

On the Performance of A Multi-Threaded RISC Architecture

Scott K. Lindsay

Bruno R. Preiss

Department of Electrical and Computer Engineering

University of Waterloo

Waterloo, Ontario, N2L 3G1

Abstract

Multi-threading is a form of parallel processing in which the processor contains several independent contexts which share a single execution pipeline.

We propose a new multi-threaded architecture which differs from previous architectures in that context switches are performed only when the running program cannot execute an instruction in the next cycle. We argue that this strategy can improve pipeline utilization in environments which do not have a large enough number of processes to fully utilize earlier multi-threaded machines.

Introduction

Modern RISC processors are designed for high speed pipelined execution. Recent designs incorporate superscalar or superpipelined organization in order to increase execution speed by exploiting instruction parallelism, i.e., by attempting to execute more than one instruction in parallel. Unfortunately, the amount of available instruction parallelism in typical programs is low; on the order of two [1] to eight [2] instructions. In addition, both techniques tend to increase the penalty for branches and, hence, require expensive hardware for branch prediction. Ultimately, processor performance is limited by the presence of data and control hazards in programs as well as the need to fetch data and instructions from slow, off-chip memory.

[†]This work was supported in part by the Information Technology Research Centre (ITRC) of the Province of Ontario (Canada) and by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

An alternate means of exploiting instruction level parallelism is the technique called multi-threading. Multi-threading increases the amount of parallelism by interleaving the execution of unrelated programs at the micro-architecture level. The processor incorporates a hardware context for each program or thread. Each context includes all of the information required to execute the program including a complete register set, program counter, and status flags (if any). Instructions from more than one program may simultaneously occupy the execution pipeline. In typical multi-threaded machines, instructions are issued in round-robin fashion from each of the threads. If the processor contains as many contexts as pipeline stages, at most one instruction from each thread occupies the pipeline at any one time, thus eliminating all control and data hazards.

One of the first machines to incorporate multi-threading was the CDC 6600 Peripheral Processor [3]. In this machine, the primary rationale for multi-threading was the large difference between the processor and memory speeds. More recent commercial attempts include the Denelcor HEP [4] and the Tera Horizon [5]. Both of these machines use multi-threading as a means of hiding memory latency in a highly parallel machine intended to execute parallel algorithms. Some interesting research proposals incorporating multi-threading include [6] based on the MIPS-X architecture and [7] which investigated various instruction issuing policies.

Most multi-threaded machines built or proposed to date have been optimized for an environment in which there are many processes ready to execute at all times. This may be true in specialized application areas, but it does not reflect the typical situation for a general-purpose computer. In the case where there are fewer runnable threads than contexts, throughput of previously proposed

machines low. In addition to low throughput, the performance of individual threads is restricted to a fraction of the aggregate machine performance, even if there are idle resources.

In this project we attempt to address the above issues by proposing a new multi-threaded design in which context switches occur only when the running thread can no longer execute instructions, as opposed to on every cycle.

The remainder of the paper is organized as follows. First, we discuss the architecture of the proposed machine. We discuss the experimental method used in evaluating the architecture and present some of the simulated measurements next. Finally, we summarize the project and suggest some possible areas for future research.

Proposed Architecture

Our goal is to design a multi-threaded architecture which provides high throughput (i.e., few idle cycles), without sacrificing the performance seen by a single process. That is, if only one process is running on the machine, it obtains the same performance as if the machine were not multi-threaded.

We base our design on the MIPS R2000 architecture as described in [8]. The MIPS R2000 is a convenient starting point because it is a simple RISC architecture with few complications, as opposed to competing machines such as the SPARC. In addition, development and trace collection tools for the R2000 are readily available.

As previously stated, the objective is to increase pipeline utilization without sacrificing single process performance. Single process performance is maximized

by continuously executing instructions from a single thread. When the executing thread can no longer issue instructions because of a cache miss or data dependency, another thread must be chosen to execute. Ideally, this context switch should have no associated cycle cost. That is, an instruction from the new thread is executed in the cycle immediately following the last instruction executed by the previous thread. This design objective imposes constraints on each of the processor components. In the following sections, we examine the restrictions on each component, outline possible implementations, and indicate which implementation has been chosen for the proposed architecture. In general, we have attempted to choose implementations which provide good performance without making the design unreasonably complex. The result is a machine which could conceivably be implemented using current or near-term technology.

To achieve the goal of zero pipeline stalls, the instruction decode stage must issue an instruction to the execution stage during each cycle. Similarly, the instruction fetch stage must pass an instruction to the decode stage during each cycle. Because of cache misses and data dependencies, it is impossible for both of these conditions to be met continuously for a single thread, T_1 . To avoid idle cycles, an instruction from another thread, T_2 , must be issued by the decode stage immediately after the last instruction from T_1 . Since dependency detection is performed in the decode stage, the decision as to which thread to execute cannot be made until decoding has been completed. Thus, two or more threads must be decoded in parallel. The probability that at least one instruction can be issued during each cycle is maximized if one instruction from each thread is decoded during each cycle. This implies that the instruction fetch stage must also attempt to fetch an instruction for each of the threads during every cycle. Only threads which hit in the cache produce instructions to be passed on to the parallel decode stages.

In order to fetch one instruction for every thread on each cycle, the instruction cache and address translation hardware must each support several parallel accesses. The cost of providing parallel instruction fetch and decode can be somewhat reduced if we limit the number of available instructions to some number less than the number of contexts. If none of the fetch/decode units produces an executable instruction in a given cycle, the pipeline will be stalled.

The work performed in [9] suggests that the best performance is obtained when a single, shared cache is used for all of the threads. This requires that each cache line is tagged with the thread number to which it belongs. Similarly, the address translation hardware must also support up to N simultaneous accesses, where N is the number of executing threads.

An instruction is issued if it has been successfully fetched from the instruction cache and decoded, and it has no unresolvable dependencies upon any executing instruction.

Of the instructions which are eligible to execute, one must be chosen. Many policies for choosing an instruction are possible—we consider only two. The first

policy, which we shall refer to as running-first issuing, requires that the thread that issued an instruction during the last cycle be given the highest priority for issuing an instruction during the current cycle. The second policy, round-robin issuing, requires that the previously issuing thread be given the lowest priority for issuing during the current cycle. Issuing using the running-first policy produces behaviour similar to that seen on a single threaded machine, in that a thread continuously executes instructions until it is forced to stall. Using the round-robin policy tends to divide available cycles evenly between the threads as in previously proposed multi-threaded machines.

The MIPS architecture includes a delayed branch with a single delay slot. Providing delay slots for the multi-threaded version is more complex because of the possibility of instruction cache misses. On the MIPS, if the instruction in the delay slot misses, the pipeline stalls. In the multi-threaded version, we avoid stalling the pipeline by switching to a new thread. If execution proceeds as in the R2000, the branch instruction finishes its execution and writes the new program counter address. The program counter write may occur before the delay slot instruction has been successfully fetched, in which case the delay slot instruction is never executed. To avoid this situation, we must save both the delay slot instruction address and the branch target address when a context switch occurs before a delay slot instruction can be issued. It is then possible to ensure that the delay slot instruction is fetched and executed before the instruction at the branch target address.

As with branches, the MIPS architecture includes a single delay slot following load instructions. The multi-threaded version cannot simply context switch on a data cache miss (after the delay slot instruction) because the cache result is not known until after the second instruction following the load has already been issued.

If we assume that cache hits are the most likely case, a reasonable strategy is to issue the instruction following the delay slot and then if the data access misses we annul the optimistically issued instruction and context switch. This solution introduces a single pipeline bubble in the case of a cache miss, but this is likely to degrade performance less than the simpler strategy of always context switching after the delay slot instruction before the cache access outcome is known. Other solutions are possible, but they involve additional complexity or incompatible changes to the MIPS instruction set.

The proposed machine has a two-level cache hierarchy coupled to a main memory which supports burst transactions for transferring cache lines to and from memory. The first level of cache consists of on-chip instruction and data caches consisting of 2,048 direct-mapped lines of eight bytes each. The second level is an off-chip secondary cache shared between instructions and data. The secondary cache consists of 32,768 lines of 32 bytes each, organized as a non-blocking, direct-mapped cache. Each cache line (in each of the caches) contains a thread identifier tag which is used as part of the tag in identifying cache hits. All of the caches are virtually addressed, primarily because of the difficulty of

implementing realistic address translation in the simulator.

Experimental Method

The performance of the proposed architecture was evaluated using a trace driven simulation. Traces were collected using *pixie*¹. Detailed descriptions of *pixie*'s operation and file formats can be found in [10]. A custom simulator was created which analyzed the executable program and trace file to predict machine performance. Input to *pixie* and the simulator consisted of three benchmark programs chosen from the SPEC benchmark suite [11]: *gcc*, *spice* and *vertex*.

The benchmark programs were used to create a total of eight separate traces using different input files. Simulations were performed with all possible combinations from the eight traces for each size of machine (i.e., number of threads) and the results were averaged to produce the measurements described below. For each of the experiments performed, we use pipeline utilization as the primary measure of performance. Additional measurements are presented as required to explain the pipeline utilization results.

Experimental Results

Our experiments were designed to investigate several questions: (1) the performance of the multi-threaded architecture as compared to a single threaded machine; (2) the number of threads required to usefully increase performance; and, (3) the impact of the chosen instruction issuing policy.

Figure 1 shows the mean pipeline utilization in the base machine and 95% confidence intervals (calculated using the Student's T distribution).

The measured pipeline utilization is encouraging, in that we obtain over two thirds of the maximum potential benefit in multi-threading (obtained with four threads) in going from one to two threads and ninety-five percent of the maximum benefit in going from one to three threads. This result is a substantial improvement over previously proposed multi-threaded architectures which require higher numbers of runnable threads to achieve high throughput (e.g., 4 threads in [9] to achieve 80% ideal utilization) The proposed architecture requires only two active threads to achieve 82% utilization, including the cost of cache misses.

The base machine uses cache sizes of 16 kilobytes each for the instruction and data caches and 1 megabyte for the off-chip secondary cache. Primary instruction and data cache hit ratios are shown in Figure 2 The results show the aggregate cache hit ratio measured for each cache. Instruction cache hit rates follow the same general pattern as the thread utilization, suggesting that instruction cache behaviour may be a key factor in determining the performance of a multi-threaded machine.

¹*pixie* is a trademark of MIPS Computer Systems

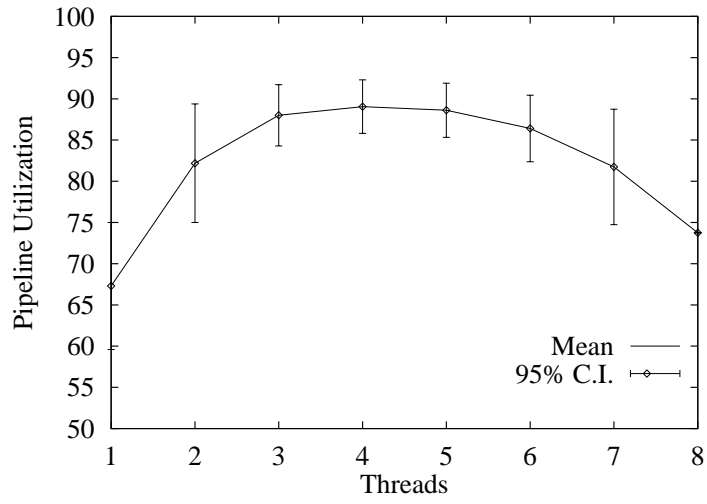


Figure 1: Base Machine Pipeline Utilization

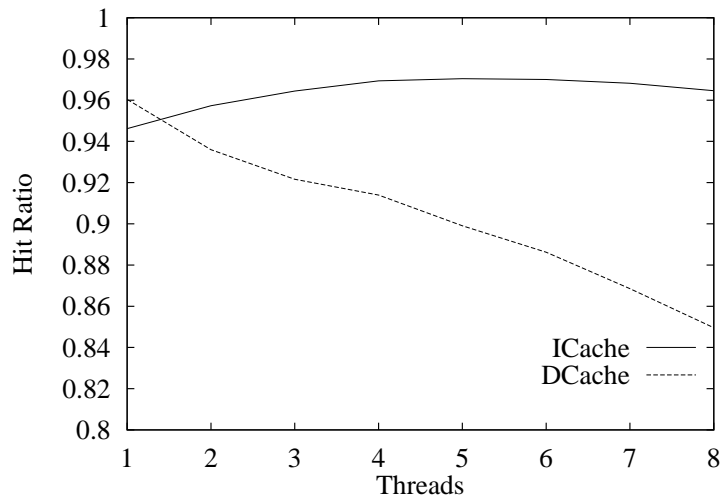


Figure 2: Data & Instruction Cache Hit Ratios

The fact that instruction cache hit ratio increases as the number of threads increases is not intuitively obvious. The reason is that the cache footprint of the benchmark programs (i.e., cache lines occupied by a program) do not entirely fill the cache. For the single threaded machine, a single program runs for the entire simulation. During the program’s execution, the working set gradually changes, but the active cache lines never entirely fill the cache. Changes in the working set translate into cache misses. For multi-threaded machines, a number of threads share the cache at any one time. Thus, the cache is more likely to be filled with active cache lines at any given time. Any given thread will experience more cache misses because its cache lines may be displaced by another thread. The aggregate cache hit ratio increases however, because the number of unused cache lines decreases. The net effect is to produce an aggregate footprint which more nearly fills the cache, and thus translates into a higher hit ratio. This effect does not continue indefinitely as we add more threads. Eventually, we reach a point where the conflict rate overwhelms any benefit from the increased cache footprint. This is likely to occur at the point when the cache is effectively full of active cache lines and each miss must displace an active line. At this point, cache hit ratios begin to decrease.

Conversely, the data cache hit rate consistently declines as the number of threads is increased. Although not shown, the secondary cache shows similar behaviour although the magnitude of the effect is larger. Decreasing data cache hit rates may be attributable to poor data locality for the benchmark programs. It is also possible that this behaviour is a simulation artifact introduced by using virtual instead of physical addressing for the caches. The addressing method is important because the linker assigned data section for all of the benchmarks begins at the same virtual address. The code sections, in contrast, start at different virtual addresses in each of the benchmarks. This more nearly represents the expected behaviour in a real system.

Figure 3 shows the effect of changing the instruction issue policy. The running-first policy attempts to continue issuing instructions from the same thread as long as possible. The round-robin policy attempts to divide execution cycles fairly between all threads which have runnable instructions, so that a single thread does not monopolize execution resources. Round-robin issuing is similar to the issue policy of previous multi-threaded machines.

The running-first instruction issue policy consistently produces higher pipeline utilization than round-robin issuing. This data indicates that the issuing policy has a significant effect on the potential performance of the machine—approximately half of the average benefit of multi-threading. In fact, a two way multi-threaded machine using round-robin issuing performs worse than the single-threaded machine. Our measurements indicate that an important reason for this effect is competition for cache lines between the threads. If a single thread is allowed to run continuously, it tends to accumulate cache lines and is, thus, less likely to experience cache misses. If execution time is divided equally between threads, no single thread acquires a large fraction of the cache lines and the aggregate

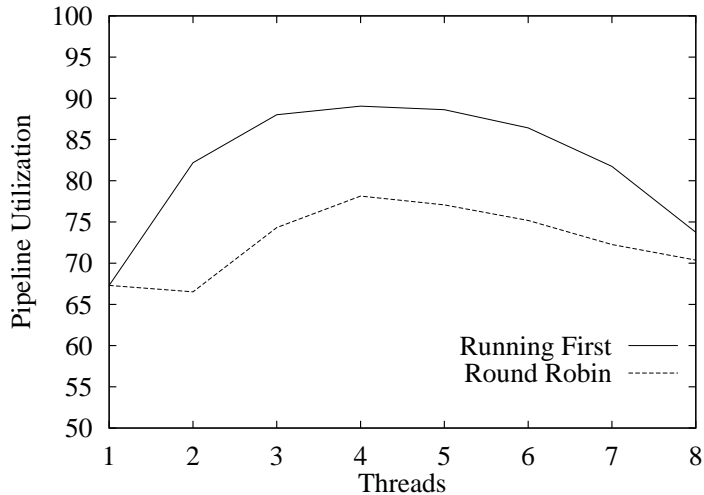


Figure 3: Instruction Issue Policy

miss rate increases.

Figure 4 shows the effect on pipeline utilization of varying the data load policy. The proposed architecture uses a switch-on-miss policy, in which a context switch occurs only when a data cache miss occurs. A simpler and cheaper strategy is to simply stall the pipeline until the cache access has been resolved without performing a context switch. Stall-on-miss is equivalent to the policy implemented by a single-threaded architecture.

The performance penalty for stalling ranges from 14.9% to 62.6%. Pipeline utilization is worse than the single threaded case for all but two threads which achieves a slight increase in performance as compared to the single-threaded machine. Thus, implementing an intelligent load policy is crucial to the success of a multi-threaded machine.

Conclusions

The work performed for this study has attempted to address the problems of previous multi-threaded architectures. Our simulations show that pipeline utilization for the multi-threaded architecture is increased by an average of 25% as compared to a single-threaded machine. The maximum improvement of pipeline utilization was achieved using four threads, but good improvements were also observed with two and three threads. This indicates that a multi-threaded

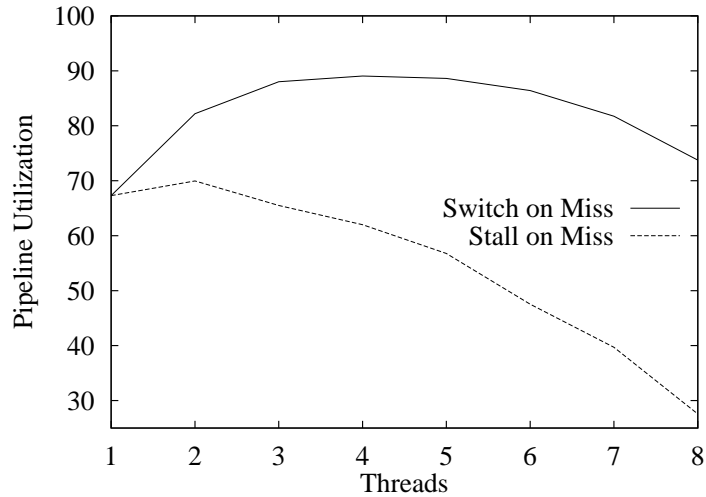


Figure 4: Load Policy

machine with only a small number of threads can still achieve acceptable performance, in contrast to previously proposed multi-threaded machines. These results were achieved using a relatively conservative memory design (i.e., using commercially available memory speeds to design a system that could easily be built).

Our experiments also yielded some insight into factors which affect the design criteria. Maximum performance of the multi-threaded architecture seems to depend on minimizing the number of context switches performed, i.e., by letting a single thread execute for as long as possible. The primary cause of this behaviour is the competition for cache lines between the threads. If a single thread is allowed to execute continuously, it accumulates more cache lines containing its working set and is, therefore, less likely to suffer a cache miss. If threads are forced to context switch too frequently, the cache lines are more evenly distributed between the threads and, thus, cache misses are more likely. Since a pipeline stall (of one cycle) occurs whenever a data cache miss occurs, more data cache misses translates into lower pipeline utilization. Increasing the number of instruction cache misses increases the probability that no thread will have an issuable instruction on a given cycle, thus also lowering the pipeline utilization. If generally true, this result implies that context switches should be reduced as much as possible in all parts of the design. Thus, instruction and data cache hit ratios should be maximized and the switch-on-load policy for data misses should be avoided. In combination with the data policy exper-

iment, this implies that some form of switch-on-miss or switch-on-use strategy is required for handling data cache misses.

While our study has demonstrated the potential performance benefits of multi-threading when designed to maximize throughput, it has also revealed some areas which warrant future investigation. A more detailed design should be evaluated to determine the impact of multi-threading on clock cycle time and chip area and, consequently, on performance. The performance of the machine using a more realistic memory system should be investigated. In particular, the effect of mapping virtual to physical addresses must be addressed.

References

- [1] Michael D. Smith, Mike Johnson, and Mark A. Horowitz. Limits on multiple instruction issue. In *ASPLOS 3*, 1989.
- [2] David W. Wall. Limits of instruction level parallelism. In *ASPLOS 4*, 1991.
- [3] J. E. Thornton. *Design of a Computer: The CDC 6600*. Scott, Foresman and Company, 1970.
- [4] Janusz S. Kowalik, editor. *Parallel MIMD Computation: HEP Supercomputer and its Applications*. The MIT Press, 1985.
- [5] Mark R. Thistle and Burton J. Smith. A processor architecture for Horizon. In *Proceedings Supercomputing 88*, 1988.
- [6] Ian MacIntyre and Bruno R. Preiss. The effect of cache on the performance of a multi-threaded pipelined RISC processor. In *The Canadian Conference on Electrical and Computer Engineering*, 1991.
- [7] Daniel C. McCrackin and Barna Szabados. Horizontal demand prefetching: A novel approach to eliminating the jump problem. *Canadian Journal of Electrical and Computer Engineering*, 16(3), 1991.
- [8] Gerry Kane. *MIPS R2000 RISC Architecture*. Prentice Hall, 1987.
- [9] Ian Donald MacIntyre. A study of horizontal pipelining in a RISC processor. Master's thesis, University of Waterloo, 1989.
- [10] Michael D. Smith. *Tracing with pixie*. Center for Integrated Systems, Stanford University, 1991.
- [11] SPEC. SPEC benchmark suite release 1.0, 1989.