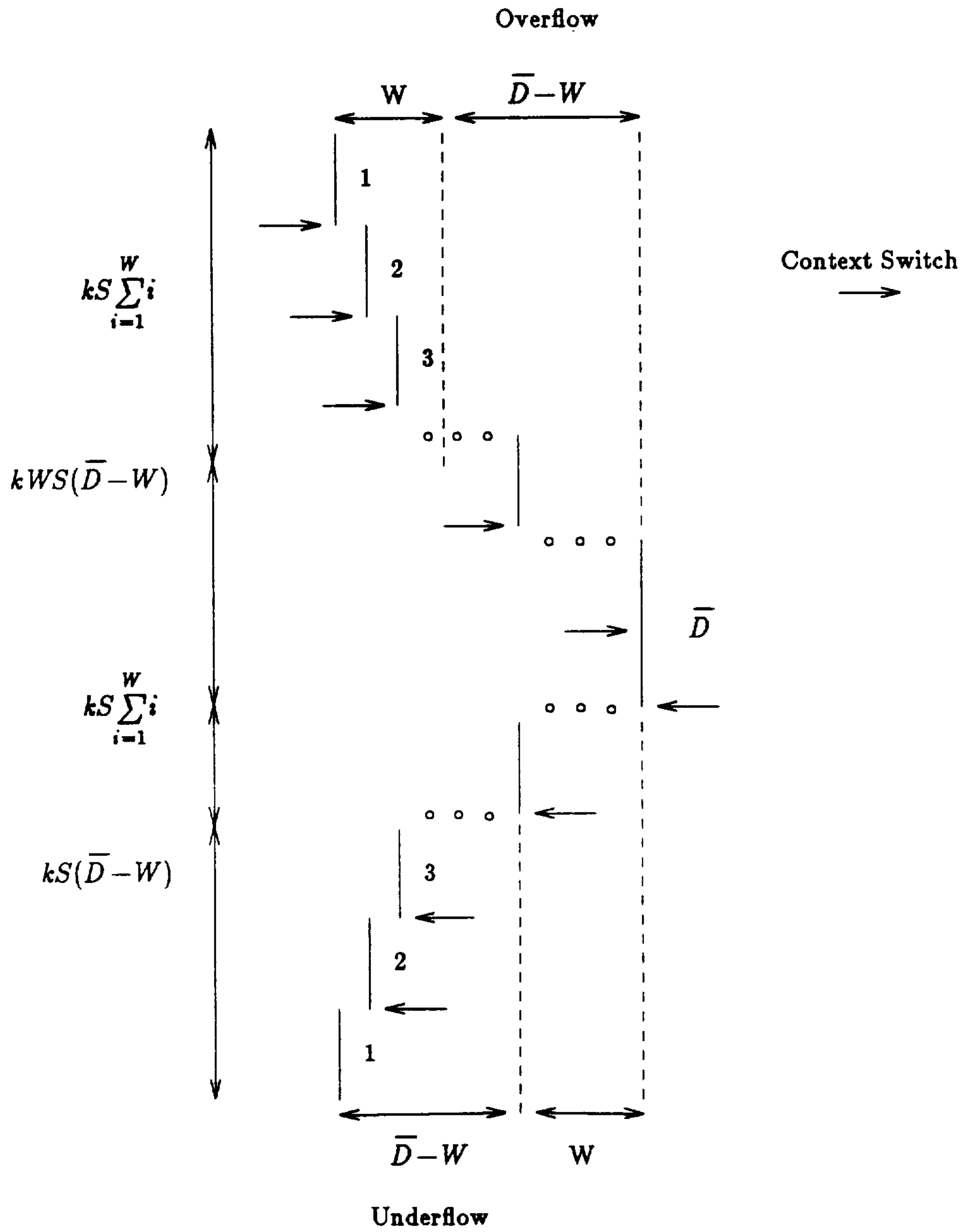


Figure 3(b) $W \leq \overline{D}$, $k = 1$
Context switching and procedure call overhead

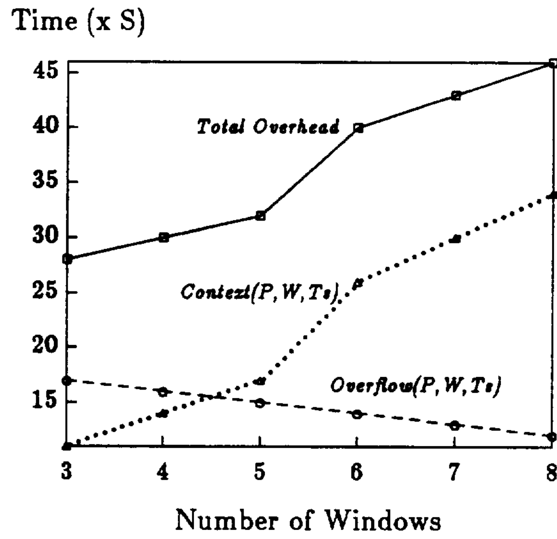


Figure 4 Procedure and Context Switch Overhead;
$\overline{D} = 20,\ T_s = \frac{Tk}{2},\ k = 6$

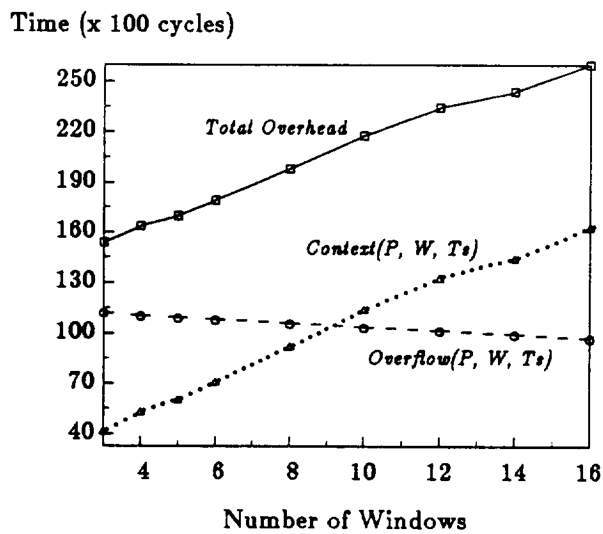


Figure 5 Procedure and Context Switching Overhead
Factorial 100, $Ts = 800$ cycles

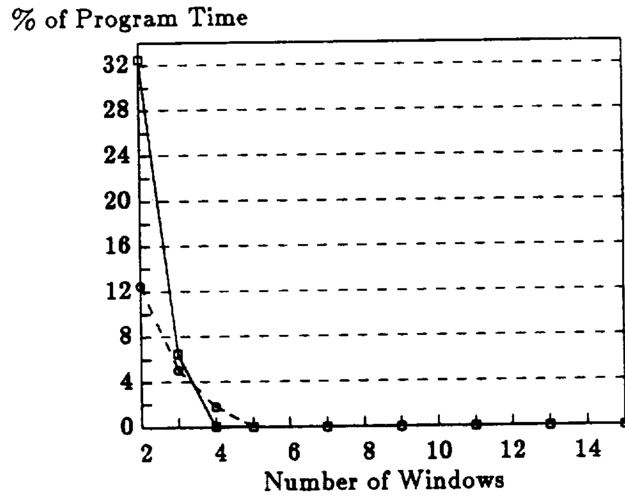


Figure 6 Procedure Overhead:
___ Dhrystone, _ _ _ Sed.

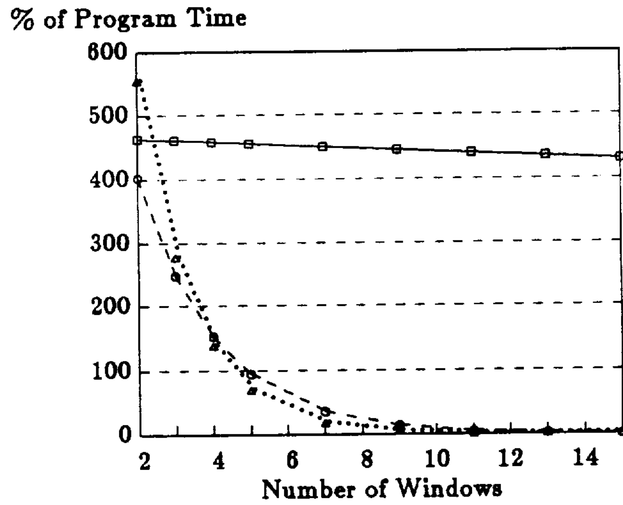


Figure 7 Procedure Overhead:
___ Ackerman, _ _ _ Fibonacci, ... Hanoi.

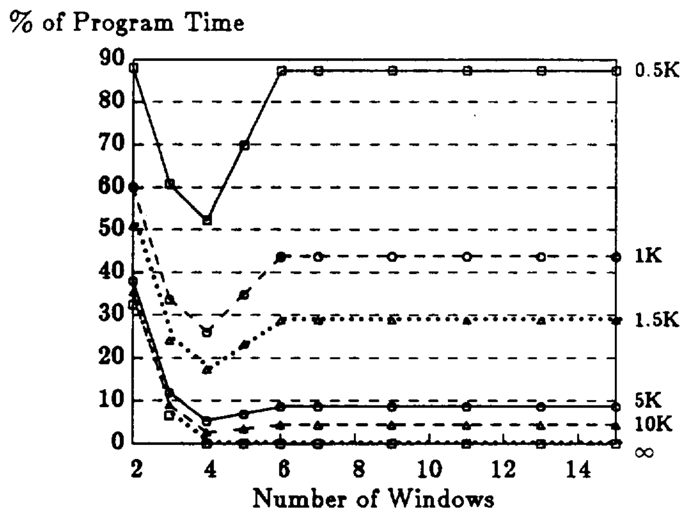


Figure 8 Procedure and Context Switch Overhead: Strategy (n,n), Dhrystone

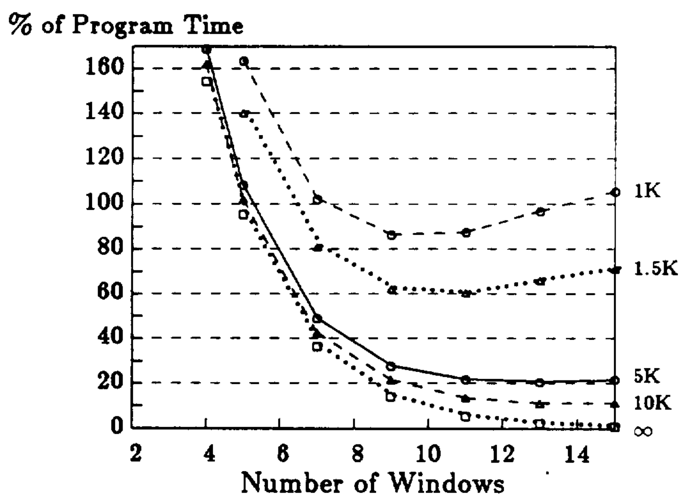


Figure 9 Procedure and Context Switch Overhead: Strategy (n,n), Fibonacci

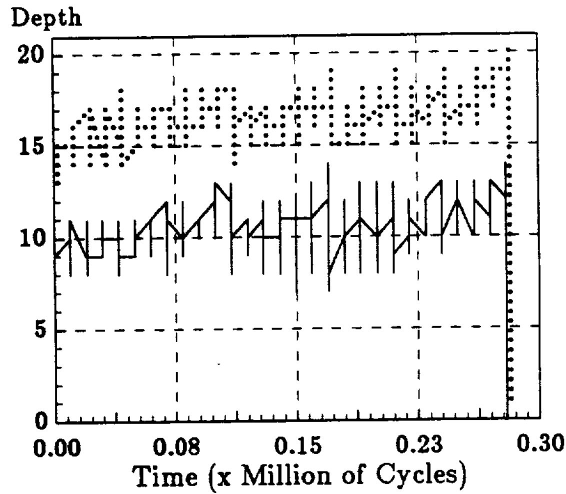


Figure 10 Procedure Nesting Depth of Fibonacci: ___ Minimum, ... Maximum

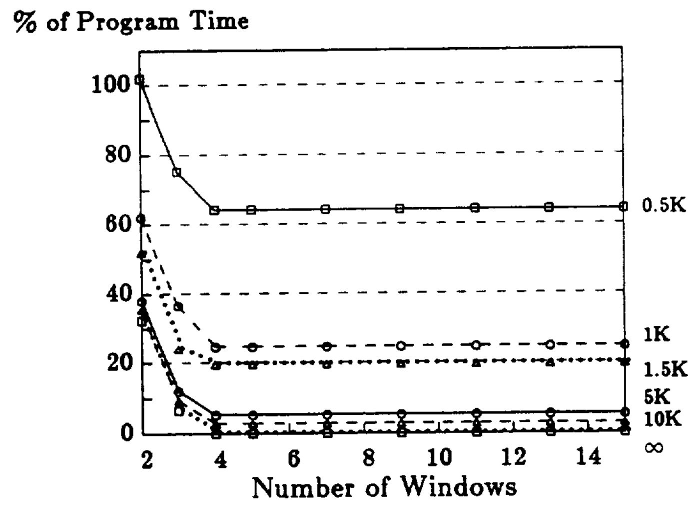


Figure 11 Procedure and Context Switch Overhead: Strategy (n,1), Dhrystone

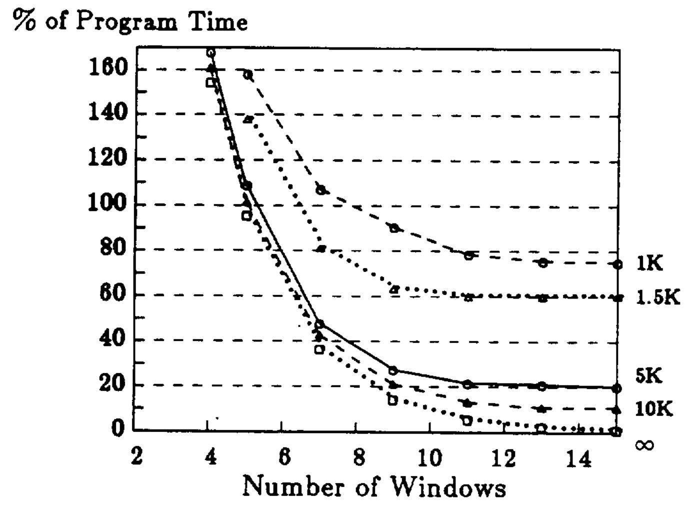
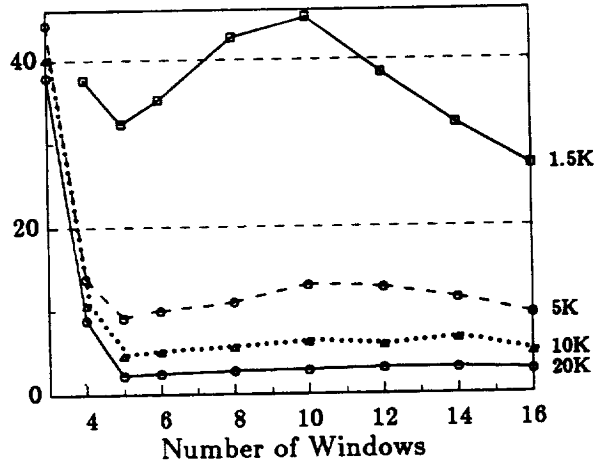
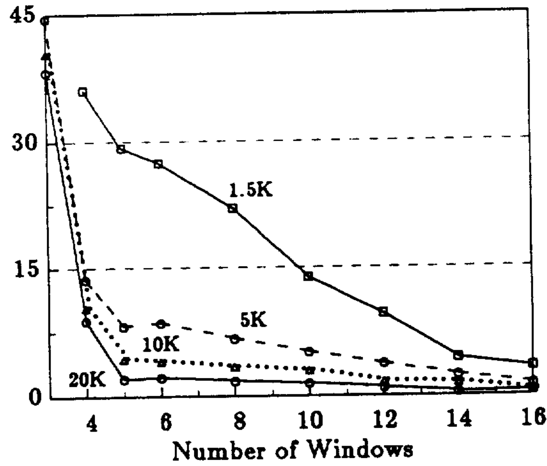


Figure 12 Procedure and Context Switch Overhead: Strategy (n,1), Fibonacci



(a) Mix 1 (Fibonacci, Dhrystone, and Puspnt).



(b) Mix 2 (Dhrystone, Dhrystone, and Qsort)

Figure 13 Procedure and Context Switch Overhead: *Strategy (0, 1)* for Dhrystone

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-87-1377	2.	3. Recipient's Accession No.
4. Title and Subtitle Context Switching with Multiple Register Windows: A RISC Performance Study			5. Report Date October 1987 6.
7. Author(s) Marian B. Konsek, Daniel A. Reed, Wittaya Watcharawittayakul			8. Performing Organization Rept. No. R-87-1377
9. Performing Organization Name and Address Department of Computer Science University of Illinois at Urbana-Champaign 1304 W. Springfield Urbana, IL 61801			10. Project/Task/Work Unit No. 11. Contract/Grant No. NSF DCR84-17948 1-5-29492
12. Sponsoring Organization Name and Address National Science Foundation 1800 G Street, NW Washington, D. C. 20550			13. Type of Report & Period Covered 14.

15. Supplementary Notes

16. Abstracts

Although previous studies have shown that a large file of overlapping register windows can greatly reduce procedure call/return overhead, the effects of register windows in a multiprogramming environment are poorly understood. This paper investigates the performance of multiprogrammed, reduced instruction set computers (RISCs) as a function of window management strategy. Using an analytic model that reflects context switch and procedure call overheads, we analyze the performance of simple, linearly self-recursive programs. For more complex programs, we present the results of a simulation study. These studies show that a simple strategy that saves all windows prior to a context switch, but restores only a single window following a context switch, performs near optimally.

17. Key Words and Document Analysis. 17a. Descriptors

RISC
Multiple register windows
Register file management
Reduced instruction set computer

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
unlimited	UNCLASSIFIED	36
	20. Security Class (This Page) UNCLASSIFIED	22. Price

FORM NTIS-35 (10-70) USCOMM-DC 40329-P71