The Effect of Cache on the Performance of a Multi-Threaded Pipelined RISC Processor

Ian MacIntyre
Bruno R. Preiss

Department of Electrical and Computer Engineering
University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1

Abstract

This paper examines the effects of multithreaded pipelining on the CPI (cycles per instruction) of a RISC processor. The desired CPI in a conventional (single-threaded) RISC processor is one instruction per cycle. However, the CPI is typically more than one because of data hazards, control hazards, and resource hazards in the pipeline.

A multi-threaded processor performs a context switch between every instruction. Multi-threaded pipelining holds out the promise of achieving a lower CPI because it can eliminate data and control hazards, and mask the effects of memory latency. However, multi-threaded pipelining reduces cache hit ratios and requires more chip area to implement.

In this paper, we present a model for predicting the CPI of a multi-threaded pipelined processor. We also present the results of trace-driven simulations of single- and multi-threaded processors. These data show that, for reasonable implementation technologies, and taking into account the chip area penalty, a multi-threaded processor can achieve a lower CPI than a single-threaded processor.

1. Introduction

Ideally, a single-threaded processor could issue a new instruction in each clock cycle. However, pipelined processors cannot sustain the optimum CPI (cycles per instruction) because of interactions between instructions in the pipeline (hazards). There are three types of hazards to consider in the design of a RISC processor — data hazards, control hazards, and resource hazards. A data hazard arises between a pair of instructions in the pipeline when a later instruction needs to use the data produced by an earlier one. A control hazard arises when executing an instruction that changes or may change the instruction execution sequence (typically a branch or jump instruction in a RISC). The principal resource hazard in a RISC processor is due to external memory latency.

1.1. Multi-threaded Pipelining

Multi-threaded pipelining has been proposed as a technique to improve the CPI of a RISC processor. The basic idea is to eliminate, as much as possible, the interactions between consecutive instructions in the execution pipeline. One way to do this is to execute consecutive instructions from independent execution threads.

In typical multi-user, multi-tasking systems, the workload consists of many independent processes. Each process has a virtual computer with a CPI that is a multiple of the CPI of the physical processor. The virtual processors are implemented using time-slicing with time-slice intervals typically about 1 ms to 100 ms[1].

In a multi-threaded processor, the time-slice interval is reduced to one cycle. Thus, if the number of processes running is greater than or equal to the number of pipeline stages, each instruction in the pipeline belongs to a different process. A single pipeline supports several virtual processors. Because the threads running on the virtual processors are independent, there can be no dependencies between instructions in the pipeline. Consequently, a multi-threaded processor has the potential to achieve a lower CPI because it is less sensitive to data, control, and resource hazards.

In this paper we assume that the multi-threaded processor supports exactly as many threads as there are pipeline stages and that the threads are serviced in a round-robin fashion. This proposal is a trade-off between two factors — chip resources needed to support multiple, virtual processors and the need to eliminate pipeline hazards. Each virtual processor requires chip resources for the storage of its state. Consequently, the number of concurrently executing threads should be as small as possible. On the other hand, to eliminate hazards entirely, there should not be more than one instruction from a given thread in the execution pipeline at any point in time. Of course, more threads can be supported on such a machine using the usual time-slicing mechanism. Note that if the number of threads is less than the number of virtual processors, the performance of the system will be severely degraded with respect to a time-sliced, single-threaded processor.

In this paper, we make the assumption that there are always enough threads available.

1.2. Origins of Multi-threaded Pipelining

The use of multi-threaded pipelining to provide several virtual processors (virtual multiprocessing[2]) is not a new concept. Multi-threaded pipelining was used in the Peripheral and Control Processors of the Control Data 6600 computer architecture of the early 1960s to provide several virtual peripheral processors[3]. More recently, the Denelcor HEP computer[4], and some proposed architectures including the Multithreaded Processor Architecture for Parallel Symbolic Computing[5], the Circulating Context RISC Multiprocessor[6], and the Cyclic Pipeline Computer[7] have all used multi-threaded pipelining to provide virtual multiprocessing.

1.3. Potential Advantages of Multi-threaded Pipelining

Data Hazards: Because each instruction in the pipeline of a multi-threaded processor belongs to a different process, there can be no data dependencies between instructions in the pipeline. Consecutive instructions from a given process will never be in the pipeline at the same time. As a result, data hazards do not arise; code reorganization is not required; and, NOPs need never be inserted into the instruction stream to prevent data hazards.

Control Hazards: As explained above, consecutive instructions from a given process will never occur simultaneously in the pipeline of a multi-threaded processor. Consequently, branch target addresses can be determined before the next instruction of a given process needs to be issued. Thus, control hazards do not arise; branch delay slots are not required; instruction squashing is not required; and, NOPs need never be inserted to prevent control hazards.
Memory Latency Effects: In a single-threaded processor, the execution pipeline must stall when an external cache miss occurs. The effect is to delay the offending instruction and any other instructions currently in the pipeline. We propose the following approach in a multi-threaded processor: When an instruction causes a miss, that instruction is squashed. An instruction squashed in this way will be retried the next time it circulates through the pipeline. However, the following instructions in the pipeline are not squashed nor are they delayed. This means that the following instructions can continue to execute. The effect of this approach is to change the sequence in which instructions are executed. However, since consecutive instructions in the pipeline belong to different threads, this change does not affect the determinacy of any given process.

If the cache can process the miss in parallel with the following cache hits, the missing data will eventually be loaded into the cache. (A mechanism for doing this is described in section 3.2 below.) If the memory latency is less than the number of stages in the pipeline, then the next time the offending instruction is issued, the cache access will succeed.

The important observation about memory latency in an multi-threaded pipelined RISC processor is that it only affects a single instruction in the pipeline rather than the offending instruction and all others following it. Thus, the effect of the memory resource hazard on the CPI of the pipeline is reduced (but not eliminated) in the multi-threaded processor.

1.4. Potential Disadvantages of Multi-threaded Pipelining

Chip Area: A multi-threaded processor with \( n \) pipeline stages must provide storage for the state of \( n \) virtual processors. I.e., the processor must have \( n \) register sets, \( n \) program counters, and \( n \) copies of any other virtual processor state information. (Only one execution pipeline is required.) Consequently, a multi-threaded processor requires substantially more chip area for the datapath than a single-threaded processor. However, in certain cases some chip area can be recovered. For example, because data and control hazards have been eliminated, no hardware is required to detect such hazards and operand forwarding is not required.

External Hardware Complexity: Peripherals and coprocessors used with a multi-threaded processor are expected to be somewhat more complicated than in single-threaded processors. For example, memory management units and floating point coprocessors must be able to support consecutive operations from different threads. One possible solution to this is to replicate such coprocessors — one for each thread in the pipeline. These issues are discussed in more detail in [8]. The focus of this paper is on the performance of the processor itself.

Cache Performance: A major problem with multi-threaded pipelining is that doing a context switch (time-slice) on every cycle destroys both the spatial and temporal locality of the memory accesses. Since consecutive instructions belong to independent threads, consecutive memory accesses are entirely independent. (This is true for both the instruction and the data streams.) Consecutive memory accesses are made to different virtual address spaces. Consequently, a given size of cache will exhibit lower cache hit ratios when used in a multi-threaded processor as compared with a single-threaded processor.

As a lower bound, the hit ratio in a multi-threaded processor with \( n \) pipeline stages will be the same as the hit ratio of a single-threaded processor with a cache that is \( 1/n \) times as large. To see why this is the case, suppose that the hit ratio observed by the \( n \) processes operating out of a single cache is lower than that observed by a single process operating out of a cache \( 1/n \) times as large. Then, clearly, the correct implementation for the multithreaded machine would be to use \( n \) separate caches and thus achieve the lower bound performance! A question examined in this paper is the degree to which the cache hit ratios degrade because of multi-threaded pipelining.

2. Analytic Performance Models

In this section we present analytic performance models for a single-threaded processor and a multi-threaded processor. The model predicts the CPI of a processor given cache hit ratio and instruction mix data.

We have chosen the Stanford MIPS-X processor as the basis for the comparison[9]. The MIPS-X processor was chosen because: its execution pipeline and instruction set are designed so that all instructions will execute in five cycles given cache hits in both instruction and external caches; it has a single register set (c.f., multiple register sets, as in [10]) which minimizes the processor resources that have to be replicated in the multi-threaded version; and, information about program characteristics is readily available. In part, the Stanford MIPS-X was also chosen because of the availability of a compiler and an instruction-level simulator for this processor[11]. Note, we are not proposing that the ideal way to design a multi-threaded pipelined RISC processor is to start with an existing single-threaded processor. We have taken this approach for the purpose of fair performance comparison.

A salient feature of the Stanford MIPS-X is that it contains an on-chip instruction cache. In addition, it is assumed that the processor will be used with an external cache (or caches) for both instructions and data.

2.1. Single-Threaded Processor Model

For the purposes of the analytical model, we consider only the memory accessing instructions and the instructions that do ALU operations (branches and jumps are grouped with ALU operations). The other instructions, for example those associated with trap processing, are ignored in this analysis. Since these other instructions are used infrequently, the effect of omitting them from the analysis is small.

The MIPS-X has a five stage instruction execution pipeline. In the first stage, instructions are fetched from the on-chip instruction cache. If an instruction cache hit occurs on an ALU instruction, then that instruction will complete its execution in five cycles. If an instruction cache miss occurs, then an external cache access is made. This access will incur a two cycle penalty. If an external cache miss occurs, then a main memory access is required. The main memory access will incur an additional penalty due to the main memory latency.

In addition to the cache accesses needed for the instruction fetch, memory accessing instructions (loads and stores) access the external cache for data. If a miss occurs in the external cache, then a main memory access must be done.

The following data are needed to estimate the CPI of the MIPS-X processor: instruction cache hit ratio; instruction cache miss penalty; external cache hit ratio; external cache miss penalty; and, the distribution of the two types of instruction.

Ideally, the MIPS-X processor would achieve a CPI of one. However, the peak execution rate can only be sustained if there are no cache misses, that is, cache hit ratios of one in both the instruction and external caches. When a miss occurs, a penalty (measured in cycles lost) is incurred, thereby increasing the CPI.

An ALU instruction can execute in three different ways depending on whether instruction and external cache hits occur. An ALU instruction can (1) hit in the instruction cache on the instruction fetch; (2) miss in the instruction cache on the instruction fetch but hit in the external cache; or, (3) miss in the both instruction cache and the external cache on the instruction fetch. (In addition, a fourth execution path is possible if the external cache uses a copy-back main memory update strategy. Here there may be some additional penalty incurred if the instruction fetch causes a copy-back operation because the target cache line is dirty. A miss on the instruction
A memory access instruction (load or store) can execute in one of six different ways. There are three possible executions of the instruction fetch (as above) followed by either an external cache hit or miss on the data access. (In addition, if a copy-back main memory update strategy is used, an external cache miss may incur an additional penalty because of the copy-back operation. If this is the case, the number of different executions for a memory access instruction doubles to 12.)

Let the probability of executing a memory access instruction be $P_m$. Thus, $(1 - P_m)$ is the probability of executing an ALU instruction. Let the instruction cache hit ratio be $h_i$ and the external cache hit ratio be $h_E$. Let $P_d$ be the probability of an external cache miss requiring a copy-back operation. Let the miss penalty associated with an instruction cache miss be $MP_I$, and let the miss penalty associated with an external cache miss be $MP_E$. As a worst-case bound, assume that the penalty associated with a copy-back is equal to $MP_E$. Thus, external cache accesses requiring copy-back take twice as long as those that do not. (Of course, the penalty in an implementation will be less if write buffers are used.) Then, the CPI is given by:

$$CPI_{raw} = 1 + MP_I(1 - h_i) + MP_E(1 - h_E)(1 + P_d)(1 + P_m). \quad (1)$$

Using appropriate values for $P_m$, $P_d$, $h_i$, $h_E$, $MP_I$, and $MP_E$, it is possible to compute $CPI_{raw}$ for the processor. The raw CPI includes the execution time for NOP instructions that have been inserted to avoid pipeline hazards as well as the instructions in the branch delay slots that have been squashed.

To compute the CPI for useful instructions, the NOPs and squashed instructions must be removed from the raw instruction throughput. The effective CPI, $CPI_{eff}$, can be computed by dividing $CPI_{raw}$ by the fraction of instructions executed that are not NOPs. I.e., if $f_{NOP}$ represents the fraction of instructions executed that are NOPs, then $CPI_{eff} = CPI_{raw}(1 - f_{NOP})$.

The computed CPI data are shown in Figures 1 and 2 for various instruction and external cache hit rates ($h_i$ and $h_E$). In these curves, the instruction cache miss penalty, $MP_I$, is two cycles (i.e., the same as for the Stanford MIPS-X). The external cache miss penalty, $MP_E$, is set to four cycles. (E.g., this corresponds to a main memory access time of 200 ns assuming a 20 MHz clock.) In addition, for all curves but one, $P_d$ (the probability of a miss on a dirty line when using a copy-back cache) was set to zero. In one case $P_d$ is set to one. This, together with the worst-case copy-back miss penalty discussed above, places an upper bound on the CPI.

The frequency of NOP instructions and the distribution of memory referencing vs. ALU instructions can be taken from studies of MIPS-X execution characteristics. In one such study[12], $P_m$ was calculated to be 0.3 and $f_{NOP}$ was found to be 0.156.

### 2.2. Multi-threaded Pipelined Processor Model

Equation (1) also applies to a multi-threaded version of the Stanford MIPS-X processor. The differences between the two architectures are captured entirely in the parameters $MP_I$, $MP_E$, $f_{NOP}$, and $P_m$. By assigning appropriate values to these parameters, the performance of the multi-threaded processor can be predicted.

**Figure 1. Instruction Throughput ($R$) vs. Instruction Cache Hit Ratio ($h_i$)**

In the multi-threaded version of the MIPS-X processor, the instruction cache miss penalty, $MP_I$, is only one cycle. When the miss is encountered, a single bubble replaces the instruction that was not fetched. The instruction can be fetched from the external cache within the next five cycles while the other streams continue to execute. (This assumes that no external cache miss occurs.)

The external cache miss penalty, $MP_E$, for the multi-threaded processor is $[T_{access}/5]$ cycles, where $T_{access}$ is the number of cycles it takes to access main memory. (I.e., $T_{access}$ is $MP_E$ for the single-threaded processor.) This is because while an external cache miss is being serviced, the thread that is waiting for the memory operation to be completed will only access the external cache once every five cycles. Pipeline bubbles will only be introduced during these cycles. This illustrates the effect that multi-threaded pipelining has on reducing the performance penalty associated with memory latency.

The principal advantage of multi-threaded pipelining is the elimination of the need for NOP instructions to prevent data and control hazards. Consequently, such NOP instructions are not required in the instruction sequence of a multi-threaded processor. Thus, $f_{NOP}$ is zero for the multi-threaded processor. Since NOP instructions do not reference memory and are eliminated from the multi-threaded code, the proportion of memory referencing instructions is higher than in the code for the single-threaded processor. The value of $P_m$ is adjusted by dividing by $(1 - f_{NOP})$. The adjusted value for $P_m$ can be calculated from the Stanford MIPS-X data to be 0.355 for the same code sequences.
1.00
1.50
2.50

performance of the cache turn out to be worse than this, then it should be

more, suppose the ratios are reduced to 0.67 and 0.98, respectively, when

the cache sizes are divided by five. Then it is possible to read the CPI

multi-threaded processor will be the same as those of the single-threaded

processor data can be obtained either by simulation or from an analytic model

of cache behaviour[13]. However, models for the effect multi-threaded

cache hit ratios for both the single-threaded processor and external

cache hit ratios. Note that the benefit of multi-threaded pipelining is small-

est when perfect (100%) cache hit ratios are used. As the hit ratios
decrease, the difference between the two processors increases.

2.3. Using the Analytic Models to Compute Performance Bounds

To use the analytic models to predict performance, the instruction

cache and external cache hit ratios for both the single-threaded processor

and the multi-threaded processor must be known. The single-threaded pro-
cessor data can be obtained either by simulation or from an analytic model

of cache behaviour[13]. However, models for the effect multi-threaded

pipelining on cache hit ratios do not yet exist.

It is possible to compute bounds on the cache hit ratios of the

multi-threaded processor as follows. At best, the cache hit ratios for the

multi-threaded processor will be the same as those of the single-threaded

processor. At worst, the cache behaves as if it is partitioned into five

separate caches each one fifth the size of the original cache. (Should the

performance of the cache turn out to be worse than this, then it should be

replaced by five independent caches, each one fifth the size of the original

cache.)

For example, suppose the instruction cache and external cache hit ratios for a single-threaded processor are 0.74 and 0.98, respectively. Fur-

thermore, suppose the ratios are reduced to 0.67 and 0.98, respectively, when

the cache sizes are divided by five. Then it is possible to read the CPI
directly from Figures 1 and 2. In this case, the single-threaded processor
would require 1.85 cycles per instruction while the CPI of the multi-
threaded pipelined processor is bounded by 1.28 and 1.33 instructions per
cycle.

3. Implementing a Multi-threaded Pipelined Processor —
Architectural Issues

In this section we discuss some of the changes in the architecture
of the processor needed to support multi-threaded pipelining. These
changes are presented in the context of the Stanford MIPS-X processor as it
is the basis for the analytic and simulation comparisons presented in this
paper. Of course, if a multi-threaded processor was to be designed from
the ground up, it might not look like the processor we describe below. The
MIPS-X is used as a sample RISC processor in this study to illustrate the
trade-offs involved in the implementation of multi-threaded pipelining.

3.1. Processor Resources for Multi-threaded Pipelining

To support multi-threaded pipelining, the processor must provide a
program counter and a complete set of registers (32 for the Stanford MIPS-
X) for each thread. (The active registers and program counter can simply be
selected using a modulo five counter.) In addition, some mechanism is
needed to isolate the address spaces of the various threads. A simple
method, used on the MIPS processor[14], is to insert the process ID into the
upper bits of the virtual address. This mechanism can be easily applied to
the multi-threaded processor by providing a register in each pipeline stage
to hold the thread ID.

The largest change in chip area needed is a result of the need to
provide a register set for each virtual processor. A total of 128 additional
general-purpose, 32-bit registers have to be added to the MIPS-X to provide
the resources of one MIPS-X processor for each of five streams. In the
design of a multi-threaded processor from scratch, one may elect to reduce
the number of registers allocated to each stream in order to reduce the chip
area needed. However, in this study we assume that the full resources of a
MIPS-X processor are made available to each stream. The increase in chip
area for multiple program counters is not significant since the MIPS-X pro-
cessor uses an array of program counters to store the addresses of instruc-
tions executing in the pipeline. This is done in the MIPS-X to make inter-
rupts precise and to aid in restarting the pipeline after servicing an inter-
rupt[15].

3.2. Execution Pipeline

Since multi-threaded pipelining removes hazards in the pipeline, it
eliminates the need for detecting those hazards in the hardware. In the
MIPS-X, forwarding registers are used to bypass the register store (i.e., to
prevent read-after-write hazards). These registers, plus the hardware to con-
trol them, can be removed from the multi-threaded processor.

Two enhancements to the operation of the pipeline are needed to
support multi-threaded pipelining. First, it must be possible to abort
instructions on a cache miss, so that the processor does not have to be
delayed while waiting for the cache update. In the MIPS-X, this can be
done by simply not storing the result from an instruction to a register and by
not modifying the program counter until it is certain that that instruction
will complete. Since the program counter has not changed, the instruction
will automatically be reexecuted five cycles later.

The second problem deals with instruction cache misses. The
basic problem is this: how can the internal instruction cache be updated
without increasing the bandwidth of the interface between the processor and
the external cache? The solution we propose uses the pipeline bubble
created by the instruction miss. Every instruction has a one cycle window
for a main memory access should the instruction turn out to be a load or a
store. When an instruction cache miss occurs, instead of replacing the
instruction by a NOP, we replace it by an instruction that loads the internal
cache from the external cache during the cycle dedicated for the memory
access. Thus, the bandwidth of the processor to external cache interface
does not need to be increased to support the multi-threaded processor.
3.3. A Pitfall of Multi-threaded Pipelining

Careful consideration must be given to when lines are loaded into the caches (both instruction and external), and to when these lines can be removed by misses. Problems arise in a multi-threaded processor that do not exist in a single-threaded processor since several, independent streams are always concurrently executing. In a cache with small associativity (or in a direct-mapped cache) it is possible for several streams to access data that occupy the same line in the cache. If the associativity of the cache is less than the number of streams, then deadlock may occur if the cache accesses are not properly synchronized. E.g., one stream may miss on a fetch forcing the removal of a line from the cache that was brought in for another stream but has not yet been accessed by the other stream. Since the other stream will reexecute the instruction that brought in the missing line, that instruction will miss again. A cache line locking mechanism could be used to prevent this type of deadlock. (E.g., a line could be locked in the cache until it is accessed by a stream.) However, care must be taken in how locking is implemented. A poor locking strategy might result in cache lines being locked permanently, or for an extended period of time, because of the occurrence of an interrupt, a page fault, or the interaction between a cache miss on the instruction fetch and a cache miss on the memory access operation of a load or store instruction.

A simple solution that we have used in our study is an external cache line locking strategy in which a line is locked on a miss until either the line is successfully accessed or a time-out occurs. The time-out mechanism is needed to unlock lines should an interrupt occur that prevents the instruction from automatically unlocking the line or should the instruction that locked the line be itself removed from the instruction cache.

If the instruction cache update strategy described above in section 3.2 is used, then no line locking is necessary in the instruction cache. This is because the multi-threaded MIPS-X the instruction would be loaded into the instruction cache during the final stage of the pipeline. I.e., the instruction is loaded into the cache during the cycle immediately before the instruction fetch that accesses that instruction.

3.4. The Main Memory Interface

In the multi-threaded processor, there is a small, but finite, possibility that several consecutive instructions all miss on the external cache access (possibly after having missed in the instruction cache). Ideally, the main memory would support five simultaneous (pipelined) accesses (e.g., using interleaved memory banks). However, the cache hit ratios are expected to be high (>90%). Consequently, the probability of consecutive external cache misses would be low. So, we propose the use of a typical main memory interface, i.e., one that supports one outstanding memory reference at a time. In those cases where consecutive accesses miss in the external cache, the second instruction would be squashed and would retry its access five cycles later.

4. Simulations

In the analysis presented above, it was not possible to compare the single-threaded processor with the multi-threaded processor because the effect of multi-threaded pipelining on cache hit ratios could not be predicted. We have used trace driven simulation to measure the effects of multi-threaded pipelining on cache hit ratios. In this section we describe the simulations and in the next section we present the results of these simulations.

In this study, five Pascal programs were acquired (see Table I) and code was generated for both the Stanford MIPS-X processor and for a multi-threaded version as described above. The workload was intended to represent a “typical” Unix workload.

The single-threaded processor executed the programs in a round-robin order with a context switch interval of 300,000 cycles. (This corresponds to a time-slice interval of 1/60 s with a 20 MHz clock.) Of course, the multi-threaded processor does a context switch on every cycle.

No attempt was made to simulate operating system calls, interrupts, or code that would be executed to do a context switch. Long simulations were done (trace lengths of at least 10 million cycles) to minimize the effects of the cache and simulation start-up transients. A total of 180 simulations were done to gather results for this study, each simulation required between 3 and 24 hours on two MicroVAX II computers to complete.

Table I

<table>
<thead>
<tr>
<th>Trace Program</th>
<th>Single-Threaded Processor</th>
<th>Multi-Threaded Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compress</td>
<td>22047239</td>
<td>27 %</td>
</tr>
<tr>
<td>RDcsim</td>
<td>19537514</td>
<td>3.5 %</td>
</tr>
<tr>
<td>Compiler</td>
<td>25086988</td>
<td>15 %</td>
</tr>
<tr>
<td>Simu</td>
<td>31529781</td>
<td>13 %</td>
</tr>
<tr>
<td>Assembler</td>
<td>30442494</td>
<td>13 %</td>
</tr>
<tr>
<td>Average</td>
<td>25728803</td>
<td>14 %</td>
</tr>
</tbody>
</table>

The benchmark programs consist of the Unix compress utility[16], two simulation programs (RDcsim and Simu), a compiler, and an assembler. The data in the table was obtained by executing each program to completion. The table lists the number of instructions fetched, the fraction of instructions executed that were memory instructions, and the fraction of instructions executed that were NOP instructions. NOP instructions are not required by the multi-threaded processor. Thus, the number of instructions fetched is lower, and the fraction of memory instructions is higher for the multi-threaded processor. (The averages in Table I are the arithmetic means of the entries in the corresponding column.)

5. Results

In this section we summarize the results of the trace driven simulations of the single- and multi-threaded processors. Figure 3 shows the hit ratio for the instruction cache vs. the size of the instruction cache for various degrees of set-associativity. Figure 4 shows the same data for the external cache. These figures illustrate that significantly larger caches are needed for a cache used with the multi-threaded MIPS-X to attain the same hit ratio as one used with the single-threaded MIPS-X.

Figures 3 and 4 also show that set associativity plays a much larger role in multi-threaded processors. Multi-threaded processors are susceptible to undesirable interactions between the independent execution streams. For a given size of cache, higher set-associativities mean the probability of one stream causing another stream’s data to be removed from the cache decreases.

5.1. Chip Area Compensation

As discussed in section 3.1, the multi-threaded processor requires more chip area to implement than does the single-threaded processor. Thus, to be fair to the single-threaded processor architecture, we cannot assume that the chip area can be increased without cost. We compensate for the added chip area needed by the multi-threaded processor by decreasing the amount of space available to its on-chip instruction cache. (The effect of this is to further reduce the hit ratio of the on-chip cache.)
0 64 128 256 512 1K 2K 4K 8K 16K 32K 64K

Figure 3. Instruction Cache Hit Ratio ($h_I$) vs.
Instruction Cache Size (words)

As was noted in section 3.1, the largest increase in chip area required to implement multi-threaded pipelining is due to the additional register sets necessary to provide the resources of a virtual MIPS-X processor for each stream. Considering that multi-threaded pipelining does eliminate certain features from the original execution pipeline, most notably the forwarding registers, and since the area needed to implement 128 registers is much greater than any of the other changes discussed for the multi-threaded MIPS-X, the total increase in chip area necessary for the multi-threaded MIPS-X processor can be approximated by the area necessary to implement the additional registers.

It can be argued that the 128 registers is about equal in chip area to the equivalent number of words of instruction cache[17]. (I.e., the area required per bit of a dual-ported register is roughly equivalent to a word of cache together with its tag data.) Thus, the effective area penalty of multi-threaded pipelining is a reduction in the on-chip cache size by 128 words. (I.e., eight lines for the Stanford MIPS-X.) In effect, we are making the approximation that a single-threaded MIPS-X and a multi-threaded version of the MIPS-X with eight fewer lines in its instruction cache would require the same chip area for implementation.

Figure 5 shows the instruction cache hit ratio vs. chip area normalized to the area of the Stanford MIPS-X processor. The chip area was determined using the fact that the instruction cache of the original Stanford MIPS-X, containing 512 words, occupies about one half of the chip area of the processor[9]. Note that the effect of chip area compensation is to further penalize the multi-threaded processor over the single-threaded processor. This is shown by the fact that the curves for the multi-threaded pipelined processor in Figure 3 are shifted to the right in Figure 5.

5.2. CPI

Figure 6 shows the measured CPI vs. chip size for various cache configurations. This figure shows the extent to which the ability of the multi-threaded processor to eliminate data and control hazards and to minimize the effects of memory latency has compensated for the lower hit ratios.

Figure 6 also shows that the break-even point for the multi-threaded processor occurs for a component density of between three and eight times that of the Stanford MIPS-X implementation. (The Stanford MIPS-X processor uses about 150,000 transistors[9]. The Intel 80860 processor uses more than one million transistors[18]. Thus, the break-even component density is reasonable.)

5.3. External Cache Effects

Figure 7 shows the CPI of both processors vs. external cache size for various instruction cache configurations. The external cache does not affect the CPI as greatly as the instruction cache since it has a low access rate. However, as for the instruction cache, variations in the associativity of the external cache do affect the CPI for the same reasons as discussed above.

6. Conclusions
In this paper, we have examined the performance of a multi-threaded pipelined RISC processor. By interleaving the execution of independent threads, data, control, and memory resource hazards are eliminated or reduced. However, multi-threaded pipelining suffers from a reduction in cache hit ratios and incurs a chip area penalty since it requires more space for the state of the virtual processors.

Performance models of both the Stanford MIPS-X processor, and a multi-threaded version were developed to show the potential benefits of multi-threaded pipelining. These models can be used to predict the performance of the processors once cache hit ratios are known.

Trace driven simulations were used to determine the effects on cache hit ratio of multi-threaded pipelining and to show the combined effects of multi-threaded pipelining on CPI. From these results, it is apparent that it is possible to design a multi-threaded pipelined processor that will provide a modest improvement in instruction throughput given a sufficiently dense implementation technology.

### References


