Fast Context Switches: Compiler and Architectural Support for Preemptive Scheduling

Jeffrey S. Snyder, David B. Whalley, and Theodore P. Baker

Department of Computer Science
Florida State University,
Tallahassee, FL 32306, U.S.A.

Abstract

This paper addresses the possibility of reducing the overhead due to preemptive context switching in real-time systems that use preemptive scheduling. The method introduced in this paper attempts to avoid saving and restoring registers by performing context switches at points in the program where only a small subset of the registers are live. When context switches occur frequently, which is the case in some real-time systems, performing context switches at fast context switch points is found to significantly reduce the total number of memory references. A new technique, known as *register remapping*, is introduced which increases the number of these fast context switch points without degrading the efficiency of the code.

1 INTRODUCTION

This paper addresses the possibility of reducing the overhead due to context switching in real-time applications. Context switching cost is a limiting factor in the frequency of cyclic tasks that can be supported with preemptive scheduling techniques. In the evolution of microprocessors, the number of clock cycles required for a full context switch has tended to increase. Thus, reducing context switch overhead appears important if full advantage is to be taken of high performance microprocessors in real-time applications.

Processes running in a real-time environment place more constraints on time. Process deadlines must be met, especially in the case of hard real-time systems. Systems failing to meet these time constraints can result in a fatal failure [Lev90]. The possibility of reducing the overhead of a context switch is especially appealing with those systems which are considered hard real-time. Such systems characteristically include processes with periodic execution requirements, which can require very frequent preemptive context switches. Reducing the length of time it takes to perform a context switch makes more processor time available for useful work. This can potentially enable a system to handle a higher work load without missing hard deadlines.

Different methods can be used to reduce the amount of overhead in a context switch. One can inspect the algorithms that perform the context switch. Modifications to these algorithms result in a more efficient context switch. From a more abstract point of view one can examine the state of a process at any given time. The question then arises of whether or not it is actually necessary to save the entire state when control is taken away from a task and given to another. Likewise, if it is not necessary to save an entire state when preempting a process, then it would not be necessary to restore the entire state when the process resumes control of the CPU.

2 REDUCTION OF OVERHEAD

When tracing the execution of a process or task, one can examine the state of the registers at each instruction. If the register is in use, then it is considered to be live, whereas if it is not in use then it is considered to be dead. A group of instructions that can be potentially executed contiguously with the same live register is called a live range. If no user-allocable registers were live at the point of a desired context switch, then the cost of the context switch could be reduced to saving and restoring the state of the program counter, stack pointer, etc. This situation introduces the concept of a fast context switch as opposed to a slow context switch where all the registers are saved and restored. By performing dataflow analysis of a function in a program one can determine the instructions in which a group of the registers are dead.[1] This introduces the concept of a fast context switch point.

With an emphasis being placed on the number of potential fast context switch points in a program, one wonders what kind of optimizations can be performed that would provide a greater number of instructions with zero live registers. One investigated optimization

[1]The word *group* refers to a subset of the user-allocable registers that is determined based on the calling convention. The calling convention used for the experiments in this paper is described later in Section 4.

that provided more fast context switch points is called *register remapping.* It reduces the total number of live registers for each instruction in a given live range without the addition of extra instructions or memory references within the program. The register remapping optimization is performed on a live range that has been allocated to a scratch register.[2] The scratch register can be remapped to a non-scratch register if no other interfering live range is allocated to the non-scratch register. The concept of remapping registers will be described in more detail in Section 4.

Another issue that needs to be investigated is the effect on cache performance from changing the points in a program where context switches occur. A processor's speed depends greatly upon its cache performance [Mog91] [Aga88], and the cost of reduced cache performance could outweigh the benefits from performing fast context switches.

More analysis of cache performance can be done with respect to the context switch interval. A context switch interval is the length of time between desired context switches. In a periodic hard real-time system, the context switch interval cannot be longer than the period of the highest-frequency process; in some systems, this could be as short as 1 millisecond. In other systems, the context switch interval may be determined by the time-slicing policy; in this case a shorter context switch interval translates into better real-time response.

The potential benefit of fast context switching is likely to depend on the context switch interval. Clearly, the longer the interval between context switches, the less significant is the cost of context switching and the benefit from improvements in context switch speed. More frequent context switching obviously has a direct cost in the execution time of the context switches. It can also reduce cache performance by breaking up the locality of references.

3 RELATED WORK

3.1 Analysis of Cache Performance

Recent research done by Mogul and Borg described the effects of context switches on cache performance [Mog91]. This paper provided good indications on how context switches generally affect cache performance. Mogul and Borg conducted their experiment by running a

[2]A *scratch* register is a register whose value cannot be guaranteed to be retained across a function call. A *non-scratch* register is one whose value is retained across a function call.

set of programs concurrently on a UNIX system, and these executions generated instruction and data address traces which were fed directly into a cache simulator. Mogul and Borg limited their cache analysis to just those short intervals of time after a context switch since this is when a context switch has its greatest impact on the cache.

The results of their study show that as processor speeds grow faster than memory speeds, the cost of cache misses and context switches is going to get worse. The main reason for this is that achieving a high rate of cache hits is the only way a fast processor will be able to maintain its peak performance. The act of performing a context switch is likely to cause a burst of cache misses because it changes the locality of reference for the system. One way of offsetting this decrease in performance was to use a larger cache.

Agarwal et al. [Aga88] also investigated the overall effects of context switches on cache performance. Their results showed that large caches suffered considerable performance loss when forced to purge their contents after each context switch, whereas small caches did not suffer as much. On the other hand, larger caches demonstrated an increase in performance when they were not required to purge their contents. This was due to the fact that a process's cache lines remained valid over many context switches.

3.2 Threaded Register Windows

Two papers [Qua91][Mil90] discuss the concept of "threaded register windows." Threaded register windows is a register hardware organization that targets RISC architectures and the exploitation of their large register sets. According to Quammen and Miller the two types of register hardware organizations, single register set and register windows, each have their own advantages and disadvantages depending on the type of program executing. A single register set will have an advantage if the program accesses global data frequently, whereas register windows will have an advantage if the program contains many procedure calls. Quammen and Miller feel that the advantages gained from using either a single register set or register windows can be combined by using their hardware approach which allows for the dynamic configuration of the register hardware organization. Both of the papers indicate that architectures with a large number of registers experience a great deal of overhead in saving and restoring the registers.

The concept of threaded register windows attempts to reduce this context switch overhead. With this hardware organization concurrent processes share a register bank in which register windows are dynamically allocated and deallocated to meet the needs of the processes. When a context switch occurs, only the state of the processor status word (PSW), which contains only pointers into the register bank, needs to be saved.

3.3 Lightweight Processes

In a multitasking environment the size of a process's state is reduced by using lightweight processes. Lightweight processes, also known as either lightweight threads or lightweight tasks, execute in the environment of a heavyweight process. This allows for the lightweight processes to share address space and the resources allocated to the heavyweight process. A context switch between two lightweight processes in the same address space reduces the amount of state that is necessary to be saved and restored [Nut92][And91][Mil90]. Saving and restoring registers is still required and becomes the dominating cost of the context switch.

4 IMPLEMENTATION

The experiments in this paper on fast context switching and cache performance analysis were conducted using a version of VPO (Very Portable Optimizer) [Dav91a] which was retargeted to the Motorola 68020/68881 architecture. VPO is a back-end optimizer and performs its optimizations on register transfers or register transfer lists (RTLs) which represent legal instructions for a target machine. Each register transfer represents a change in the machine's state, and instructions which cause more than one state change are represented by lists of register transfers. An environment called EASE (Environment for Architecture Study and Experimentation), which is designed to measure code produced by VPO, was used to gather frequency and cache performance measurements [Dav91a].

To determine if reducing the overhead of context switches was feasible, a histogram was generated from the execution of a set of programs to indicate the number of instructions that are executed between fast context switch points. One can see that the delay is usually short from the histogram in Table 1. Note that even though the average number of instructions was small, there still existed a few programs that had execution sequences which did not encounter fast context switch points for long periods of time.

Table 1: Histogram

n Instructions Executed Before Reaching FCSP				
	$0 \leq n \leq 100$	$101 \leq n \leq 1000$	$1001 \leq n \leq 9998$	$n \geq 9999$
Occurrences	99.862%	0.046%	0.090%	0.001%

4.1 Hardware Supported Model

The hardware proposed for the model investigated in this paper has a single bit field in each instruction indicating whether or not the instruction is a fast context switch point. When an interrupt signal is received by the CPU, a flag is set that tells the CPU to look for a fast context switch point. When the flag is set, the CPU continues to execute instructions, counting clock cycles. If an instruction is found with the bit set that represents a fast context switch point, then a fast context switch occurs. If more than a predetermined number of clock cycles elapse without a fast context switch point being found, then the CPU will perform a slow context switch. One of the advantages to using this model is that there is no added overhead. Another advantage is that the compiler can be used to locate the points in a program where no registers with a group are live. It should also be noted that one disadvantage is that one bit has to be sacrificed in each instruction.

The hardware model was incorporated into EASE by modifying the cache simulator. An extra parameter was passed to the cache simulator which indicated whether or not the given instruction was a fast context switch point. A context switch window (CSW), which was defined as 100 instructions, centers around the boundary between context switch intervals (CSI). The cache simulator began looking for a fast context switch point at the beginning of the window. If one could not be found before reaching the end of the window, then a slow context switch was performed.

4.2 Calling Conventions

A calling convention is essentially a mechanism which preserves the values of registers across function calls. VPO uses a hybrid calling convention [Dav91b] which partitions the registers into two groups. A caller-save register, which is a scratch register, will be saved and restored by the caller if it is live across a call, and a callee-save register, which is a non-scratch register, will be saved and restored by the callee if it is used in the function. Without interprocedural analysis, a compiler cannot determine if a non-scratch register that is not used within a function is live. Therefore, a fast context switch point can be more precisely defined as an instruction in which no scratch registers are live. The 68020/68881 uses a total of 22 user-allocable registers which consists of 6 address registers, 8 data registers, and 8 floating point registers. Measurements were collected from a set of programs to determine the most effective combination of scratch and non-scratch registers. These combinations are shown in Table 2.

Table 2: User-Allocable Register Partitions

	Scratch	Non-Scratch
address	3	3
data	5	3
floating-point	4	4

4.3 Register Remapping

When modifying RTLs in order to produce more fast context switch points, it is important to ensure that run-time performance of the program does not degrade. Schemes which introduced additional instructions or memory references were initially attempted and resulted in a loss of performance since the benefit from additional fast context switch points was outweighed by the introduced overhead [Sny92]. An optimization that increases the total number of fast context switch points without performance degradation is called *register remapping*. This optimization is performed after all other optimizations have been invoked. First, an interference graph of the live ranges that have been allocated to scratch registers is built. Each live range of a scratch register is examined to determine if it interferes with non-scratch registers that have been used elsewhere in the function. If it does not interfere with one, then all references to the scratch register within the live range are replaced with the non-scratch register and the interference graph is updated. Using a non-scratch register that had not been used elsewhere in the function would result in the added overhead of saving and restoring this register at the beginning and end of the function, respectively. Live ranges of scratch registers were also prioritorized for remapping. This priority was based on the estimated length of time that the range could execute, which was calculated by the number of instructions in the live range and its loop nesting level. In other words, live ranges in a deeply nested structure with a large number of instructions were given the highest priority for remapping.

4.4 Gathering Measurements

Context switch measurements were gathered for three different types of systems: systems performing slow context switches, systems performing fast and slow context switches on unoptimized programs, and systems performing fast and slow context switches on optimized programs.[3] The length of the context switch intervals were varied to be 0.5k, 1k, 2k, 4k, 8k, and 16k instructions. Instruction, data, and unified cache measurements were obtained using each of these intervals along with the following different cache sizes: 1k, 4k, and 16k. For these experiments, when a context switch occurs, the entire cache is invalidated. Table 3 contains a list of the test programs used. The headings for the tables depicting the results shown in the following sections are defined in Table 4.

5 RESULTS

5.1 Memory References Avoided by Using Fast Context Switches

Each of the test programs displayed different characteristics when executed. These characteristics ranged from programs performing no fast context switches to programs performing only fast context switches. Also, some of the programs benefited from the remapping of live ranges, whereas others did not.

[3]An unoptimized program is one without register remapping, and an optimized program is one with register remapping.

Table 3: Test Programs

Program	Description
arraymerge	Merges arrays
cal	Calendar program
cpp	C preprocessor
des	DES encryption algorithm
dhrystone	Dhrystone benchmark
diff	Differential file comparator
fft	Fast fourier transformation
polly	Calculates total redundancy of sample dimensional vectors
queens	Array processing program
quicksort	Sort algorithm
sieve	Sieve benchmark
sed	Stream line editor
tbl	Format tables for nroff or troff
yacc	Parser generator

Table 4: Table Heading Descriptions

ALR	Average number of live registers per instruction
CMRS%	Percentage of context switch memory references saved
CSI	Context switch interval
FCS	Number of fast context switches
FCS%	Percentage of fast context switches performed
SCS	Number of slow context switches
TMRS%	Percentage of program's total memory references saved
ZLR%	Percentage of instructions that had zero live registers

By looking at the averages of the test programs shown in Table 5, one can see a higher frequency of fast context switches than slow context switches. Table 5 illustrates improvements from fast context switching with the first row for each context switch interval representing the values from the unoptimized programs and the second row representing the values from the optimized programs. The remapping provides somewhat better performance at run-time than the unoptimized test programs by increasing the percentage of fast context switches; these increases ranged from 0.6% to 2.8%. Remapping also increased the percentage of fast context switch points from 28.5% to 32.8% and reduced the average number of live registers per instruction from 2.2 to 2.0. It was also discovered that a small set of instruction types accounted for a large percentage of the fast context switch points [Sny92]. Most of these instructions occur near transfers of control. This indicates that a bit need not be sacrificed from all types of instructions to indicate fast context switch points. The percentage of memory references saved during a context switch ranged from 32.6% to 34.9%. It should be noted that since a fast context switch only avoids the saving and restoring of scratch registers, the maximum percentage of memory references that could be saved during a context switch for this experiment is 54.5%. The impact of reductions in context switch cost on total memory references depends on the frequency of context switches. In our experiments, the total number of memory references saved ranged from 0.1% for the longest context switch interval to 5.7% for the shortest context switch interval. The maximum percentage of total memory references that could be saved ranged from 0.3% for the longest context switch interval to 8.2% for the shortest context switch interval. Even though the register remapping did not substantially increase the number of fast context switches, a reduction of the context switch window could possibly lead to more of a relative improvement.

Table 5: Overall Fast Context Switches

		ALR	ZLR%
No Remapping		2.2	28.5
Remapping		2.0	32.8
CSI	FCS%	CMRS%	TMRS%
0.5k	61.0	33.2	5.4
	63.6	34.6	5.7
1k	61.4	33.4	2.9
	64.1	34.9	3.2
2k	60.9	33.2	1.5
	63.7	34.7	1.6
4k	61.0	33.3	0.7
	63.5	34.6	0.8
8k	61.9	33.7	0.4
	62.6	34.1	0.4
16k	59.8	32.6	0.1
	60.4	32.9	0.1

5.2 Cache Performance

The results from the cache performance analysis proved to be very interesting even though the relative changes were small when comparing the three different versions of the test programs. The following sections contain the summaries of results for the different sized instruction, data, and unified caches. The table that corresponds to each summary is comprised of columns for the context switch intervals (CSI), the cache hit ratios for the programs that only performed slow context switches, the cache hit ratios for the unoptimized programs that performed both fast and slow context switches, the relative changes in cache hit ratios with unoptimized fast context switching, the cache hit ratios for the optimized programs that performed both fast and slow context switches, and the relative changes in cache hit ratios with optimized fast context switching. The cache hit ratios for both the unoptimized programs and optimized programs are compared with the cache hit ratios of the versions performing slow context switches. A positive relative change in value indicates that the cache hit ratio was higher for the version performing both fast and slow context switches, and a negative value indicates that the cache hit ratio was lower.

5.2.1 Instruction Cache Comparisons

The results from all six context switch intervals and all three instruction cache sizes, which can be found in Table 6, indicate that performing fast context switches causes an improvement in instruction cache performance. In all six of the context switch intervals, the optimized test programs had a higher cache hit ratio than did the unoptimized test programs. The fact that the programs performing fast context switches had better instruction cache performances was probably due to many of the fast context switches taking place at the point of instructions that were transfers of control (i.e. calls, jumps, etc). These locations are an ideal place to perform a context switch since transfers of control also cause a change in locality of instruction references.

Table 6: Instruction Cache Performance

CSI	SCS Cache Hit Ratio	FCS Cache Hit Ratio	FCS Net Change	Optimized FCS Cache Hit Ratio	Optimized FCS Net Change
1k Cache					
0.5k	0.956110	0.957796	0.001686	0.958984	0.002874
1k	0.967148	0.968075	0.000927	0.969358	0.002210
2k	0.973827	0.974329	0.000502	0.975668	0.001841
4k	0.977795	0.978002	0.000207	0.979362	0.001567
8k	0.980276	0.980370	0.000094	0.981738	0.001462
16k	0.981685	0.981712	0.000027	0.983081	0.001396
4k Cache					
0.5k	0.961653	0.963805	0.002152	0.964303	0.002650
1k	0.975784	0.976984	0.001200	0.977396	0.001612
2k	0.984603	0.985307	0.000704	0.985644	0.001041
4k	0.990083	0.990419	0.000336	0.990685	0.000602
8k	0.993550	0.993715	0.000165	0.993939	0.000389
16k	0.995542	0.995597	0.000055	0.995794	0.000252
16k Cache					
0.5k	0.962221	0.964392	0.002171	0.964729	0.002508
1k	0.976561	0.977783	0.001222	0.978020	0.001459
2k	0.985677	0.986404	0.000727	0.986561	0.000884
4k	0.991455	0.991800	0.000345	0.991895	0.000440
8k	0.995135	0.995312	0.000177	0.995371	0.000236
16k	0.997262	0.997322	0.000060	0.997358	0.000096

5.2.2 Data Cache Comparisons

The results from the data cache measurements are found in Table 7. Even though the fast context switches did not improve the cache performance in most cases, the decrease in the cache hit ratios were less than 0.6%. When arguments are passed through parameters to functions, they are placed on the run-time stack. If a fast context switch occurred at the point of a call, then loading the parameters into registers in the called function will result in cache misses since the cache gets purged every time a context switch occurs.

Table 7: Data Cache Performance

CSI	SCS Cache Hit Ratio	FCS Cache Hit Ratio	FCS Net Change	Optimized FCS Cache Hit Ratio	Optimized FCS Net Change
1k Cache					
0.5k	0.762766	0.761444	-0.001322	0.760834	-0.001932
1k	0.788300	0.787688	-0.000612	0.786712	-0.001588
2k	0.815638	0.815067	-0.000571	0.813916	-0.001722
4k	0.828853	0.828951	0.000098	0.827706	-0.001147
8k	0.835334	0.835312	-0.000022	0.834023	-0.001311
16k	0.838354	0.838439	0.000085	0.837128	-0.001226
4k Cache					
0.5k	0.780115	0.778165	-0.001950	0.778672	-0.001443
1k	0.810616	0.808465	-0.002151	0.808882	-0.001734
2k	0.843570	0.840586	-0.002984	0.840979	-0.002591
4k	0.861037	0.858417	-0.002620	0.858807	-0.002230
8k	0.870489	0.867866	-0.002623	0.868272	-0.002217
16k	0.876499	0.874058	-0.002441	0.874479	-0.002020
16k Cache					
0.5k	0.783723	0.780215	-0.003508	0.780689	-0.003034
1k	0.815244	0.810946	-0.004298	0.811326	-0.003918
2k	0.849083	0.843510	-0.005573	0.843867	-0.005216
4k	0.867353	0.861834	-0.005519	0.862189	-0.005164
8k	0.877612	0.871805	-0.005807	0.872178	-0.005434
16k	0.888799	0.883006	-0.005793	0.883394	-0.005405

5.2.3 Unified Cache Comparisons

The results from the unified cache measurements are found in Table 8. There was a mixture of improvements and degradations in cache performance for fast context switching. This was expected since a unified cache combines both the instruction and data references.

Table 8: Unified Cache Performance

CSI	SCS Cache Hit Ratio	FCS Cache Hit Ratio	FCS Net Change	Optimized FCS Cache Hit Ratio	Optimized FCS Net Change
1k Cache					
0.5k	0.865689	0.873280	0.007591	0.873687	0.007998
1k	0.881030	0.888062	0.007032	0.888357	0.007327
2k	0.890950	0.897470	0.006520	0.897708	0.006758
4k	0.896919	0.903242	0.006323	0.903454	0.006535
8k	0.901296	0.907537	0.006241	0.907723	0.006427
16k	0.904053	0.910030	0.005977	0.910221	0.006168
4k Cache					
0.5k	0.895700	0.896315	0.000615	0.895844	0.000144
1k	0.918697	0.918285	-0.000412	0.917794	-0.000903
2k	0.934503	0.933007	-0.001496	0.932476	-0.002027
4k	0.944995	0.942592	-0.002403	0.942027	-0.002968
8k	0.953688	0.950548	-0.003140	0.949950	-0.003738
16k	0.959206	0.955477	-0.003729	0.954870	-0.004336
16k Cache					
0.5k	0.899024	0.901364	0.002340	0.901281	0.002257
1k	0.922908	0.925131	0.002223	0.924976	0.002068
2k	0.939550	0.941549	0.001999	0.941285	0.001735
4k	0.950843	0.952600	0.001757	0.952247	0.001404
8k	0.960261	0.961720	0.001459	0.961308	0.001047
16k	0.966341	0.967459	0.001118	0.967036	0.000695

6 FUTURE WORK

6.1 Other Architectures

The results obtained related directly to the Motorola 68020/68881 processor. This processor is an example of a CISC architecture and only has a total of 22 user-allocable registers. Most new RISC architectures have a larger number of registers.[4] For example, the MIPS R3000 has a total of 69 registers, the IBM RS/6000 has a total of 100 registers, and the SPARC has a total of 174 registers, and these large number of registers constitute a larger register state to be saved and restored for each context switch. It would be very informative to find out the benefits of performing fast context switches on these types of architectures since a larger percentage of memory references could be saved.

By applying the same proportion of scratch registers used in the experiment for these RISC architectures, the percentages of the total number of memory references saved can be extrapolated from Table 5. These results are shown in Table 9. The first row in each interval represents the estimated percentage of memory references saved for the unoptimized test programs and the second row represents the estimated results for the optimized test programs. Deriving the values for the R3000 and RS6000 processors was straightforward since these processors use a single register set. For the R3000 and RS6000 the following assumptions were made. The percentage of fast context switches remained the same for each context switch interval and the percentage of scratch registers remained the same at 54.5%. The number of memory references within each program was also assumed not to change, although they would likely decrease due to the additional registers. The SPARC processor, on the other hand, was handled in a different manner since it uses multiple register sets. The estimated results for the SPARC were based on the following assumptions. The SPARC has a total of 167 user-allocable registers and 128 of these are used for the overlapping register windows. If there are 8 windows, then there are 24 registers per window. The first 8 registers in a window overlap with the previous window, and the last 8 overlap with the following window. This means that the first 16 registers in a window are non-scratch since their values will be retained across function calls, and the last 8 are scratch since their values cannot be guaranteed upon return from the called function. The other registers are comprised of 7 global registers and 32 floating-point registers. On average, 3 register windows are saved and restored for each context switch. Therefore, a total of 95 registers are saved and restored with each slow context switch. Of the 56 registers used in the register windows only the last

[4]For this experiment, we did not have access to a RISC machine with a single register set for which EASE could obtain measurements.

Table 9: Extrapolated Percentage of Memory References Saved

CSI	TMRS% MIPS R3000	TMRS% IBM RS6000	TMRS% SPARC
0.5k	12.4	15.4	8.5
	13.0	16.2	8.9
1k	7.5	10.0	5.4
	8.3	10.9	5.9
2k	4.2	5.8	3.2
	4.5	6.2	3.4
4k	2.1	2.9	1.6
	2.4	3.3	1.8
8k	1.2	1.7	0.9
	1.2	1.7	0.9
16k	0.3	0.4	0.2
	0.3	0.4	0.2

8 are considered scratch. For the rest of the 39 registers only 54.5% are considered scratch. The percentage of fast context switches also remained the same for the SPARC processor. One can conclude that fast context switching is more effective as the number of registers to save and restore increases.

The frequency of fast context switch points on the RISC architectures will probably differ from the 68020/68881. First, in addition to having more registers, there are fewer types of registers (no address registers). This would probably result in having a larger ratio of scratch to non-scratch registers for the most effective combination [Dav91b]. Also, most RISC machines pass a limited number of arguments to functions through scratch registers. Fast context switches may have to be redefined to include saving and restoring some of these registers.

7 CONCLUSION

While performing a context switch, the CPU cannot be used for the executing processes in the system. This overhead is unwanted, but necessary, in a real-time environment that uses preemptive scheduling. By using threads most of the context switch overhead is reduced to only the saving and restoring of the register states. This will help to reduce some of the overhead in a context switch, but with recently introduced architectures using considerably more registers, the total amount of time to save and restore process states will still impact the overall performance.

7.1 Performing Fast Context Switches

The results from the research indicated that the potential for performing fast context switches does exist. The run-time analysis exhibited a high percentage of fast context switches that ranged from 59.8% to 64.1% for the different context switch intervals. With the current register partitioning, this means that a system performing fast context switches could reduce the total overhead by avoiding anywhere from 32.6% to 34.9% of the memory references that are used to save and restore registers due to context switches. With respect to the total number of memory references from a program, the amount of memory references saved is relatively small, but in hard real-time systems meeting time constraints is critical, and a more efficient context switch provides a better opportunity for processes to meet their deadlines. Systems with more registers than the 68020/68881 could potentially reduce the total number of memory references by an even larger percentage.

7.2 Context Switch Interval

The context switch interval affected both the number of memory references avoided and the cache performance. The effects on the percentage of fast context switches were minor. By reducing the context switch interval from 16k to 0.5k, the overall percentage of fast context switches increased by at most 2.8%. For fast context switching to make a significant reduction in the total number of memory references, the context switch interval had to be relatively short. Without a short context switch interval the benefits of performing fast context switches were barely noticeable.

7.3 Remapping Registers

The optimization of remapping registers was implemented to never introduce additional instructions or memory references that would degrade the performance of the code generated for a program. The overall results indicated a slight improvement for the test programs that used remapping over the test programs that did not use remapping. Although not having a large impact on the reduction of overhead, this optimization proved to be useful. The remapping of registers may have been more effective if the contexts switch window were smaller.

7.4 Relative Cache Performance

The results from the cache measurements indicated that the overall changes were negligible. In all cases the net change was less than one percent. There was no clear distinction on whether or not the test programs performing fast context switches had better cache performances over the test programs performing only slow context switches. In the instruction cache, the programs that performed fast context switches had better cache performances, but in the data cache the test programs that performed slow context switches had better cache performances. The unified cache had some results which favored the fast context switching and other results that favored slow context switching. Of course, the cache analysis was for the purpose of comparing the relative changes between these three different types of environments. These changes do not indicate that one method of performing context switches provides superior cache performance over the other methods.

References

[Aga88] Anant Agarwal, John Hennessy, and Mark Horowitz, "Cache Performance of Operating System and Multiprogramming Workloads", *ACM Transactions on Computer Systems*, November 1988, 6(4):393–431.

[And91] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska, "The Interaction of Architecture and Operating System Design", *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991, 108–120.

[Dav91a] Jack W. Davidson and David B. Whalley, "A Design Environment for Addressing Architecture and Compiler Interactions", *Microprocessors and Microsystems*, November 1991, 15(9):459–472.

[Dav91b] Jack W. Davidson and David B. Whalley,' 'Methods for Saving and Restoring Register Values across Function Calls", *Software–Practice & Experience*, February 1991, 21(2):149–165.

[Gro90] Randy D. Groves and Richard Oehler, "RISC System/6000 Power Architecture", *Microprocessors and Microsystems*, July/August 1990, 14(6):357–366.

[Lev90] Shem-Tov Levi and Ashok K. Agrawala, *Real-Time Systems Design*, McGraw–Hill, 1990.

[Mil90] D. Richard Miller and Donna J. Quammen, “Exploiting Large Register Sets”, *Microprocessors and Microsystems*, July/August 1990, 14(6):333–340.

[Mog91] Jeffrey C. Mogul and Anita Borg, “The Effect of Context Switches on Cache Performance”, *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991, 75–84.

[Nut92] Gary J. Nutt, *Centralized and Distributed Operating Systems*, Prentice–Hall, Inc., 1992.

[Qua91] Donna J. Quammen and D. Richard Miller, “Flexible Register Management for Sequential Programs”, *The 18th Annual International Symposium on Computer Architecture*, May 1991, 19(3):320–329.

[Sny92] Jeffrey S. Snyder, “Fast Context Switches”, Masters Thesis, Florida State University, 1992.

[Spa90] “SPARC: Architecture to Implementations”, *Microprocessors and Microsystems*, July/August 1990, 14(6):417–420.