Appears in *Proceedings of the 20th Annual International Symposium on Computer Architecture, May 1993*

Register Relocation: Flexible Contexts for Multithreading

Carl A. Waldspurger *   William E. Weihl *

MIT Laboratory for Computer Science
Cambridge, MA 02139

Abstract

Multithreading is an important technique that improves processor utilization by allowing computation to be overlapped with the long latency operations that commonly occur in multiprocessor systems. This paper presents *register relocation*, a new mechanism that efficiently supports flexible partitioning of the register file into variable-size contexts with minimal hardware support. Since the number of registers required by thread contexts varies, this flexibility permits a better utilization of scarce registers, allowing more contexts to be resident, which in turn allows applications to tolerate shorter run lengths and longer latencies. Our experiments show that compared to fixed-size hardware contexts, register relocation can improve processor utilization by a factor of two for many workloads.

1 Introduction

Multithreading is an important technique for tolerating latency in multiprocessor systems [3, 7, 19, 21]. Support for multiple contexts and rapid context switching permits high latency operations such as remote memory references and synchronization events to be overlapped with computation, which improves processor utilization. Because the number of registers required by thread contexts varies across applications and among threads within a single application, the ability to partition the register file into contexts of varying sizes enables more efficient use of the available registers. We present *register relocation*, a simple mechanism that efficiently supports flexible, variable-size processor contexts with minimal hardware support. The register relocation hardware should affect only the instruction decode stage of the processor pipeline, and would be a simple addition to many existing architectures.

Existing multithreaded architectures typically provide several separate, fixed-size contexts managed in hardware. This fixed, inflexible division of the register file often results in a significant waste of scarce high-speed registers. More efficient partitioning of the register file would permit a larger number of resident contexts. Since the optimal number of contexts is both application- and machine-dependent [19], enabling more resident contexts will allow programs with sufficient parallelism to tolerate longer latencies and shorter run lengths. Alternatively, better utilization of the register file would permit a smaller register file to support a given number of contexts, which has architectural advantages in terms of chip area and processor cycle-time [11].

Register relocation adheres to the RISC philosophy [17] by maintaining a simple processor architecture and relying upon the compiler and runtime system to manage the allocation and use of contexts. We discuss several runtime-level software techniques that exploit the register relocation hardware, managing the division of the register file into contexts in software. Because the size of contexts is not dictated by the hardware, there is considerable flexibility in the use of the register file to support multithreading. Possible organizations range from *static* partitioning into contexts with fixed or varying sizes to *dynamic* allocation of contexts with varying sizes as needed. The flexibility to provide a better match between application requirements and the organization of the register file into contexts enables better utilization of scarce registers, which allows more resident contexts and higher processor utilization. Our experiments show that register relocation can improve performance by a factor of two or more for many workloads.

*E-mail: {carl,weihl}@lcs.mit.edu. The first author was supported by an AT&T USL Fellowship and a grant by the MIT X Consortium. This research was also supported by the National Science Foundation under grant CCR-8716884, by the Defense Advanced Research Projects Agency (DARPA) under contract N00014-91-J-1698, by grants from AT&T and IBM, and by an equipment grant from DEC. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. government.

Improvements in processor utilization can be expected whenever more contexts are able to remain loaded through an efficient partitioning of the register file. This is likely to be a common case, since many threads cannot make effective use of a large number of registers. Moreover, most programs exhibit decreasing marginal performance improvements as the number of available registers is increased. For example, one performance study [9] revealed a pattern of decreasing marginal savings in memory references as the number of available registers is increased. Another [5] found that even in workloads containing programs with large basic blocks, the degradation in execution time given 16 registers instead of 32 averaged only 12%; the improvement given more than 32 registers averaged only 1%. This study also showed that sophisticated code generation strategies require fewer registers. We believe that these effects are likely to be even more pronounced in systems with many fine-grained threads and sophisticated optimizing compilers.

In the next section, we discuss the hardware and software support required for register relocation. Section 3 discusses the results of several quantitative experiments comparing register relocation to conventional multithreaded architectures. In Section 4, we examine related work. Extensions and directions for future research are discussed in Section 5. Finally, we summarize our conclusions in Section 6.

2 Register Relocation

The register relocation mechanism is very simple. Instruction operands specify *context-relative* register numbers, which are numbered consecutively starting with register 0. These context-relative register numbers are dynamically combined with a special *register relocation mask* (`RRM`) to form absolute register numbers that are used during instruction execution. A bitwise `OR` is used as the combining operation.

The `OR` operation permits a flexible division between the number of `RRM` bits treated as the register relocation *base*, and the number of register operand bits treated as the register relocation *offset*, as shown in Figure 1. This simple mechanism allows the register file to be partitioned into a collection of variable-size contexts. For example, the register file can be divided into a small number of large contexts, as is conventionally done in hardware. Alternatively, the register file can be divided into a large number of small contexts, providing support for many fine-grained threads. The register file can also be divided, statically or dynamically, into different combinations of context sizes, supporting a mix of both coarse and fine-grained threads.

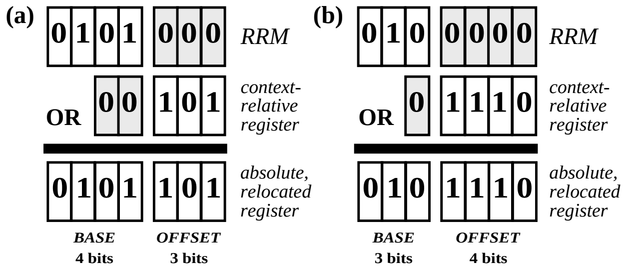

Figure 1: Register Relocation Examples. In both examples, there are a total of 128 general registers, the RRM is 7 bits wide, and register operands are 5 bits wide. The shaded portions of registers are effectively unused. (a) The RRM provides relocation for a context of size 8. Context-relative register 5 is relocated to absolute register 45. (b) The RRM provides relocation for a context of size 16. Context-relative register 14 is relocated to absolute register 46.

2.1 Hardware Support

A *register relocation mask* (`RRM`) is maintained in a special hardware register. The `RRM` register requires $\lceil \lg n \rceil$ bits for a processor architecture with n general registers. A special `LDRRM R` instruction is used to set the `RRM` from the low-order $\lceil \lg n \rceil$ bits of register `R`. Depending on the organization of the processor pipeline, there may be one or more delay slots following a `LDRRM` instruction.

RISC architectures typically employ a *fixed-field decoding* scheme in which register operands are always specified at the same location within an instruction [18]. During every instruction decode, a bitwise `OR` operation is performed with each of the instruction's register operand fields and the `RRM`, yielding *relocated* register operand fields, as shown in Figure 2. After the instruction decode phase, no additional work needs to be performed.

Another hardware change that would be necessary in some architectures is to widen the internal paths that carry the register operands specified by an instruction. This is because a relocated register operand requires $\lceil \lg n \rceil$ bits to address the entire register file, while a register operand field in an instruction may only be able to address a smaller portion of the register file, due to limitations on the width of a machine instruction. We will denote the width of an instruction

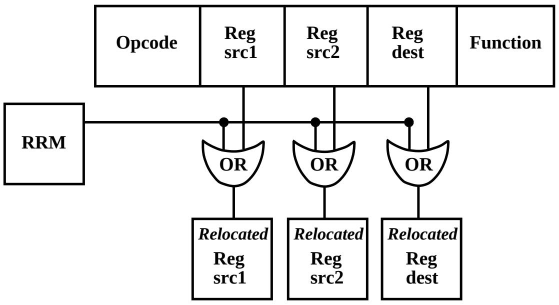

Figure 2: Register Relocation Hardware. During instruction decode, a bitwise `OR` is performed with each of the instruction's operands and the `RRM`, yielding relocated register operands.

register operand by w, which constrains the number of addressable registers and places an upper bound of 2^w on the size of a single context.

2.2 Context Scheduling

Unlike multithreaded architectures with hardware control over scheduling [2, 10, 12, 22], we schedule contexts entirely in software. We have developed an approach for very fast context switching that does not require hardware support or the use of a separate scheduler context. Sample assembly-language code for this fast context switch is listed in Figure 3. The scheduler "ready queue" for loaded contexts is implemented as a circular linked list of register relocation masks. This list is maintained by storing a `NextRRM` mask in each resident context. A transfer of control to the next runnable thread context is implemented as follows:

- Store the current program counter in a register associated with the current context. In the listed code, context-relative register `R0` is used to store the `PC`.
- Execute a `LDRRM` instruction to install the register relocation mask for the next thread context. Depending on the organization of the processor pipeline, there may be one or more delay slots following this instruction. In the listed code, `R2` is used to store the `NextRRM` relocation mask, and there is one delay slot.
- If necessary, save any processor state that must be restored when the current thread is resumed. When the newly installed `RRM` becomes active, restore any processor state associated with the new context.[1] In the listed code, a processor status word (`PSW`) is stored in `R1`.
- Jump to the program counter associated with the new context.

```
| Context-Relative Register Conventions
|
|     R0:  thread program counter (PC)
|     R1:  processor status word (PSW)
|     R2:  mask for next thread (NextRRM)

fault:  | jump and link; save next PC in r0
        jalr    yield, r0
          :

yield:  | install new relocation mask,
        | save old status register
        | (single ldrrm delay slot)
        ldrrm  r2
        mov    PSW, r1

        | restore new status register,
        | execute code in new context
        mov    r1, PSW
        jmp    r0
```

Figure 3: Context Switch Code. The instruction labelled `fault` may be explicit (as shown), or the result of a trap. The register move (`mov`), unconditional jump (`jmp`), and jump and link (`jalr`) instructions are similar to those found in many RISC architectures. The `LDRRM` instruction is described in Section 2.1.

This approach requires approximately 4 to 6 RISC cycles, depending on the number of `LDRRM` delay slots and the need to save and restore processor status information. More sophisticated scheduling policies can also be implemented by altering the order in which contexts are linked together by their `NextRRM` masks. For example, separate linked lists of register relocation masks could be maintained to implement different thread classes or priorities. Such flexibility is possible because context scheduling is under software control.

[1] On some architectures, condition codes such as carry flags must be preserved. This is unnecessary on architectures which don't use condition codes, such as the MIPS R3000. At the other extreme, architectures such as the SPARC may require expensive traps to manipulate this state.

2.3 Context Allocation

The register relocation mechanism can allocate a context with size 2^k registers, for $0 \leq k \leq w$; the maximum context size is limited to 2^w by the number of address bits used for register operands. However, the minimum context size should be large enough to maintain some state other than a program counter. For example, practical context sizes for an architecture with 256 registers and 6-bit register operands would be 4, 8, 16, 32, and 64 registers.

Context allocation is performed entirely in software, and is thus extremely flexible. One option is to partition the register file statically into contexts (with identical or differing sizes) for a particular application, which makes allocation and deallocation extremely inexpensive. Another option is to partition the register file dynamically into contexts of varying sizes as needed.

For the experiments discussed in Section 3, we coded general-purpose dynamic context allocation and deallocation routines for a RISC architecture with 128 registers. The implementation employs simple shift and mask operations to search an allocation bitmap for available contexts. Linear search is used for some context sizes, and binary search is used for others. General-purpose allocation executes in approximately 25 RISC cycles, and general-purpose deallocation requires fewer than 5 RISC cycles.[2] Sample C code for general-purpose allocation is listed in Appendix A.

2.4 Compiler Support

Compilers can generate code as usual, and may assume that the available registers are numbered from 0 to $2^w - 1$. Although the compiler is permitted to use all 2^w registers, many threads will require fewer registers. For each thread, the compiler must inform the runtime system about the number of registers that the thread requires, which can be determined by traversing the thread call graph. In systems that support separate compilation, the compiler will need to provide this information to the linker. However, this should not present any difficult challenges; more sophisticated cooperation between compilers and linkers has already been demonstrated for register allocation [23].

The register relocation mechanism also presents some interesting opportunities for compiler optimizations. As noted in Section 1, most programs realize decreasing marginal improvements from additional registers. A compiler can thus make tradeoffs between allocating additional registers to a thread or using fewer registers to enable more resident contexts. For example, a compiler may normally achieve some marginal benefit by allocating 17 (versus 16) registers to a thread; there is no reason to conserve registers if they would otherwise be wasted. However, due to the power-of-two constraint on context sizes, a thread that uses 17 registers will require a context of size 32. The 15 extra registers that are consumed could instead be used to support a higher degree of multithreading, and the corresponding increase in processor utilization is likely to exceed the original gain from using an extra register.

Finally, by guaranteeing not to use any additional registers, the compiler – not the hardware – is responsible for ensuring protection among thread contexts. However, similar protection issues arise for memory locations due to the execution of multiple threads within a single address space. Note that we are assuming that threads are associated with a single application, and hence are logically related. Thus, erroneous register overwrites are not inherently more problematic than memory overwrites; they simply occur at different levels in the memory hierarchy. In order to facilitate low-level debugging of compilers and runtime system routines, a separate tool could be used to statically check executables or object files for most violations of context boundaries.[3]

2.5 Context Loading

The register relocation mechanism requires the compiler to determine the number of registers needed by each thread, as described in Section 2.4. This information can also be exploited to save or restore the exact number of registers used by a thread when its context is loaded or unloaded. The runtime system can provide one context unload routine that successively stores registers numbered $2^w - 1$ to 0 to memory, with 2^w separate entry points corresponding to every possible number of context registers used by threads. Similarly, a single context load routine with multiple entry points can be provided to successively load registers from memory.

[2] If an operation such as the Motorola MC88000's `FF1` instruction is available that can find the first bit set in a word, then general-purpose allocation can be performed in approximately 15 RISC cycles.

[3] One of the referees suggested the use of MUXs to select each bit from either the `RRM` or the register operand; this might be faster in CMOS than an `OR`, and would also prevent a thread from accessing registers outside its allocated context. Alternatively, hardware could be added for "bounds checking" on contexts.

Parameter	Description	(units)
R	average run length	(cycles)
L	average fault latency	(cycles)
S	context switch cost	(cycles)
F	register file size	(registers)
C	required context size	(registers)

	Cost (cycles)	
Operation	Flexible	Fixed
context allocate (succeed)	25	0
context allocate (fail)	15	0
context deallocate	5	0
context load/unload	C	C
thread queue insert/remove	10	10

Figure 4: Parameters and Assumptions. The first table describes the experimental parameters. The second table lists the cost assumptions used for both the register relocation (*Flexible*) and conventional fixed-size contexts (*Fixed*) architectures.

3 Experiments

We ran a large number of experiments, over a wide range of system parameters, to evaluate the register relocation mechanism. The experiments focus on a single multiprocessor node executing multiple threads with stochastic run lengths and varying fault latencies. We assume a coarsely multithreaded processor architecture similar to APRIL [2], which switches contexts only when a high-latency operation such as a remote cache miss or synchronization fault occurs. We conducted our experiments using PROTEUS [6], a high-performance parallel architecture simulator, which we modified to support multiple contexts.

Below we discuss several experiments involving only cache faults, and others involving only synchronization faults. We also ran experiments involving both types of faults, with similar results; the main effect was to increase the overall fault rate. The data presented in this paper is representative of our experimental results; additional experiments appear in [24].

3.1 Parameters and Assumptions

In each experiment, a supply of synthetic threads was created with particular fault rates and fault service latencies. All threads executed until completion, and statistics were extracted over a substantial fraction of the execution that avoided transient startup and completion effects.[4] Our experimental parameters are summarized in Figure 4.

For each set of parameters, we performed two experiments: one to simulate a conventional multithreaded processor architecture with fixed hardware contexts, each containing 32 registers, and another to simulate an architecture using the register relocation mechanism. In both experiments, local thread queue insertion and removal operations cost 10 cycles, and loading contexts from memory and unloading contexts to memory cost 1 cycle per register. An additional charge of 10 cycles was assessed for the software overhead of blocking and unblocking contexts when loading and unloading.

For register relocation, successful context allocation and deallocation cost 25 and 5 cycles, respectively, and unsuccessful context allocation was charged 15 cycles. These costs are consistent with the general-purpose dynamic allocation routines listed in Appendix A. For the conventional hardware architecture with fixed contexts, these costs were all set to 0, assuming some hardware support for context scheduling. This assumption was deliberately conservative for comparison with the register relocation approach.

3.2 Tolerating Cache Faults

The experiments described in this section explore the use of multithreading to hide the latency associated with remote memory references. In each experiment, the average run length between cache faults (R) is geometrically distributed, and the average cache fault latency (L) is constant. Thus, there is a fixed probability of a cache miss on each execution cycle, and network response time is uniform, which is reasonable for lightly loaded networks. These distributions are also consistent with the assumptions and models used in earlier studies [3, 19]. The context switch cost is set to $S = 6$ cycles, which is consistent with the code presented in Figure 3, and better than the 11 cycle cost incurred by the current APRIL implementation [2]. To avoid effects due to the selection of a particular thread unloading policy, contexts are never unloaded.

Figure 5 summarizes many experiments; each data point represents a separate simulation. The graphs plot *efficiency* (i.e., processor utilization) vs. memory latency for a family of curves corresponding to various run lengths. The solid curves denote results for fixed-size hardware contexts, and the dotted curves denote

[4] We also collected statistics from entire runs; these differed only slightly from the statistics that excluded transients.

results for register relocation. For these experiments, the number of registers required by each context (C) is uniformly distributed between 6 and 24 registers. Due to the power-of-two constraint on context sizes for register relocation, these experiments are biased toward large contexts. Despite this bias, register relocation consistently outperforms conventional fixed-size contexts, resulting in significantly higher efficiencies over a wide range of values for L and R. Thus, even for workloads consisting of many large contexts and few small contexts, more contexts remain resident, improving processor utilization.

3.3 Tolerating Synchronization Faults

This section describes experiments that examine the use of multithreading to hide the latency associated with synchronization events. In each experiment, the average run length between synchronization faults (R) is geometrically distributed, and the average synchronization fault latency (L) is exponentially distributed. Thus, there is a fixed probability of a synchronization fault on each execution cycle, and wait times for synchronization are exponentially distributed, which is reasonable for producer-consumer synchronization [14]. The context switch cost is set to $S = 8$ cycles, which is 2 cycles more than the cost used in Section 3.2. This allows for simple bookkeeping and test operations (e.g., an add and conditional branch) which can be used to implement a thread unloading policy. The thread unloading policy used in these experiments is a competitive, two-phase algorithm [14]. A context is unloaded when the cost of repeated, unsuccessful attempts to continue execution equals the cost of unloading and blocking the context. Note that the cost assessed for loading and unloading a context is based on C, the number of registers *required* by the context (see Section 2.5), not on the size of the *allocated* context; this is true for all our simulations. Since conventional architectures with fixed-size contexts typically save and restore all registers allocated to a context, including unused ones, our experiments conservatively overestimate the performance of conventional approaches using hardware contexts.

Figure 6 summarizes the results for synchronization faults; each data point represents a separate simulation. The graphs plot efficiency vs. synchronization latency for a family of curves corresponding to various run lengths. The solid curves denote results for fixed-size hardware contexts, and the dotted curves denote results using register relocation. For these experiments, the number of registers required by each context (C) is uniformly distributed between 6 and

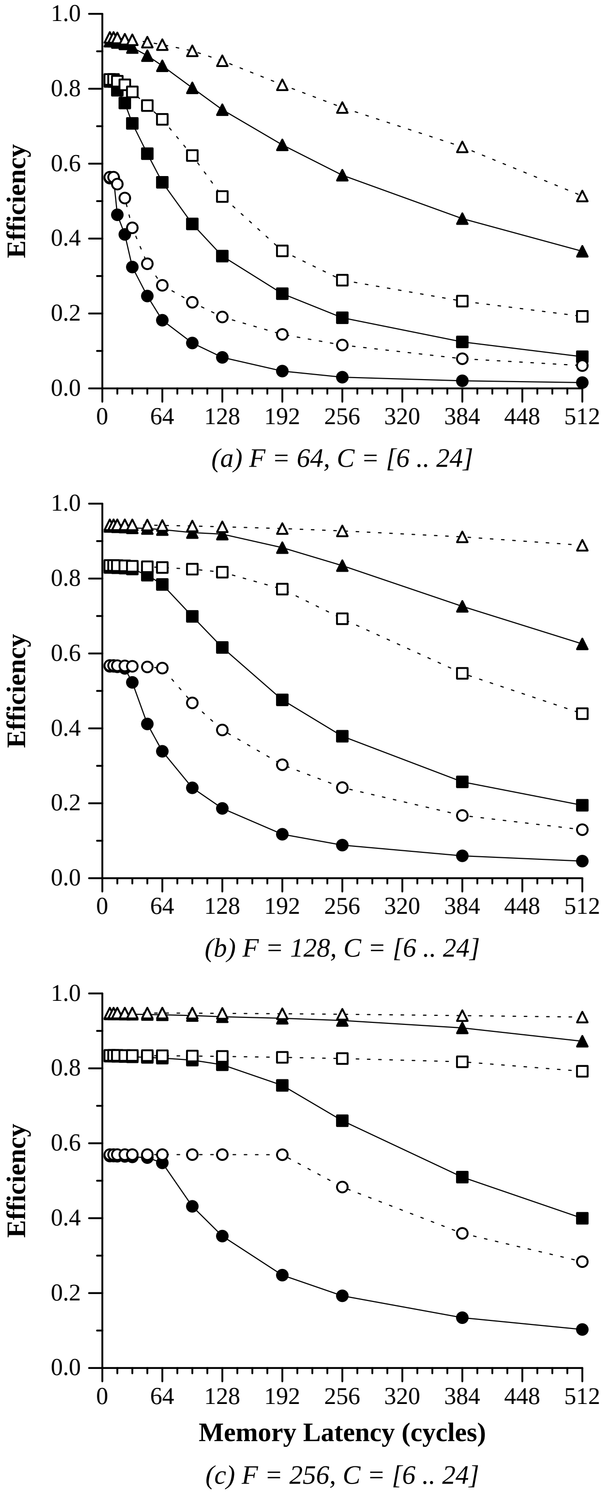

Figure 5: Cache Faults. Efficiency for F = 64, 128, and 256 registers, and C uniformly distributed from 6 to 24 registers. Curves: solid – fixed-size contexts, dotted – register relocation. Data points: circles – $R = 8$, squares – $R = 32$, triangles – $R = 128$.

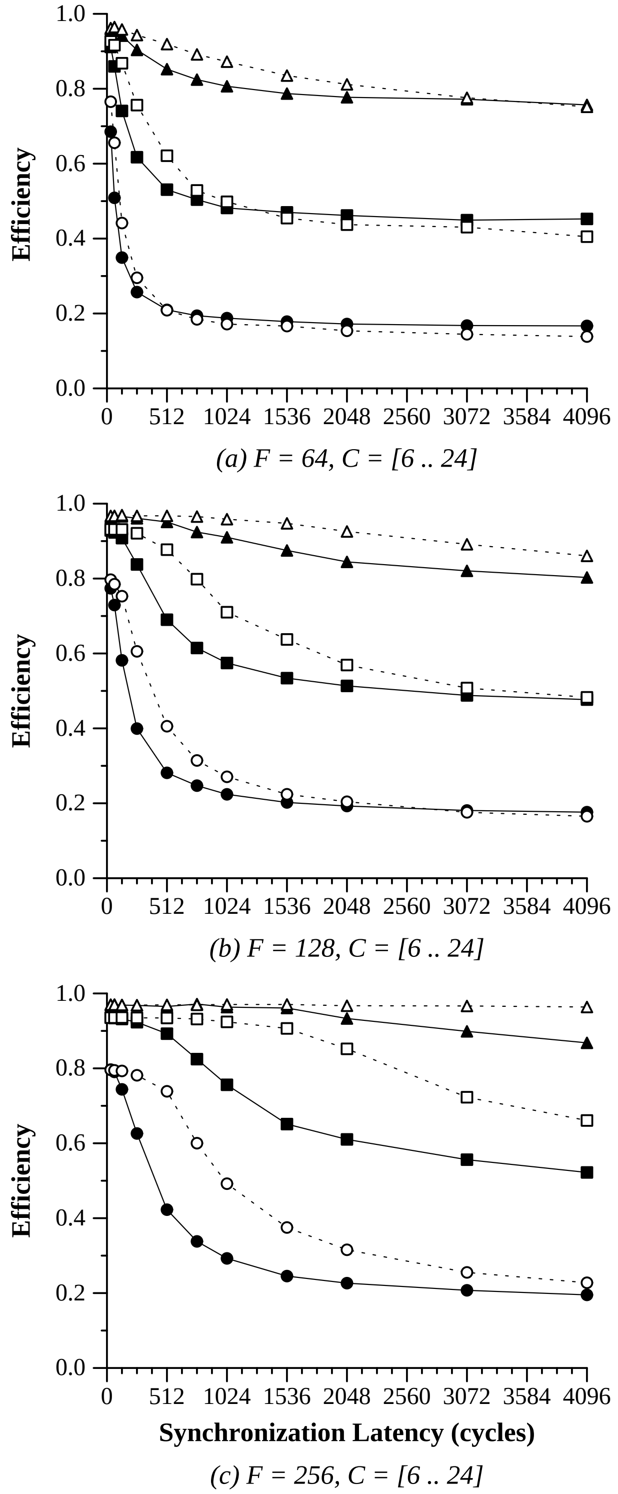

Figure 6: Synchronization Faults. Efficiency for $F = 64$, 128, and 256 registers, and C uniformly distributed from 6 to 24 registers. Curves: solid – fixed-size contexts, dotted – register relocation. Data points: circles – $R = 32$, squares – $R = 128$, triangles – $R = 512$.

24 registers. As explained earlier, this distribution is biased toward large contexts.

Register relocation improves processor utilization for virtually all parameter values. The only notable exception is provided by Figure 6(a), in which relatively large contexts are competing for a small register file ($F = 64$). As L increases, the advantages provided by register relocation diminish, and fixed-size hardware contexts marginally outperform register relocation for large L. This is due to the software overhead associated with context allocation in the register relocation approach. When the register file is small and contexts are large, contexts are continually loaded and unloaded for small R and large L, resulting in a large number of allocation operations. Re-executing the experiments in Figure 6(a) with lower allocation costs confirmed this explanation; in this case register relocation consistently outperformed the fixed-size contexts. Recall that allocation costs were set to 0 for the fixed-size hardware contexts in order to conservatively overestimate the performance of conventional approaches. If workload parameters similar to those in Figure 6(a) were common, a different, specialized allocation policy could be adopted. For example, two sets of context sizes could be implemented extremely cheaply; an allocation bitmap for contexts of size 16 and 32 can be encoded in four bits, and a direct lookup table indexed by this bitmap could be used to allocate contexts. The flexibility of performing allocation in software makes such schemes possible.

Nevertheless, even with a general-purpose dynamic allocation policy, register relocation still results in substantial efficiency gains over a wide range of values for L and R. Thus, for most workloads, we expect register relocation to enable more resident contexts, making it possible to improve processor utilization by tolerating long synchronization latencies.

3.4 Discussion

We also performed numerous experiments similar to those presented above, using homogeneous context sizes $C = 8$ and $C = 16$. The results were similar to those presented in Figures 5 and 6, but the relative improvements due to register relocation were often substantially larger [24]. This is not surprising, since the primary effect of smaller context sizes is to increase the number of contexts that can be supported by a given register file.

Register relocation significantly outperforms fixed-size hardware contexts over a wide range of system parameters, sometimes by huge margins. By providing the flexibility to efficiently partition the register file, scarce register resources are better utilized, allowing more contexts to remain resident. The number of resident contexts has a dramatic impact on processor utilization. As run lengths decrease and latencies increase, more contexts are needed to prevent the processor from idling. This trend is clearly indicated by the data presented in Figures 5 through 6, and can also be explained analytically.

A simple mathematical analysis of multithreaded architectures [19] reveals that for constant run lengths and latencies, processor efficiency can be determined from the parameters R, L, and S.[5] When there is always a resident context that is ready to execute, the processor is *saturated*, and its efficiency is independent of L: $\mathcal{E}_{sat} = \frac{R}{R+S}$. When the number of resident contexts is below the saturation point, the processor will not be fully utilized, and its efficiency is *linear* in the number of resident contexts (N): $\mathcal{E}_{lin} = \frac{NR}{R+S+L}$.

Thus, processor efficiency increases linearly in the number of resident contexts until saturation, after which it remains constant. From these two equations, we find that for $N < 1 + \frac{L}{R+S}$, processor efficiency is in the linear region. Given current trends toward large parallel machines and extremely fast processors, we expect R to decrease and L to increase, requiring a large number of contexts before processor efficiency saturates. Thus, systems with a small number of hardware contexts are likely to operate in the linear regime, where register relocation can substantially improve performance.

Some earlier studies have suggested that only a small number of contexts (2 to 4) is required to achieve high processor utilization [2, 25]. Although it is true that a small number of contexts significantly boosts efficiency for relatively low L and sufficiently high R, the optimal number of contexts is both application- and machine-dependent [19]. For example, the Horizon architecture [22] provides 128 hardware contexts in order to tolerate long latencies and short run lengths. Moreover, even proponents of a small number of hardware contexts agree that a large supply of runnable, loaded contexts is needed to tolerate synchronization latencies, which are typically much longer than latencies for remote memory access. For example, additional hardware for *dribbling registers* is currently being explored by the APRIL designers for tolerating longer latencies [20].[6] Our register relocation mechanism provides a simple, effective alternative to increasingly complex and specialized hardware solutions.

[5] A considerably more complex analysis is also presented in [19] that accounts for stochastic run lengths. However, the simpler equations for the deterministic case still provide a reasonable approximation for processor efficiency.

[6] The dribbling registers idea is completely orthogonal to the register relocation mechanism.

4 Related Work

Most multithreaded processor architectures employ hardware-intensive and inflexible mechanisms for multithreading. Our approach is software-intensive, and attempts to minimize the hardware required to support multithreading efficiently and flexibly. In this section we compare register relocation in more detail to other approaches.

A number of processor architectures with multiple hardware contexts have been proposed. *Finely* multithreaded processors, such as the Denelcor HEP [21], execute an instruction from a different thread on each cycle. A drawback to this approach is that the interleaving of threads in the processor pipeline degrades single-thread performance. Also, these processors require a steady supply of runnable threads that can be interleaved cycle-by-cycle to keep the processor pipeline busy. The more recent MASA architecture [10] also suffers from this problem. The Horizon and Tera architectures [4, 22] also switch among instruction streams on every cycle, but allow several instructions from the same thread to co-exist in the pipeline.

Coarsely multithreaded processors, such as APRIL [2], execute larger blocks of instructions from each thread, and typically switch contexts only when a high-latency operation occurs. A drawback to this approach is that a context switch typically bubbles the processor pipeline, degrading multithreaded performance. Hybrid dataflow / von-Neumann architectures [12, 15] also have pipelines that only contain instructions from a single thread, and typically provide support for hardware task queues.

Most of the architectures described above maintain a fixed, inflexible division of high-speed register resources into multiple contexts. Our register relocation mechanism supports coarse multithreading, but permits unusual flexibility in the organization of the register file by managing contexts in software. Context scheduling is also performed entirely in software, yet only a few RISC cycles are required to implement a context switch.

The AMD Am29000 processor [1] implements a base plus offset form of register addressing[7] that could be used to support multiple variable-size contexts. An `ADD` operation for register addressing is more general than our proposed `OR` operation for register relocation,

[7] The Denelcor HEP provided a similar capability.

and eliminates the power-of-two constraint on context sizes. However, an `ADD` is much more expensive than an `OR` in terms of hardware and time on the critical path. Moreover, the software for managing arbitrary-size contexts is likely to be more complex.

A completely different approach is the Named State Processor [16], which replaces a conventional register file with a *context cache*. The context cache binds variable names to individual registers in a fully associative register file, and spills registers only when they are immediately needed for another purpose. Our register relocation mechanism supports a binding of variable names to contexts that is finer than conventional multithreaded processors, but coarser than the context cache approach.

5 Extensions and Future Work

We are currently exploring a variety of issues related to register relocation and multithreading.

5.1 Software-Only Approach

We have devised a related approach for multithreading that requires *no* hardware support, and can be used with many existing processors. The basic idea is to have the compiler generate *multiple versions* of code that use disjoint subsets of the register file. Thus, register relocation is effectively performed at compile-time. This scheme has the obvious disadvantage of code expansion. However, the restrictions on context sizes no longer apply, and *any* partitioning of the register file is possible.

We performed some simple experiments by modifying `gcc`, the GNU C compiler, to investigate this scheme on a uniprocessor using the MIPS R3000 architecture. Our preliminary results were encouraging, but because of the limited number of general registers on the MIPS architecture, the technique was not practical for more than two contexts.[8]

5.2 Cache Interference Effects

Threads sharing a common cache can interfere with each other. Several studies have indicated that most cache interference is destructive, increasing the cache miss ratio [19, 25]. However, Agarwal has observed that the working set size of fine-grained threads tends to decrease with increasing parallelism, reducing cache interference [3].

There is a tradeoff between improving processor utilization and exacerbating cache interference as the number of contexts is increased. Limiting the number of contexts to improve cache performance is analogous to the problem of controlling the degree of multiprogramming to improve virtual memory performance. Starting with some of the literature on multiprogramming, thrashing, and working sets [8], we are currently investigating methods for adaptively limiting the number of resident contexts at runtime.

5.3 Multiple Active Contexts

We are also examining extensions to the basic register relocation hardware primitive. One powerful extension is to provide *multiple* register relocation masks that can be selected during instruction execution. As with the basic mechanism, this extension should only impact the instruction decode stage of the processor pipeline.

For example, the high-order bit of each register operand in a machine instruction could be used to select among two different `RRM`s. This would permit instructions to perform inter-context operations such as `ADD C0.R3, C0.R4, C1.R6`. The resulting instruction set would make an interesting compilation target for a concurrent intermediate language such as TAM [7], which attempts to minimize context switches by scheduling threads to share activation frames. This mechanism is also sufficiently powerful to emulate fixed-size, overlapping register windows.

Since an `RRM` requires only $\lceil \lg n \rceil$ bits for an architecture with n general registers, allowing multiple `RRM`s would require little additional hardware; the `LDRRM` instruction could simultaneously load several masks from a single general register. The most costly aspect of allowing multiple relocation masks is likely to be the need for multiplexers to permit each register operand to select the desired `RRM` for relocation.

6 Conclusions

We have presented a new mechanism that efficiently supports multiple variable-size processor contexts with minimal hardware support. Simple register relocation hardware, combined with appropriate software support, provides significant flexibility in the use of the register file to support multithreading. This flexibility enables better utilization of scarce register resources, allowing more contexts to remain resident

[8] The MIPS architecture has only 32 integer registers, and several are reserved for the operating system and standard calling conventions [13].

than is possible with conventional fixed-size hardware contexts.

A larger number of resident contexts makes it feasible to tolerate shorter run lengths and longer latencies, improving processor utilization over a wide range of system parameters. We have presented and analyzed a collection of experiments that demonstrates that register relocation can achieve substantial performance gains over fixed-size hardware contexts; a factor of two improvement is possible for many workloads. We are currently exploring the effects of multithreading on cache interference, and are examining extensions to the basic register relocation primitive.

Acknowledgements

We would like to thank Anant Agarwal, Eric Brewer, Bill Dally, Wilson Hsieh, Bruce Leban, Beng-Hong Lim, Peter Nuth, Greg Papadopoulos, and Paige Parsons for their comments and assistance. Thanks also to the anonymous reviewers for their many helpful suggestions.

References

[1] Advanced Micro Devices. *Am29000 User's Manual*, 1990.

[2] A. Agarwal, *et al.* "APRIL: A Processor Architecture for Multiprocessing", *Proc. 17th Annual Intl. Symp. on Comp. Arch.*, June 1990.

[3] A. Agarwal. "Performance Tradeoffs in Multithreaded Processors", *IEEE Trans. on Parallel and Distributed Systems*, September 1992.

[4] R. Alverson, *et al.* "The Tera Computer System", *Intl. Conf. on Supercomputing*, 1990.

[5] D. Bradlee, S. Eggers, R. Henry. "The Effect on RISC Performance of Register Set Size and Structure Versus Code Generation Strategy", *Proc. 18th Annual Intl. Symp. on Comp. Arch.*, 1991.

[6] E. Brewer, C. Dellarocas, A. Colbrook, and W. Weihl. "Proteus: A High-Performance Parallel Architecture Simulator", Tech. Rep. MIT/LCS/TR-516, MIT Lab. for Comp. Sci., September 1991.

[7] D. Culler, *et al.* "Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine", *Proc. 4th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[8] P. J. Denning. "Working Sets Past and Present", *IEEE Trans. on Software Engineering*, January 1980.

[9] R. Eickemeyer and J. Patel. "Performance Evaluation of Multiple Register Sets", *Proc. 14th Annual Intl. Symp. on Comp. Arch.*, 1987.

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

[11] J. Hennessy. "VLSI Processor Architecture", *IEEE Trans. on Computers*, December 1984.

[12] R. A. Iannucci. "Toward a Dataflow / von Neumann Hybrid Architecture", *Proc. 15th Annual Intl. Symp. on Comp. Arch.*, 1988.

[13] G. Kane. *Mips RISC Architecture*, Prentice-Hall, 1989.

[14] B. H. Lim and A. Agarwal. "Waiting Algorithms for Synchronization in Large-Scale Multiprocessors", MIT VLSI Memo #91-632, July 1991.

[15] R. S. Nikhil and Arvind. "Can Dataflow subsume Von Neumann Computing?", *Proc. 16th Annual Intl. Symp. on Comp. Arch.*, May 1989.

[16] P. Nuth and W. Dally. "A Mechanism for Efficient Context Switching", *Proc. IEEE Conf. on Computer Design*, October 1991.

[17] D. Patterson. "Reduced Instruction Set Computers", *Communications of the ACM*, January 1985.

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

[19] R. H. Saavedra-Barrera, D. E. Culler, and T. von Eicken. "Analysis of Multithreaded Architectures for Parallel Computing", *ACM Symp. on Parallel Algorithms and Architecture*, July 1990.

[20] V. Soundararajan. *Dribble-Back Registers: A Technique for Latency Tolerance in Multiprocessors*, Bachelor's thesis, MIT, 1992. Supervised by A. Agarwal.

[21] B. J. Smith. "A Pipelined, Shared Resource MIMD Computer", *Proc. Intl. Conf. on Parallel Processing*, 1978.

[22] M. R. Thistle and B. J. Smith. "A Processor Architecture for Horizon", *Proc. Supercomputing '88*, November, 1988.

[23] D. W. Wall. "Global Register Allocation at Link Time", *Proc. ACM SIGPLAN '86 Symp. on Compiler Construction*, 1986.

[24] C. A. Waldspurger and W. E. Weihl. "Register Relocation: Flexible Contexts for Multithreading", Tech. Rep., MIT Lab. for Comp. Sci. (to appear).

[25] W. D. Weber and A. Gupta. "Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results", *Proc. 16th Annual Intl. Symp. on Comp. Arch.*, June 1989.

A Context Allocation Code

```
/*
 * AllocMap is an allocation bitmap, represented by
 * a 32-bit integer. Each bit represents a 'chunk' of
 * 4 contiguous registers in a register file with 128
 * general registers. A set bit (1) denotes an unused
 * chunk; an unset bit (0) denotes a used chunk.
 *
 */
int AllocMap;

void ContextDealloc(Thread *t)
{
  /* Update bitmap to reclaim thread context. */
  AllocMap |= t->allocMask;
}

int ContextAlloc64(Thread *t)
{
  /*
   * Attempt to allocate a context for thread t
   * with 64 registers (16 'chunks'). Uses linear
   * search. Returns SUCCESS or FAILURE.
   *
   */
  int tempMap;

  /* check low-order halfword */
  tempMap = AllocMap & 0xffff;
  if (tempMap == 0xffff)
    {
      /* success: update bitmap, thread state */
      AllocMap &= ~tempMap;
      t->rrm = 0;
      t->allocMask = 0xffff;
      return(SUCCESS);
    }

  /* check high-order halfword */
  tempMap = AllocMap >> 16;
  if (tempMap == 0xffff)
    {
      /* success: update bitmap, thread state */
      AllocMap &= 0xffff;
      t->rrm = (16 << 2);
      t->allocMask = (0xffff << 16);
      return(SUCCESS);
    }

  /* fail: unable to alloc context */
  return(FAILURE);
}

int ContextAlloc16(Thread *t)
{
  /*
   * Attempt to allocate a context for thread t
   * with 16 registers (4 'chunks'). Uses binary
   * search. Returns SUCCESS or FAILURE.
   *
   */
  int rrm, tempMap;

  /*
   * Construct bitmap for blocks of chunks.
   * Use bit-parallel prefix scan. Combine to form
   * map of size-2 blocks, then map of size-4 blocks.
   * Then mask out irrelevant unaligned bits.
   *
   */

  tempMap  = AllocMap & (AllocMap >> 1);
  tempMap &= tempMap >> 2;
  tempMap &= 0x11111111;

  /* fail quickly if unable to alloc context */
  if (tempMap == 0)
    return(FAILURE);

  /*
   * Search bitmap for free block of chunks, setting
   * the rrm. Use binary search. First choose a 16-bit
   * block with an unused chunk, then an 8-bit block,
   * and finally a 4-bit block. A 'find first bit'
   * instruction could eliminate most of this code.
   *
   */

  rrm = 0;
  if ((tempMap & 0xffff) == 0)
    { rrm |= 16; tempMap >>= 16; }
  if ((tempMap & 0x00ff) == 0)
    { rrm |=  8; tempMap >>=  8; }
  if ((tempMap & 0x000f) == 0)
    { rrm |=  4; }

  /* success: update bitmap, thread state */
  tempMap = 0x000f << rrm;
  AllocMap &= ~tempMap;
  t->rrm = (rrm << 2);
  t->allocMask = tempMap;
  return(SUCCESS);
}
```