Multiple Threads in Cyclic Register Windows

Yasuo Hidaka Hanpei Koike Hidehiko Tanaka

Department of Electrical Engineering, The University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113 Japan
E-mail : {hidaka,koike,tanaka}@mtl.t.u-tokyo.ac.jp

Abstract

Multi-threading is often used to compile logic and functional languages, and implement parallel C libraries. Fine-grain multi-threading requires rapid context switching, which can be slow on architectures with register windows. In past, researchers have either proposed new hardware support for dynamic allocation of windows to threads, or have sacrificed fast procedure calls by fixed allocation of windows to threads.

In this paper, a novel window management algorithm, which retains both fast procedure calls and fast context switching, is proposed. The algorithm has been implemented on the SPARC processor by modifying window trap handlers. A quantitative evaluation of the scheme using a multi-threaded application with various concurrency and granularity levels is given. The evaluation shows that the proposed scheme always does better than the other schemes. Some implications for multi-threaded architectures are also presented.

1 Introduction

Overlapping register windows have been incorporated into RISC architectures[9] to speed up procedure calls by reducing the number of register saves and restores. While register windows have proven quite effective in improving the sequential program performance, it has resulted in lengthening context switching time. Slow context switching is usually quite detrimental to the performance of multi-threaded applications, which are becoming increasingly important. For instance, distributed operating systems, object-oriented programming languages, functional and logic languages, all demand good multi-threading performance[2]. Furthermore, good multi-threading performance is essential for medium-grained and fine-grained parallel machines.

In this paper, a novel algorithm for managing register windows is proposed. It enables both *fast context switching* by sharing windows among threads, and *fast procedure calls* by allowing multiple windows to be used by a thread. Since the algorithm relies on cyclic window allocation, there is no need of special hardware support for dynamic window allocation. A key to the algorithm is the treatment of *window underflow trap*, which occurs on a procedure return when the caller's window has been spilt into memory. In the conventional algorithm, the caller's window is restored into the same place where the window resided before being spilt. If windows were to be allocated among several threads, the conventional algorithm would not work as is discussed in Section 3.

In our algorithm, the caller's window is restored in the same place that the *callee* had used. This always works because the caller's window is needed only when the callee terminates and thus, the callee's window is no longer needed. This simple idea avoids all the problems of window sharing in the conventional algorithm. Thus, it becomes possible to share simple cyclic register windows among several threads, and fast context switching is attained without totally sacrificing fast procedure calls.

Though the idea is quite simple, we have not seen any publications on this algorithm. In fact, SunOS on SPARC does not use this algorithm[6], and most researchers believe that overlapping register windows have lengthy context switching[2, 5, 10, 16]. In the following, we briefly discuss some published techniques for implementing multi-threading.

Cypress Semiconductor Co. have suggested a technique in their user's guide[12] for fast context switching for register windows. Agarwal et al. have used this technique in the implementation of APRIL[1]. In this method, one window is statically assigned to each thread. Though it enables fast context switching, the advantage of fast procedure calls is lost, because each thread can use only the assigned window.

Quammen, Kearns and Soffa have suggested another way to share register windows among multiple

0884-7495/93 $3.00

threads. They use register windows to hold temporary values in expression evaluation in a single stack for all the threads[11]. However, activation records of procedures are not held in registers, and thus, windows are not used to speed up procedure calls.

In the MASA architecture, proposed by Halstead and Fujita, *task frames* are provided for fast context switching, fast procedure calls and fast trap handling[4]. Task frames are non-overlapped multiple register sets, and a task can access at most three task frames – that of the current task, its child and its parent tasks. However, task frames are dynamically allocated for each task by sophisticated hardware.

Threaded windows proposed by Quammen and Miller are non-overlapping windows, which can be used in various manners, e.g. a register window-stack, a single register set, or a dynamically sized global area[10]. Though multiple threads can share threaded windows, sophisticated hardware is needed for dynamic window allocation.

Thus, all these approaches are quite different from ours. Namely, some researchers sacrifice the advantage of fast procedure calls for the sake of static window allocation for fast context switching, and the others require special hardware support for dynamic window allocation. Our algorithm maintains the advantage of fast procedure calls, without dynamic window allocation. In fact, we have implemented it on SPARC[3] which has simple cyclic window organization. It is of great benefit, because SPARC is widely used in workstations all over the world, and further, SPARC is already used in many parallel machines, e.g. CM5, EDS[14] and AP1000[13], in which multi-threading performance is especially important.

An implementation of the algorithm, and its evaluation are described in this paper. The evaluation was done quantitatively using real hardware, and a real multi-threaded application program with controllable concurrency and granularity levels. The evaluation was done also for various number of windows under fair scheduling and adaptive scheduling. Under adaptive scheduling, the proposed scheme works well even for a small number of windows.

The paper is organized as follows: Section 2 describes a basic management algorithm of register windows. In Section 3, problems in sharing windows among threads using the conventional window management algorithm, along with a solution, are presented. Section 4 discusses variations of the proposed algorithm, and Section 5 gives a discussion of the program behaviors which should be taken into account in the evaluation. The evaluation of the scheme is described in Section 6, and conclusions are presented in Section 7.

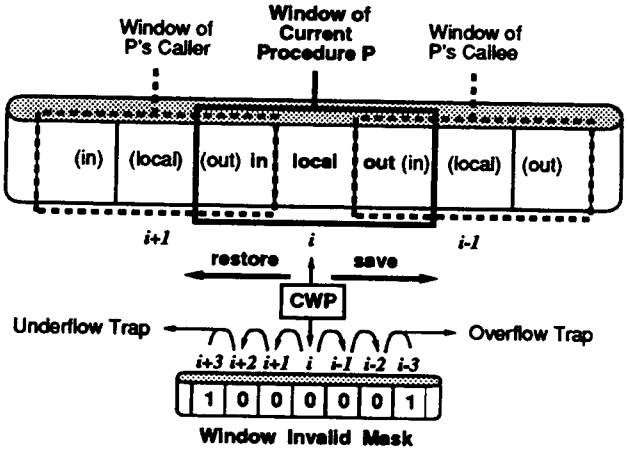


Figure 1: Organization of register windows of SPARC. The currently used window is indicated by CWP, which is changed by `save` and `restore` instructions. WIM is used to detect *window overflow* and *underflow*.

2 Basic register window management

Overlapping register windows were first introduced by Berkeley RISC machines[9], and then they were incorporated in the SPARC architecture[3]. Since our method was developed for SPARC, the terminology of SPARC is used in the paper. However, the proposed algorithm should be applicable to other processors with register windows.

Figure 1 shows the organization of the register windows of SPARC. The Current Window Pointer (CWP) indicates the window of the current procedure, which consists of three parts; 1) eight **in** registers shared with the caller (the parent procedure), 2) eight **out** registers shared with a callee (a child procedure), and 3) eight **local** registers which are private to the current procedure. `Save` and `restore` instructions are provided to change the CWP. On procedure entry, a `save` instruction is executed, which decreases CWP by one. Similarly the `restore` instruction on a procedure return increases CWP by one. In this paper, we regard window *i-1* as being *above* window *i*, and window *i+1* as being *below* window *i*.

Since the number of the windows on a processor is limited, only some contiguous windows near the stack top are held in the processor, and the rest of the stack is saved in memory (see **Figure 2**). The two ends of the contiguous windows in the processor are called *stack-top* window and *stack-bottom* window in this paper. The window buffer is managed in a cyclic manner

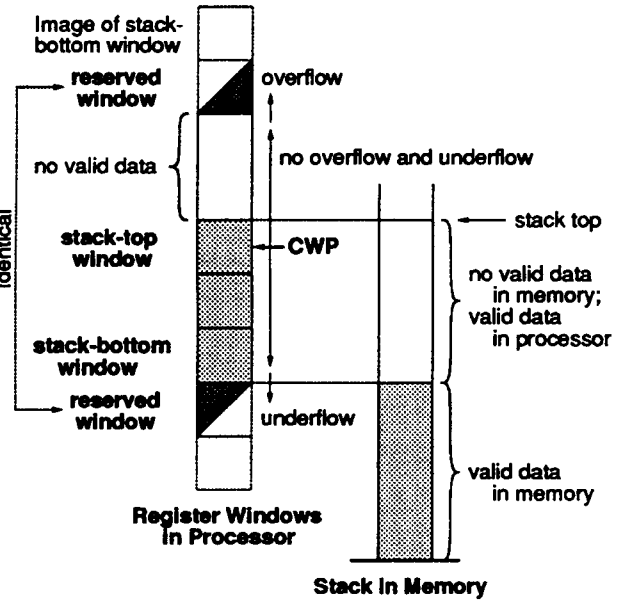
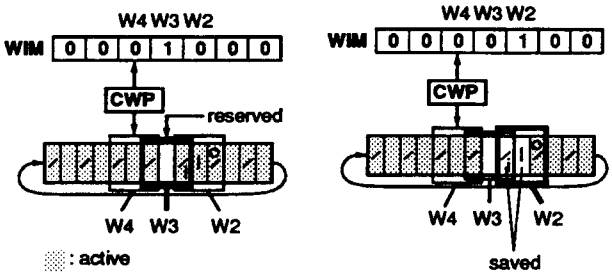


Figure 2: Mapping of stack onto register windows. At least one window has to be reserved to detect *window overflow* and *underflow*.

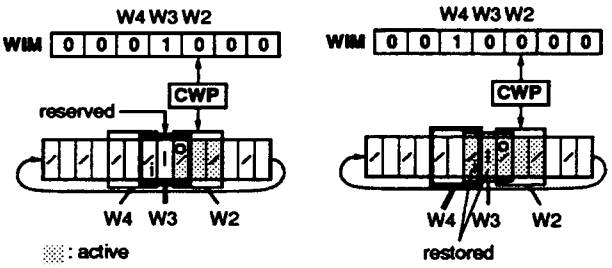
With the help of a *reserved* window, which indicates the limit of growth for the stack-top window. Window Invalid Mask (WIM) register shows whether each window is reserved or not. If the **save** instruction encounters a reserved window, an *overflow trap* occurs, and the trap handler saves some windows "above" the reserved window in the memory. Similarly, if a **restore** instruction encounters a reserved window, an *underflow trap* occurs, and the trap handler restores some windows from the memory. Tamir and Sequin studied the effect of the number of windows to be saved or restored for each *overflow* or *underflow trap*, and showed that transferring one window is the best in most cases[15].

Figure 3 shows an example of an *overflow trap*. Since CWP indicates window W4, and the window above is reserved, **save** causes an *overflow trap*. The trap handler saves **in** and **local** registers of window W2 in memory, and makes W2 the reserved window[6]. **Figure 4** shows an example of an *underflow trap*. Since CWP indicates window W2, and the window below is reserved, **restore** causes an *underflow trap*. The trap handler restores **in** and **local** registers of window W3 from memory, and makes W4 the reserved window. In this way, **in** and **local** registers are handled together, and **out** registers are handled as **in** registers in the window above. Since the alias of **in** and **out** makes window borders ambiguous, the term "window" means only **in** and **local** registers in the



(a) Before the trap. (b) After the trap.

Figure 3: An *overflow trap*. W4 is the current window. The trap occurs because W3 has been reserved. The **in** and **local** registers in W2 are saved, and W2 becomes the new reserved window.



(a) Before the trap. (b) After the trap.

Figure 4: An *underflow trap*. W2 is the current window. The trap occurs because W3 has been reserved. The **in** and **local** registers in W3 are restored, and W4 becomes the new reserved window.

following sections and figures, unless the term **out** is explicitly mentioned. While the figures look like non-overlapping windows, actual windows are overlapped.

3 Multiple threads in register windows

3.1 Problems

Figure 5 shows the basic idea of multiple threads in register windows. We may indicate the windows of the currently active thread by setting the corresponding WIM bits to 0, while setting all other WIM bits to 1. In order to minimize extra overhead, it is desirable

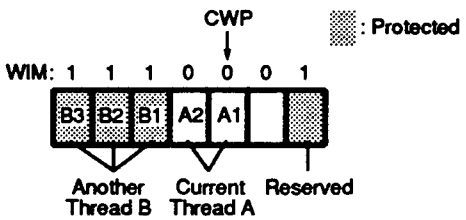


Figure 5: Multiple threads in register windows. Windows of the other threads are protected by setting the corresponding WIM bits to 1.

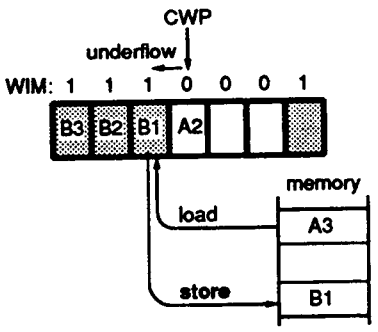
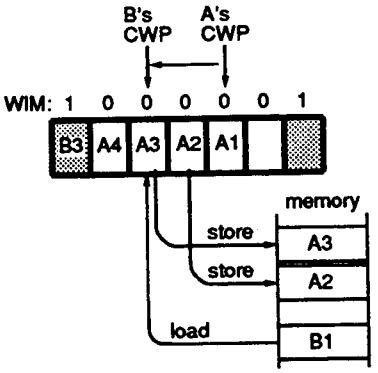


Figure 6: Window spillage on an *underflow trap*. Thread A demands window A3. Restoration of A3 requires saving window B1, which is the stack-top window of thread B.

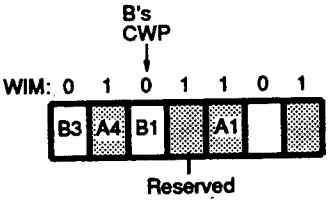
that all windows of a thread be contiguous, and represent the top fraction of the real stack of the thread. The basic window management algorithm needs to be modified to solve the following problems:

1. In the basic window management algorithm, window spillage can occur only on an *overflow trap*. However, if windows belonging to other threads were allowed, it could also occur on an *underflow trap*. That is, it might be required to save *another* thread's window before restoring the missing window of the current thread (see **Figure 6**).
2. In **Figure 6**, an *underflow trap* from thread A would cause the stack-top window of thread B to spill. Accordingly, windows of thread B would be no longer the top fraction of the real stack of thread B. However, since any thread demands its stack-top window first, and its stack-bottom window last, it is much more desirable to spill windows from the stack-bottom than the stack-top.
3. If windows were spilt from the stack-top window, windows of the thread may become non-contiguous, complicating context switching and trap handling. As an example of this, consider switching from thread A to B in **Figure 7 (a)**. The stack-top window B1 would have to be restored in the same place as A3, because of the relative position of window B1 to B3. As can be seen, the windows of thread A and B are no longer contiguous (see **Figure 7 (b)**).

These problems would involve awful complications in window management and lose most of the benefits of sharing windows among threads. A notable point is that all the above problems are caused by window spillage at *underflow traps*; there is no such problem for *overflow traps*. While a window spilt by an



(a) Context switching from thread A to B.



(b) After context switching.

Figure 7: Complicated context switching, and scattered windows. The stack-top window B1 would have to be restored in the same place as A3. The windows of thread A and B are no longer contiguous.

overflow trap may be another thread's window, there is no significant change in the window management, because the spillage is always from the stack-bottom window. Hence, if *underflow traps* could be handled without any window spillage, all the above problems may disappear.

3.2 Solution

Though *overflow* and *underflow* are handled symmetrically in the basic management algorithm, is this symmetry necessary? Suppose on an *underflow trap*, the missing window is restored in the same place as the current window (see **Figure 8**). This will involve the extra work of copying the active **in** registers of the current window (W3) into the **out** registers (into the **in** registers of W2), before the missing window can be restored. Though the current window does not physically move between before and after **restore**, it virtually goes back to the window below. In this way, window spillage is no longer needed at *underflow traps*, and all the problems mentioned earlier are successfully solved.

It should be noted that not all **in** registers of A2 need to be copied. The registers to be copied are usually only the values returned from the procedure, and the stack pointer. However, partial copying changes

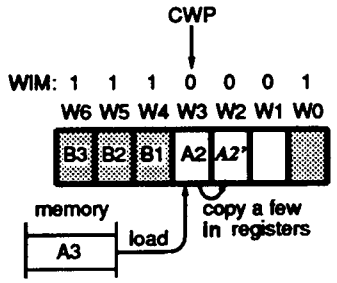


Figure 8: *Underflow trap* handling without any window spillage. Window A3 can be restored in W3, because most registers in A2 are no longer needed.

the semantics of the **restore** instruction. If other features of **restore** are used by the compiler, it may be necessary to copy all the **in** registers. Still the overhead is small.

4 Variations of the algorithm

The above description of the algorithm is not complete, because there are still several choices left to be made regarding reserved windows, window allocation and thread scheduling. We discuss these issues next.

4.1 With or without private reserved windows (PRW)

The proposed algorithm has two possibilities for the reserved window as shown in **Figure 9**; whether there is only one reserved window, or each thread keeps its own private reserved window (PRW).

Remember that actual windows are overlapped and **out** registers of the stack-top window may be held in the reserved window. In Figures 5 through 8, it was assumed that there is only one reserved window. Since the current thread must have a reserved window on the top, some extra work is required at the time of context switching. Suppose we want to switch to thread B in **Figure 9 (a)**. Window A2, the stack-bottom window of thread A, must be spilt into memory to make it into a reserved window for thread B. Also **out** registers of the stack-top window always have to be saved and restored on context switches.

On the other hand, if each thread keeps its own PRW as shown in **Figure 9 (b)**, **out** registers of the stack-top window need not be saved and restored on context switches. Furthermore, switching to another thread already resident in the windows never causes any window transfer. A minor technical point is that, on a context switch, if the suspended thread has some free windows above the stack-top, the PRW of the suspended thread is moved immediately above the stack-top. However, since the reserved window has no information to be copied, there is no overhead in doing so.

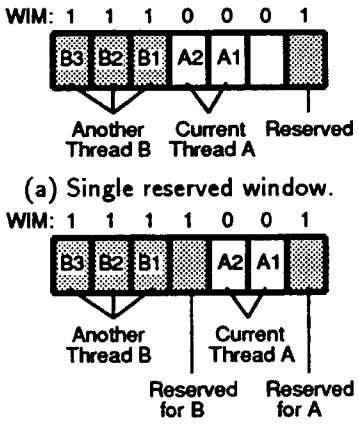


Figure 9: Alternative algorithm.

Since extra space is required to keep PRW for every thread, the rate of window spillage would potentially increase. Moreover, if a scheduled thread has no windows, two windows may have to be saved to allocate one for the new stack-top window, and another one for the new PRW. In the former scheme without PRW, only one window may have to be saved, because the old reserved window is available.

Both schemes are evaluated in later sections.

4.2 Window allocation

When a scheduled thread has no windows on a context switch, two windows have to be allocated. A simple allocation scheme is to allocate windows above the suspended-thread's windows. However, if it is used in conjunction with the scheme without PRW, one can imagine an undesirable case. Suppose a context switch occurs from thread A to thread B, and B does not have a window. B's window will be allocated above A's windows. Then, B immediately suspends without any procedure calls, and A is scheduled again. In order to make space for the reserved window of A, B's window will be spilt into memory. If there is repeated switching between threads A and B, this process may repeat itself causing unnecessary spillage and restoration.

It may be worth the extra cost to search for a free window, or to select the least-recently-used stack-bottom window when all windows are in use. Another concern is that there is external fragmentation of free windows. In our evaluations, we have only considered the simple allocation scheme.

4.3 Semantics of restore instruction

If only a few **in** registers are copied on an *underflow trap*, semantics of **restore** may be changed. However, even if all **in** registers are copied, there is still a potential problem, because the **restore** instruction on SPARC also acts as a kind of **add** instruction. Compilers often use this feature in a peephole optimization; if the instruction immediately before the **restore** instruction is an **add**, **sub**, or **move** instruction to set the return value in the particular register, that instruction and the **restore** instruction can be replaced by a single **restore** instruction which also performs the calculation.

There are two solutions for this problem. One is not to do the peephole optimization. In this way, one extra instruction may be occasionally added at the end of a procedure. The other solution is to interpret and emulate the trapped **restore** instruction by the *underflow trap* handler. This can be done with a small overhead, because the instruction format is simple and the destination register is either the particular return-value register if the adding function is used, or the zero register if it is not used. We used this emulation technique in our implementation.

4.4 Two types of context switching

If a suspended thread is known to sleep for a long time, it is sometimes more efficient to flush all the windows of the suspended thread. This is so because flushing a window at the time of context switch is cheaper than causing an *overflow trap* to save the window, that is, the overhead of entering and leaving the trap handler can be avoided.

Even if the proposed scheme is employed, it is a good idea to provide for both types of context switching; one that leaves windows of the suspended thread in situ, and the other, that completely flushes them. The reason for the suspension of a thread may be helpful in choosing the type of context switching. For instance, if a thread suspends on a cache miss and waits for the completion of a remote memory load, the thread is likely to wake up soon, and it may be better to leave its windows. On the other hand, if a thread suspends for the completion of a disk read, it may be better to flush its windows.

In the following evaluation, it is assumed that all threads are likely to wake up soon, and hence no flushing of windows is done in the sharing scheme.

4.5 Evaluated schemes

We have implemented and evaluated the following three schemes:

NS : Non-sharing scheme
Windows are not shared among threads like the conventional algorithm. All active windows are flushed on a context switch.

SNP : Sharing scheme without PRW
Windows are shared. There is no private reserved window for each thread, but only one reserved window. If the newly-scheduled thread has no windows, the window above the suspended thread's is allocated.

SP : Sharing scheme with PRW
Windows are shared. Each thread has its own private reserved window, which is located immediately above its active windows. If the newly-scheduled thread has no windows, the window above the reserved window of the suspended thread is allocated.

The scheduling is non-preemptive in all the evaluation. Besides, it is first-in-first-out (FIFO) except in the evaluation of the working set concept described below.

4.6 Working set concept on register windows

If there are insufficient number of windows, it is also possible for the sharing schemes to incorporate the working set concept of the virtual memory into register windows[8]. In the virtual memory, multiprogramming level is controlled so that the working set memory of all concurrently active programs fits in the available physical memory. This avoids thrashing.

Similarly in register windows, concurrency level can be controlled so that the working set of windows of all concurrently scheduled threads fits in the available physical windows. It can be done by changing the scheduling policy to give higher priority to the threads whose windows remain on the processor. Thus, the probability of window spillage is reduced, and the probability of fast context switching is increased.

The extra overhead in scheduling can be avoided by selecting threads *only* when a thread is awoken by somebody else. If the thread just awoken still has windows, it is enqueued in front of the ready queue; otherwise, it is enqueued at the back. This can be done with little overhead. Thus, the basic scheduler still remains FIFO, and no additional overhead is added to the time of context switching.

5 Program behavior

Evaluation of almost any aspect of processor architecture requires an understanding of the behaviors of the programs to be run[5]. In this section, we discuss what aspects of program behavior are likely to affect the performance of the proposed window management scheme.

The performance of the proposed scheme depends on how often a scheduled thread finds windows that were allocated to it prior to its suspension. If most windows are often spilt before the thread is scheduled again, the proposed schemes would be ineffective, because saving windows by *overflow traps* is more expensive than flushing them on context switches. This frequency crucially depends upon the total number of windows concurrently used by all threads. Here, we define several terms which concern us:

- **Total window activity:** The number of windows used during a given period, assuming there are infinite number of windows, (that is, assuming there are no window *overflows* and *underflows*). A repeatedly-used window is counted as one.
- **Window activity per thread:** The number of windows used between two successive context switches, assuming there are infinite number of windows. A repeatedly-used window is counted as one.
- **Concurrency:** The number of threads which are concurrently scheduled, at least once, during a given period. A repeatedly-scheduled thread is counted as one.
- **Granularity:** Execution run length between two successive context switches. The shorter the run length, the finer the granularity.
- **Parallel slackness:** The number of threads available for execution at a given time, excepting currently executed threads. (Length of the ready queue.)

Efficiency of the proposed scheme is directly affected by the *total window activity*; if it is smaller than the number of physical windows, the proposed scheme works well. Therefore, it is important to understand the *total window activity* of a given application, and how it is affected by other characteristics.

Total window activity is the product of *window activity per thread* and *concurrency*. The *window activity per thread* varies according to *granularity*. As the *granularity* becomes finer, the *window activity per thread* will decrease, and the *total window activity* will decrease, provided the *concurrency* is unchanged.

Concurrency is determined primarily by the characteristics of applications, but it also varies according to the scheduling policy. As an example, suppose threads are divided into two groups according to some criterion, and threads in different groups are scheduled in disjoint periods. Then, the *concurrency* will be less than that in the FIFO scheduling. The *total window activity* will also decrease, provided the *window activity per thread* is unchanged. In the working set concept, the scheduling policy reduces *concurrency* so as to make *total window activity* below the number of physical windows.

Parallel slackness represents how much leeway scheduling policy has in controlling *concurrency*. If the *parallel slackness* is always zero or one, the execution order of the threads is completely deterministic, regardless of scheduling policy; the working set concept will not work. On the contrary, if the *parallel slackness* is high, the scheduling policy can reduce *concurrency*, choosing a desirable thread at the time. Thus, higher *parallel slackness* is better for the working set concept.

The *granularity* greatly affects the frequency of context switching, and thus, has first order influence on performance. But it does not directly affect whether the proposed scheme works well or not. However, it does have an indirect effect on the success of the scheme through *window activity per thread* and *concurrency*.

Next, we will show how we generated various *granularity* and *concurrency* levels from a single application program. The application has realistic *window activity*, and this activity varies according to the *concurrency* and *granularity*. The application also has sufficient *parallel slackness*. The evaluation was done on a single processor, since the scheduling effects are difficult to understand on a parallel machine.

5.1 The application program: a multi-threaded spell checker

We used a spell checker for LaTeX source files as our application program. It is a multi-threaded version of the spell checker commonly used on UNIX systems. A draft version of this paper was used as the input in the evaluation. The draft used was 40500 bytes long.

Figure 10 shows the basic organization of the program. T1 through T7 are threads, and S1 through S6 are streams. T1 through T3 behave like UNIX filters, and constitute the main part of the program. T1 removes LaTeX commands from the input, and makes each line have just one word. T2 and T3 are spell-check threads. T3 filters out correct words, and feeds

incorrect words to T5, taking account of derivatives of words in the dictionary. T2 picks up and feeds incorrect derivatives to T5, and feeds other words to T3; otherwise, those derivatives would be filtered out as correct words by T3.

File input and output are *simulated* by T4 through T7. These threads, instead of actually reading (writing) disks, merely copy data from (to) their internal memory buffers into (from) the stream. These threads correspond to OS kernel threads, and their internal buffers correspond to disk cache.

T1 is written in lex (lexical analyzer generator) on UNIX, and the rest of the program is written in C. There are two primary differences in our program from the UNIX version; UNIX one has "deroff" for roff files instead of T1, and there is "sort -u" between T1 and T2 to reject duplicates of words. We omitted sort, because it accumulates the whole input, and it will be an obstacle to the concurrent execution of threads. However, the spell-checking algorithm is similar to the UNIX version. The *window activity* depends on the input, which has highly irregular patterns. Such *window activity* is much more complex and realistic than that of simple artificial benchmarks.

Each stream is FIFO, and is organized as a cyclic buffer. The buffers of S1 and S4 through S6 are each M bytes long, and those of S2 and S3 are each N bytes long. Since the scheduling is non-preemptive, a thread execution continues until an input (output) buffer becomes empty (full). Thus, there is usually more than one thread available for scheduling, and *parallel slackness* is sufficient.

Granularity and *concurrency* of the program can be changed as follows:

- **Granularity can be changed by the absolute value of M and N.**
- **Concurrency can be changed by the relative value of M and N.**

The *granularity* of a thread is determined by the size of the smallest input or output buffer related to the thread; the smaller the size, the finer the *granularity*. Besides, the overall *granularity* is determined by the smaller value of M and N.

If M and N are equal and rather small, many threads will be scheduled concurrently, and contribute to the *concurrency*. Thus, the *concurrency* will be *high*.

On the other hand, suppose M is much bigger than N, and N has a rather small value. T4 through T7 will have greater *granularity* and less frequent context switches than T1 through T3. Under such assumptions, the *concurrency* during execution of T4 through T7 will be almost one, and only T1 through T3 will contribute to the *concurrency*. Accordingly, the *concurrency* will be *low*.

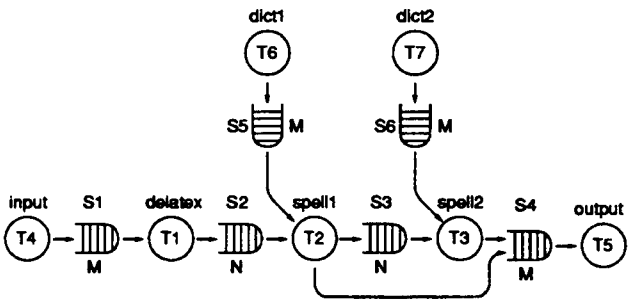


Figure 10: Organization of the application program. Threads are connected by streams. Each stream has an M or N bytes long buffer.

In this way, relative value of M and N provide a good way to control the *concurrency* in this application.

5.2 Evaluated program behavior

We have evaluated six kinds of behavior. Though it is difficult to show *concurrency* and *granularity* directly, one can realize them somewhat from **Table 1**. The reason of showing these numbers is that they are completely independent of the window management schemes and the number of physical windows, provided the scheduling is FIFO. Besides, the dynamic count of **save** instructions is independent of the buffer size and scheduling strategy, i.e. there are always the same number of function calls.

It is rather easy to understand *granularity*; the more the context switches, the finer the *granularity*. To understand *concurrency*, some consideration has to be given to the execution phases of the program; not all threads can be active in all phases due to various dependencies. Execution of the spell checker consists of two phases: reading the dictionaries and doing the spell check. During the first phase, the *concurrency* can be varied from a high value of four (all of T2, T3, T6 and T7) to the low value of one (one of T2, T3, T6 and T7). During the second phase, the high *concurrency* is between four and five (T1 through T4 and occasionally T5), and the low case is between three and four (T1 through T3 and occasionally T4 and T5).

It should be noticed that the change in *granularity* and *concurrency* are not completely independent. The *concurrency* is rather constant for different *granularity*, except in the second phase of the computation,

Table 1: Controlling *granularity* and *concurrency* by buffer sizes.

Program behavior							
Concurrency	high			low			
Granularity	fine	medium	coarse	fine	medium	coarse	
Buffer size (bytes)							
M	1	4	16	1024	1024	1024	
N	1	4	16	1	4	16	
Number of context switches (FIFO scheduling)							Dynamic count of save instructions
T1 (delatex)	60566	12680	2653	29838	8925	2001	113015
T2 (spell1)	102447	23497	5400	49952	9983	2049	110740
T3 (spell2)	80578	21327	5400	29887	8791	2049	75526
T4 (input)	40501	11548	2653	4817	4612	1974	10127
T5 (output)	1005	314	146	197	196	135	262
T6 (dict1)	50001	12501	3126	49	49	49	12502
T7 (dict2)	50001	12501	3126	49	49	49	12502
Total	385099	94368	22504	114789	32605	8306	334674

when the buffers are set for low *concurrency*. There finer *granularity* results in lower *concurrency*. However, the *granularity* is not constant for different *concurrency*. Therefore, care must be taken in comparing such cases.

When the working set concept is incorporated, the number of context switches is still within ±10% of the numbers in Table 1. Thus, the change in the *granularity* is also within ±10%. We have not measured the *concurrency* directly, because it is difficult to determine adequate length for a measurement period, which in turn depends on the *granularity*. However, the following result will show that the *concurrency* was definitely reduced in comparison with the FIFO scheduling.

6 Evaluation

6.1 Environment

Since implementing the window management schemes involves modification to trap handlers, it is difficult to evaluate them on a SPARC-based Unix system without deep understanding of the operating system. Fortunately, we are developing a parallel computer, PIE64[7], in which each processing element has a version of SPARC, S-20 made by Fujitsu. Therefore, this evaluation was done on a processing element of PIE64. As an aside, the algorithm described in this paper was discovered while developing a multi-tasking monitor for PIE64!

In order to evaluate performance at various numbers of physical windows, we have also developed a register window emulator. In this emulator, usual instructions are executed at real speed, but instructions which concern windows are trapped and emulated. A cycle counter for measurement is stopped during the emulation, so that the exact result is obtained.

6.2 Number of cycles for a context switch

Before showing performance evaluation, we begin with the number of cycles for a context switch in each scheme. While this measurement is rather static one, it is not merely instruction counts, but takes account of all cycles on S-20; i.e. cycles for instruction fetch, data transfer, pipeline stall, and pipeline flush. It was done by monitoring SPARC's bus traffic with a dedicated logic analyzer of PIE64.

Table 2 shows the result. The number of windows transferred on a context switch is dependent on the scheme and the situation of windows at the time. In addition to the window transfer, there is other overhead such as calculation of the WIM and scheduling, because everything is implemented in software.

In the NS scheme, as the number of active windows increases, the cost grows substantially high. Besides, even in the best case, two windows have to be transferred. Furthermore, the cost is terrible in the worst case, in which all windows except a reserved one have to be saved; this is an undesirable characteristic in hard real time systems.

The NS scheme has also hidden overhead of *underflow traps*; if two or more windows are saved at a context switch, some of the saved windows will have to be restored by *underflow traps*. (More precisely the stack-top window is restored on the context switch.)

On the contrary, the sharing schemes take less cycles in many cases than the NS scheme. In the best case, no windows have to be transferred. Though there

Table 2: Number of cycles for a context switch.

Scheme	Window Transfer		Cycles
	save	restore	
NS	1	1	145 - 149
	2	1	181 - 185
	3	1	217 - 221
	4	1	253 - 257
	5	1	289 - 293
	6	1	325 - 329
	...	...	...
SNP	0	0	113 - 118
	0	1	142 - 147
	1	0	162 - 171
	1	1	187 - 196
SP	0	0	93 - 98
	0	1	136 - 141
	1	1	180 - 197
	2	1	220 - 237

is still software overhead in the best case, it will be reduced to zero or a few cycles, if the proposed algorithm is implemented in multi-threaded architecture.

The SP scheme takes less cycles in the best case than the SNP scheme, because **out** registers in the stack-top window and program counters can be held in PRW. However, the SP scheme is more expensive in the worst case than the SNP scheme, because two windows have to be saved in the worst case.

6.3 High-concurrency case

Figure 11 shows performance comparison among the schemes in the *high-concurrency* case. Execution time of each scheme was evaluated for three levels of *granularity* at various numbers of windows. The range of the number of windows was from four to thirty two.

If there are sufficient number of windows, the best scheme is SP, and if the number of windows is small, the NS scheme is best; there is no region where the SNP scheme outperforms both the SP and NS schemes. Besides, as the *granularity* becomes fine, the advantage of the sharing schemes increases.

Variation in *total window activity* is also observed in this figure. Look at the number of windows where the effect of more windows is saturated in the sharing schemes. That number is proportional to the *total window activity*. Thus, as the *granularity* becomes fine, the *total window activity* decreases.

Figure 12 shows average time of a context switch in the *high-concurrency* case. Compare this figure with Table 2. If there are sufficient number of windows, context switch time of the SP and SNP schemes, is very close to the best case, especially in the fine

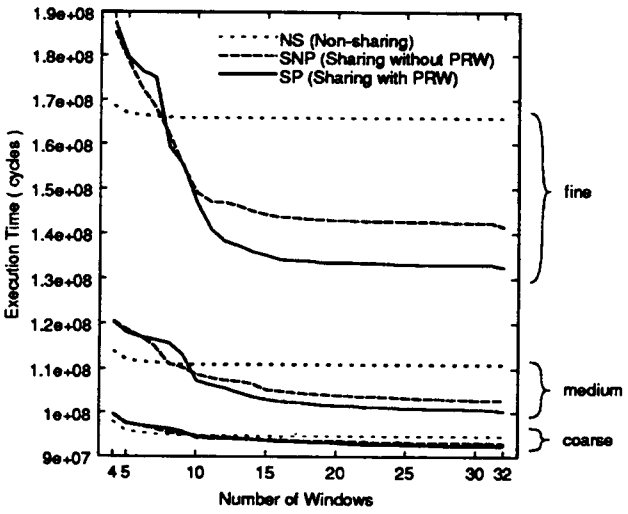


Figure 11: Performance at *high concurrency*. With sufficient number of windows, SP is best. *Total window activity* varies according to *granularity*.

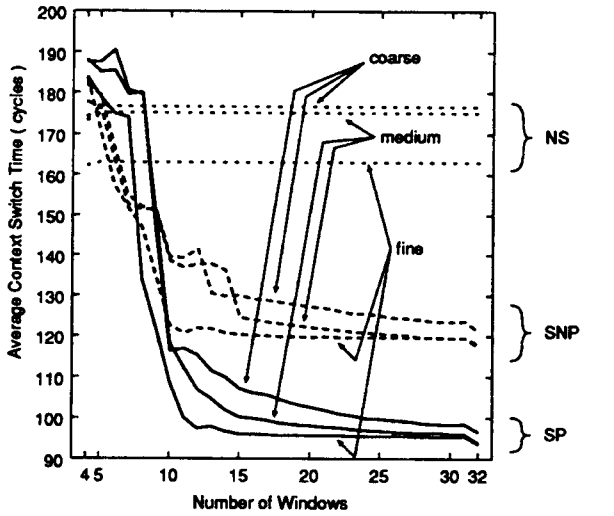


Figure 12: Average context switch time at *high concurrency*. With sufficient number of windows, the result of SP and SNP is very close to the best case (see Table 2).

granularity. It means most context switches are done without any window transfer.

This fact is very important to multi-threaded architecture. Since such frequency of the best case is independent of the implementation, this fact suggests that multi-threaded architecture with the proposed algorithm will have excellent multi-threading performance.

Figure 13 is the probability of *window overflow* and *underflow traps*. Since the number of function calls is constant, it shows that the SP and SNP scheme is also very effective for fast procedure calls with sufficient number of windows.

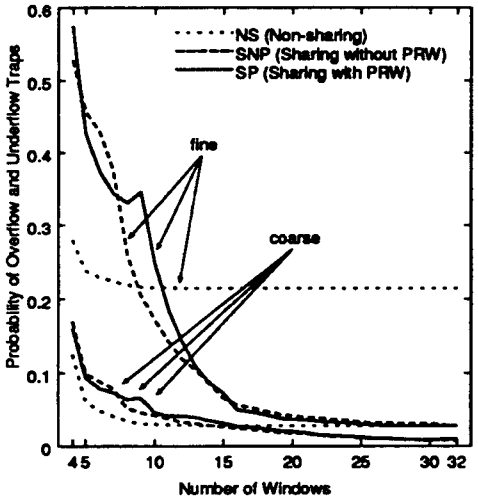


Figure 13: Probability of *window traps* at *high concurrency*. The number of *window overflow* and *underflow traps* divided with the number of executed `save` and `restore` instructions.

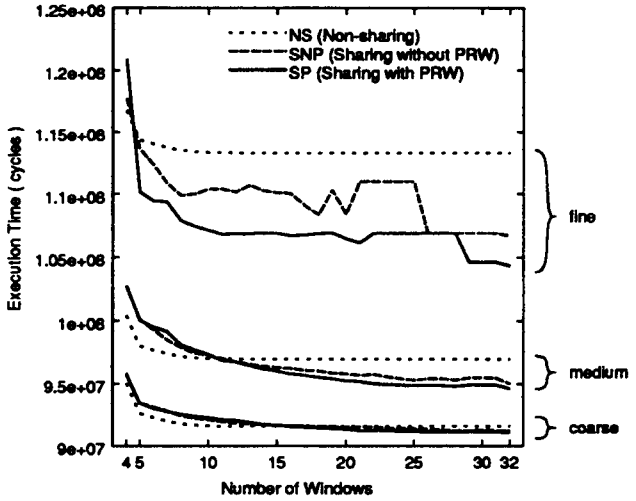


Figure 14: Performance at *low concurrency*. The variation in *total window activity* is greater than the *high-concurrency* case. 20 or more windows are required for SP at *coarse granularity*.

6.4 Low-concurrency case

Figure 14 is the performance comparison in the *low-concurrency* case. The SNP scheme has strange behavior at fine *granularity*. It is probably caused by the undesirable effect of the simple window allocation mentioned before, because the order of threads in windows is affected by the number of windows. The variation in *total window activity* is greater than the *high-concurrency* case, and 20 or more windows are required for the SP scheme at the coarse *granularity*.

Thus, *total window activity* greatly varies according to *granularity* and *concurrency*. Though the variation in *concurrency* is rather small in this evaluation, it is natural to imagine that other programs have quite-different *concurrency*, and hence, quite-different *total window activity*. Therefore, unless the working set concept is incorporated, the proposed scheme works well only for applications whose *total window activity* is low enough for the number of available physical windows.

6.5 Working set concept on register windows

Figure 15 is the performance comparison when the working set concept is incorporated into the *high-concurrency* case. The performance at a small number of windows is much improved, and the sharing schemes work well with even seven or eight windows. Besides, there is no significant performance loss from the FIFO scheduling at a large number of windows.

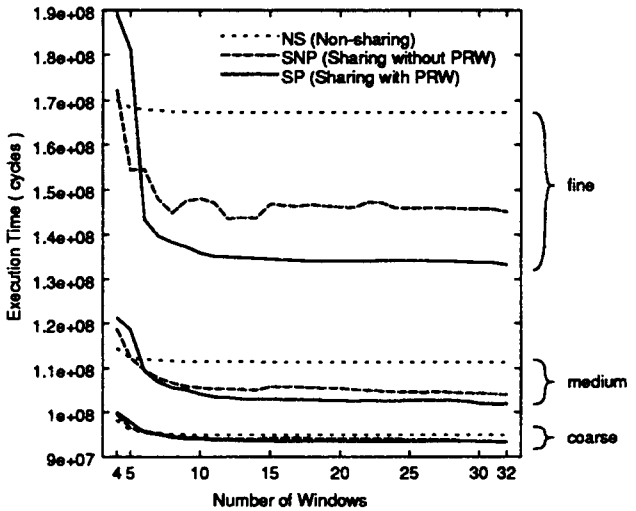


Figure 15: Performance at *high concurrency* with the working set concept. *Total window activity* is reduced to the number of available windows unless the available windows are extremely limited.

The fact that the sharing schemes do not work well at four or five windows means that the scheduling cannot reduce the *total window activity* to such low level. If an application program has little *parallel slackness* but high *concurrency*, this lower limit of *total window activity* will be high. In order to deal with such applications, *it is worth having as many windows as possible*.

However, a reason for lack of *parallel slackness* is bad program design; too much invocation of threads which cannot be executed in parallel actually. For

such an application, it may be possible to rewrite the program to have lower *concurrency* and lower *total window activity*. Thus, the sharing scheme with private reserved windows together with the working set concept always works well.

7 Conclusion

A new management algorithm of cyclic register windows was described. It enabled multiple threads to share windows, and improved multi-threading performance by converting the drawback of lots of registers to a benefit. If the working set concept is incorporated in register windows, ***the proposed scheme always works well***.

The following are the most important implications of our study:

1. The proposed algorithm will improve the performance for a certain class of applications on stock processors which have a small number of windows.
2. Since, according to our algorithm, the advantage of fast procedure calls is not at the expense of lengthy context switching, it is possible to use more register windows profitably. The trade-off in new processor design will be between the advantage of fast context switching and the lengthening of register-access time.
3. The proposed algorithm is also applicable to multi-threaded architecture, where excellent multi-threading performance as well as excellent sequential performance can be obtained without any sophisticated hardware support for dynamic window allocation.

Acknowledgements

We are deeply indebted to Professor Arvind of MIT for his help in writing this paper. We had useful discussions with Mr. Hosaka of Soum Corporation in Japan. We are also thankful to Mrs. Gita Mithal for editing an earlier draft of this paper. This work is supported by Grant-in-Aid for Scientific Research (No.03555071, No.03003891) from the Ministry of Education, Science and Culture. One of the authors, Hidaka, is supported by JSPS Fellowships for Japanese Junior Scientists.

References

[1] A. Agarwal, B. H. Lim, D. Kranz, and J. Kubiatowicz, "APRIL: A Processor Architecture for Multiprocessing," *Proc. of 17th Annual Intl. Symp. on Comp. Arch.*, pp. 104-114, 1990.

[2] T. E. Anderson, H. M. Levy, B. N. Bershad, and E. D. Lazowska, "The Interaction of Architecture and Operating System Design," *Proc. of ASPLOS-IV*, pp. 108-120, 1991.

[3] R. B. Garner, et al., "The Scalable Processor Architecture (SPARC)," *Proc. of COMPCON88*, pp. 278-283, 1988.

[4] R. H. Halstead and T. Fujita, "MASA: A Multithreaded Processor Architecture for Parallel Symbolic Computing," *Proc. of 15th Annual Intl. Symp. on Comp. Arch.*, pp. 443-451, 1988.

[5] J. L. Hennessy and D. A. Patterson, *Computer Architecture : A Quantitative Approach*, Morgan Kaufmann Publishers, 1990.

[6] S. R. Kleiman and D. Williams, "SunOS on SPARC," *Proc. of COMPCON88*, pp. 289-293, 1988.

[7] H. Koike and H. Tanaka, "Overview of the Parallel Inference Engine: PIE64," *Annual Report of Engineering Research Institute*, Faculty of Eng., Univ. of Tokyo, Vol.48, pp. 63-68, 1990.

[8] M. Maekawa, A. Oldehoeft, and R. Oldehoeft, *Operating Systems - Advanced Concepts*, The Benjamin/Cummings Publishing Co., Inc., 1987.

[9] D. A. Patterson and C. H. Sequin, "RISC I: A Reduced Instruction Set VLSI Computer," *Proc. of 8th Annual Intl. Symp. on Comp. Arch.*, pp. 443-457, 1981.

[10] D. J. Quammen and D. R. Miller, "Flexible Register Management for Sequential Programs," *Proc. of 18th Annual Intl. Symp. on Comp. Arch.*, pp. 320-329, 1991.

[11] D. Quammen, J. P. Kearns, and M. L. Soffa, "Efficient Storage Management for Temporary Values in Concurrent Programming Languages," *Trans. on Comp.*, Vol.C-34, No.9, pp. 832-840, 1985.

[12] ROSS Technology, Inc., *SPARC RISC USER'S GUIDE, 2nd Edition*, Cypress Semiconductor Corporation, 1990.

[13] T. Shimizu, et al., "Low-Latency Message Communication Support for the AP1000," *Proc. of 19th Annual Intl. Symp. on Comp. Arch.*, pp.288-297, 1992.

[14] C. J. Skelton, et al., "EDS: A Parallel Computer System for Advanced Information Processing," *Proc. of the 4th International PARLE Conference*, LNCS 605, Springer-Verlag, pp. 3-18, 1992.

[15] Y. Tamir and C. H. Sequin, "Strategies for Managing the Register File in RISC," *Trans. on Comp.*, Vol.C-32, No.11, pp. 977-988, 1983.

[16] M. Weiser, A. Demers, and C. Hauser, "The Portable Common Runtime Approach to Interoperability," *Proc. of the Twelfth ACM Symposium on Operating Systems Principles*, pp. 114-122, 1989.