Instruction Fetch Deferral using Static Slack

Gregory A. Muthler David Crowe Sanjay J. Patel Steven S. Lumetta
Center for Reliable and High-Performance Computing
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign

Abstract

In this paper we present an approach to boosting performance and tolerating latency by deferring non-critical instructions into a deferred queue for later processing. As such, instruction deferral allows more critical instructions to be fetched, dispatched, and possibly executed, earlier.

We present methods for identifying deferrable instructions using previously investigated notions of instruction slack. In particular we use static slack to determine if an instruction is deferrable. The static slack of an instruction corresponds to the number of cycles an instruction can be delayed without impacting overall execution time when considering all dynamic paths from that instruction. A significant fraction of the dynamic instruction stream has enough static slack to be deferred by 10 or more cycles on an aggressive execution model. Futhermore, the small amount of register-based communication from deferred instructions to non-deferred instructions makes a deferral-based approach to fetch and execution very attractive.

We use a trace cache based microarchitecture to overcome some significant implementation challenges associated with instruction deferral. Overall, instruction deferral boosts the performance of a 4-wide processor by approximately 11% and an 8-wide processor by 6% on eight of the SPEC2000 integer benchmarks.

1 Introduction

Many conventional approaches to boosting processor performance rely on reordering instruction execution. In this paper, we describe a complementary approach to boosting performance by reordering instruction fetch. The basic premise is to identify those instructions that can be *deferred* for later fetch without increasing overall execution time. Deferring such non-critical instructions offers the possibilities of processing critical instructions earlier, and of using deferred instructions to fill execution bandwidth otherwise lost to instruction cache misses or branch mispredictions. While many instruction deferral techniques are possible, this paper focuses on deferring the fetch of non-critical instructions.

Figure 1 demonstrates instruction fetch deferral through a conceptual example. Instructions A through H are about to be fetched. Instructions A, B, G, and H are identified as critical, whereas instructions C, D, E, and F are determined to be deferrable. Instructions A, B, G, and H will be the next to be fetched and C, D, E, and F are queued to be fetched at some convenient point in the future. Instructions within each category are still fetched in order. That is, all critical instructions are fetched in program order relative to each other, and all deferred instructions are fetched in program order relative to each other. Deferred instructions can slip backwards relative to critical instructions. The key benefit of instruction deferral lies in the possibility of fetching the critical instructions G and H earlier.

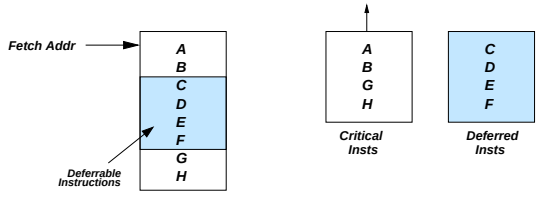

Figure 1. The block on the left is a block of instructions about to be fetched. The block is divided into instructions that are critical, and fetched immediately, and those that are deferred, fetched when bandwidth is available.

Estimates of slack provide the basis for separating instructions into critical and deferrable categories. We build on the foundation developed by Fields et al. [3], which defines and quantifies global, local, and apportioned slack at the instruction execution level, and investigates the benefits of prioritizing instruction execution according to dynamic measurements of slack.

We extend this work by introducing a static notion of

slack based on estimate of dynamic *global slack*, and use a threshold in this *static slack* to determine whether or not to defer an instruction. An instruction's global slack is the number of cycles an instruction's execution can be delayed without impacting the overall execution time of the application. An instruction's static slack is then the minimum global slack over all dynamic instances of the instruction. Identifying deferrable instructions in this manner creates computation clumps, as static and global slack are functions of instruction dataflow. The process of estimating and using slack is described in more detail in Section 2.

Instructions identified as deferrable using the global slack property are promising candidates for separation into a deferred instruction stream. Over 30% of all dynamic instructions have a static slack of ten or more cycles, for example. The similarity between this result and those of Fields et al. demonstrates that most dynamic instructions with substantial global slack can also be identified with a static measure generated by a profiler or compiler. Equally importantly, the deferrable instructions have significant register-independence from the critical instructions (those with smaller values of static slack). In particular, deferred instructions selected in this manner produce values that are predominately consumed by other deferred instructions, generating few register values that are required by the critical instructions. The deferred stream can therefore slip backwards in time relative to the critical stream, provided that values generated by the critical stream are correctly delivered to any deferred instruction that requires them, and that all memory-based communication happens correctly.

As such, instruction deferral is similar to the slack-based instruction scheduling performed by compilers [4]. The key difference is that dynamic deferral allows an instruction to be delayed without regard to region boundaries. Compilers using slack for instruction scheduling must either operate within region boundaries (e.g., hyperblock boundaries) or must replicate instructions that percolate across boundaries into all possible adjacent blocks. In dynamic deferral, an instruction with slack can move freely within the static regions spanned by the dynamic instruction window until spare fetch bandwidth is available or a critical instruction requires the output of the deferred instruction (a rare event).

While a conceptual description of instruction fetch deferral is fairly straightforward, two key issues must be addressed in devising a feasible microarchitectural mechanism. First, to enable fetch deferral, instructions must be arranged to allow separation of critical and deferred instructions prior to fetch. Second, the mechanism must maintain a consistent view of architectural state between critical and deferred instructions. That is, a deferred instruction must obtain proper source values and must provide any dependent instructions with its output value, even if the deferred instruction is fetched many cycles after its correct position in program order.

We propose the use of a modified trace cache mechanism to solve both problems in a hardware-centric manner. With the modified trace cache, traces are divided into two portions: critical and deferred. Traces are analyzed at construction time to identify dataflow to and from deferred instructions. This dataflow is encoded into the trace header and used at fetch time. The details of the hardware mechanism are provided in Section 3.

The instruction deferral mechanism is able to boost the performance of a deeply pipelined 4-wide machine by 11% on the average benchmark, and by 6% on a less-fetch constrained 8-wide machine.

The remainder of the paper is organized as follows. Section 2 describes the process of identifying deferrable instructions using static slack and presents measurements on the deferrable instruction stream. Section 3 describes our microarchitectural extensions for supporting instruction deferral. Section 4 provides details on our experimental setup. In Section 5, we present a performance characterization of instruction deferral. Related work is discussed in Section 6, and our conclusions appear in Section 7.

2 Instruction Deferral

The core of the instruction deferral technique is the notion of instruction slack. This section begins with an example motivating the use of slack for deferral and providing some insight on potential performance benefits. We next describe a compiler/profiler-based method for estimating static slack and present some characteristics of the deferred instruction stream that illustrate the potential benefits of using instruction fetch deferral.

2.1 Deferral using Slack

A recent study by Fields et al. [3] explores the notion of instruction slack. Broadly speaking, an instruction's slack refers to the number of cycles that its execution can be delayed without increasing the execution time of the program. An instruction on a critical path, for example, has no slack.

The study defines several variants of slack. *Local slack* refers to the time between execution of an instruction and execution of its consumers. If an instruction completes execution in cycle M, and the first of its consumers executes in cycle N, the instruction has a local slack of $N - M - 1$ cycles. In other words, the local slack of an instruction is the number of cycles that an instruction can be delayed without delaying any consumer instruction. *Global slack* of an instruction corresponds to the number of cycles that the instruction can be delayed without delaying the last instruction in the application. An instruction's global slack is equal to the sum of its local slack and the minimum global slack

of all consumers, and can be calculated by propagating local slack backwards through the dataflow graph. Fields et al. estimates that 40% of all dynamic instructions have at least 50 cycles of global slack on a 6-wide, dynamically scheduled machine.

For our deferral criterion, we introduce a static measure of instruction slack based on measurement of dynamic global slack. Global slack has the beneficial property of identifying clumps of instruction dataflow: that is, if an instruction has a large global slack, its consumers probably also do. We define the *static slack* of a static instruction to be the minimum global slack over all dynamic instances of the instruction. The deferral criterion is then a threshold based on static slack. In effect, static instructions with global slack above a threshold in all dynamic instances are identified as potential candidates for deferral.

Figure 2 shows a simple example that demonstrates the potential value of using static slack for instruction deferral. In the source code for this example, a small loop increments the elements of an array for use later in the code. The figure also shows the loop in terms of Alpha-like instructions and as a dataflow graph. The dashed boxes in the dataflow section represent loop iterations, and the load (LDQ, or load quad) instructions to the right represent array value consumers (not part of the loop). As illustrated by the dataflow, the instructions that increment the array elements (shaded) do not interact with the iteration control of the loop (unshaded).

As the consumption of the array elements occurs much later in the code, the store (STQ) instructions have substantial local slack, whereas the results of other instructions involved in the increment sequence are needed by the stores and are likely to have little or no local slack. In contrast, all instructions in the shaded region have substantial global slack. The local slack available to each store propagates backwards to the shaded add and load instructions. When the global slack is available for all loop iterations, static slack is also large, and the instructions are candidates for deferral.

The iteration control of the loop, on the other hand, is unlikely to have substantial static slack. In particular, the loop-ending branch is likely to be mispredicted on the last loop iteration, giving a global slack of zero for that dynamic branch instance. The dynamic instances of the loop iteration control instructions feeding into the final branch also have little global slack, thus static slack for these instructions is small.

A static slack above a sufficient threshold can thus separate the deferrable instructions in the array increment code from the critical instructions in the iteration control. Classifying the static loop iteration instructions as critical is important: although these instructions can be delayed in early iterations of the loop without affecting program execution

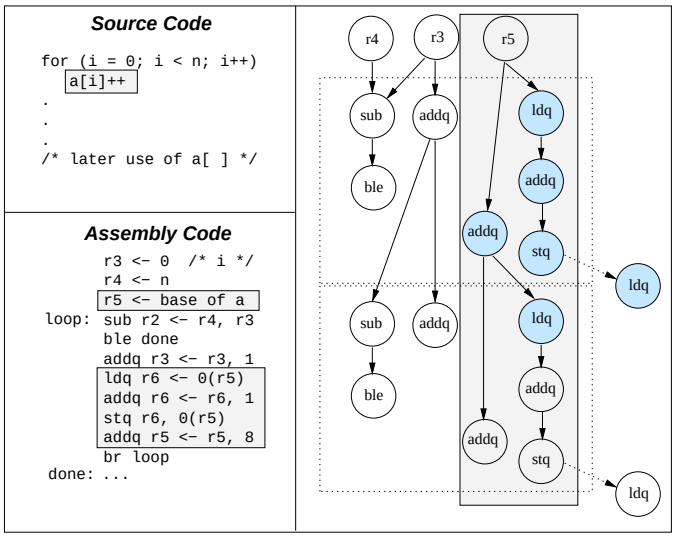

Figure 2. A simple loop shown as source, assembly, and dataflow. The array increment dataflow is not connected to that of the iteration control. Static slack is large for the increment code, but small for the iteration control.

time, the end of the loop is unpredictable, and should be detected as early as possible to avoid any misprediction penalty. With instruction deferral, critical instructions can be fetched ahead of deferred instructions, even if the deferred instructions precede the critical instructions in program order. For the loop of Figure 2, instruction deferral enables a 4-wide machine to fetch and execute one loop iteration control per cycle, whereas a machine without deferral takes two cycles to fetch a single iteration. Executing these iteration control instructions early can reduce execution time, as a processor without deferred fetch is likely to waste more cycles on the branch misprediction penalty at the end of the loop.

2.2 Estimating Slack

Instruction fetch deferral based on static slack requires that static slack be estimated in advance and used to reorganize instructions into deferrable and critical streams. Although dynamic identification techniques are possible, this paper focuses on an offline approach based on profiling. Using a profile-driven analysis, we estimate static slack for each instruction in a program binary and add a one-bit annotation marking the instruction as deferrable if its static slack is estimated to be above a threshold. As the profile execution may not contain all possible execution paths, we regard the slack value as an estimate.

Although we chose to use profiling to produce the in-

struction annotations, a compiler-based approach should produce comparable results. The binary decision necessary for each instruction should be a fairly straightforward calculation given a local dependence and control graph. While a compiler may lack some of the dynamic information available with a profile, our analysis similarly lacks some of the source-level information available to a compiler.

We use a fast execution model simulating a target machine to determine dispatch, execution, and retire times for all dynamic instructions. Given a small, profile data set, this simulator produces a trace file containing execution and dataflow information. The trace is then reversed to permit analysis starting from the last dynamic instruction backwards. This reversal process renders the slack calculation a linear-time scan of the execution trace file. Each instruction is processed from last to first. That is, consumers are encountered before producers, and slack is calculated by walking upwards through the instruction dataflow graph, monitoring both register and memory dataflow. For each static instruction, we maintain a record of minimum observed global slack. Once the entire trace is analyzed, the per-static-instruction slack measure is, in effect, the static slack for each instruction, as estimated from all dynamically observed paths.

The profiler annotates each static instruction as deferrable or critical based on the relation between the instruction's static slack and a threshold value. The experiments in this paper, for example, use threshold values of five, ten, and twenty cycles. The instruction deferral bit serves only as a hint to the hardware in the sense that, while incorrect deferral bits may degrade performance, they do not lead to incorrect program outputs. The marking policy used by the profiler can change the pool of deferrable instructions, but cannot impact the correctness of the code.

The potential impact of branch mispredictions make branch instructions a particularly interesting case for deferral. As mentioned earlier, a mispredicted branch is assigned a global slack of zero due to the fact that delaying execution of the branch further delays recovery and increases execution time. Correctly predicted branches, on the other hand, may be given large global slack, as their execution has no impact on performance. This assignment implies that the static slack of a branch is only non-zero if the branch is successfully predicted in every dynamic instance. Branches that are sometimes mispredicted are considered risky, and their condition calculations are pushed ahead of deferrable instructions to avoid misprediction penalties. While many different policies are possible in the treatment of branches for deferral, for this study we assigned all conditional and indirect branches to have no static slack.

2.3 Characteristics of the Deferred Stream

Deferring instructions based on static slack generates a deferred instruction stream that has some promising properties. In particular, deferrable instructions are plentiful, but their results are rarely used by critical instructions, and even more rarely used by critical instructions within the next few cycles.

Figure 3 shows the fraction of all static and dynamic instructions that are identified as deferred using the static analysis technique described above, using a threshold of ten cycles. The results exclude NOP instructions—they account for neither deferred nor critical instructions. On average across all benchmarks shown, approximately 30% of dynamic instructions are deferrable. The generation of a sizeable set of deferred instructions using a static annotation is an important result. Obtaining performance benefits through instruction reordering techniques such as instruction fetch deferral requires both that some instructions can be deferred and that other instructions benefit from their deferral. The results in the graph are also consistent with the result of Fields et al. showing that 40% of instructions have at least 50 cycles of slack when measured *per dynamic instance*. The static slack measurement thus demonstrates that most of the dynamically-identified instructions can also be identified using a conservative, static estimate of slack.

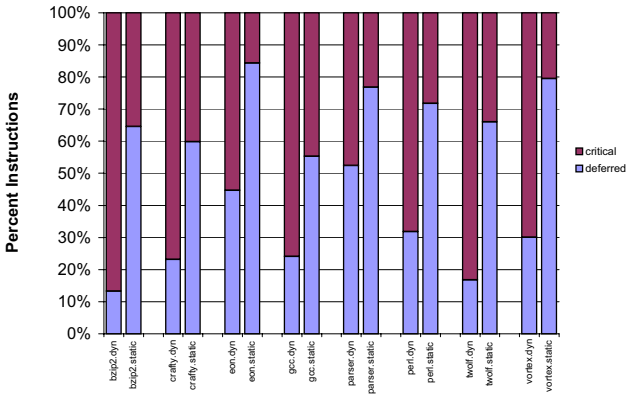

Figure 3. Percentage of dynamic and static instructions identified as deferrable according to a static slack threshold of ten cycles.

Allowing the fetch of deferred instructions to lag behind the execution of critical instructions requires that no critical instruction depend on the result of a deferred instruction. Frequent communication of this form forces synchronization between the two execution streams and defeats any gains achieved in allowing the critical stream to slip ahead. Specifically, a synchronization event is induced whenever an in-flight critical instruction requires a value that will be

produced by a deferred instruction (i.e., that instruction is not yet in-flight).

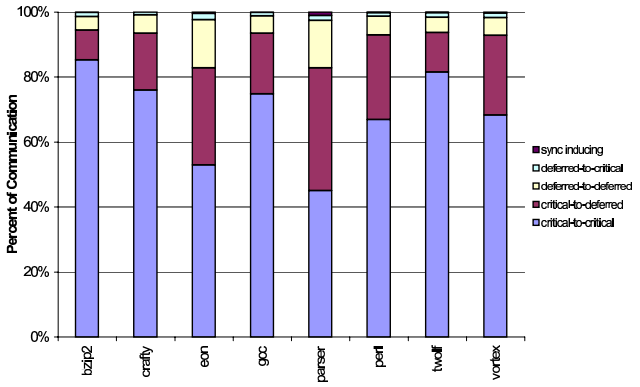

Figure 4. Communication frequency through registers as a function of producer and consumer instruction classifications.

Figure 4 shows the frequency of register-based communication between instructions. Communication is divided into four categories based on the classifications of the producer and the consumer: (1) a critical instruction reads a value produced by another critical instruction, (2) a deferred instruction reads a value produced by another deferred instruction, (3) a deferred instruction reads a value produced by a critical instruction, (4) a critical instruction read a value produced by a deferred instruction. The figure shows the percentage of all register reads—register communication arcs in a dataflow graph—that are in each category. Perhaps the most important feature is that the communication from deferred instructions to critical instructions is rare, accounting for approximately 1% of all register-based communication when averaged over the benchmarks. Further, synchronization inducing register-based communication between deferred instructions and critical instructions is even more uncommon. In contrast, deferred instructions often use the results of critical instructions.

With regard to communication through memory, the bulk of the deferred computation is similar in nature to that described in the example in Figure 2. Dataflow segments that calculate new values for in-memory data structures, such as the array in the example, are often candidates for deferral. The results of these computations are stored to memory and not consumed until some later point in the program. This delay provides the terminal store instruction of the data structure update with a large amount of local slack, and thus its associated computation flow a large amount of global slack. When this slack is available on all execution paths, static slack is high, and the instructions can be deferred. The use of global slack as the basis for static slack allows the computation to be connected to the store.

Clearly, the prevalence and ease of identification of static slack presents an opportunity. The question is how to exploit these properties for performance. In particular, we want to develop a mechanism, whether in hardware or software, to exploit the facts that (1) by definition, deferred instructions can be delayed, and that (2) deferred-to-critical register communication is minimal. In the next section, we describe a hardware model for exploiting this relative independence between deferred and critical instructions through instruction fetch deferral.

3 Fetch Deferral Microarchitecture

This section describes a microarchitecture for performing instruction fetch deferral. The primary benefit of deferral arises from the ability to optimize the order in which instructions are fetched. Critical instructions can be fetched before deferred instructions, even if the deferred instructions appear earlier in program order. To support such reordering of the fetch stream, the microarchitecture must address two challenging problems.

First, for maximum benefit, critical and deferrable instructions must be cached in separate stores. That is, for a particular fetch address, the fetch mechanism must be able to distinguish between deferrable and critical instructions *without* first fetching the instructions. To address this problem, we use a trace cache.

Second, architectural state must be maintained as if the program executes in a normal, sequential mode. The difficulty introduced by instruction fetch deferral is that the register renaming process typically assumes that the instruction stream is processed in program order. Usefully deferring instruction fetch requires out-of-order renaming. To address this problem, we use out-of-order register renaming techniques similar to those used in out-of-order and decoupled fetch techniques [2, 8]. The remainder of this section provides further detail and discussion of the issues.

3.1 Instruction Deferral Microarchitecture

Our mechanism for instruction fetch deferral builds on a superscalar processor with a trace cache, such as the Intel Pentium 4. We make modifications to the fetch and renaming mechanisms of this substrate to support deferral.

3.1.1 Fetch

Figure 5 illustrates changes to the processor's fetch mechanism. The trace cache storage space is partitioned into critical and deferrable stores. Each trace created by the trace fill unit is then divided, at trace construction time, into a critical section and a deferred section. This separation is facilitated by the deferrable bit annotation on each instruction. One

implication of the use of a trace cache to separate deferrable instructions is that only those dynamic instructions that occur in traces can be deferred. Instructions drawn from the instruction cache are treated as critical, regardless of their deferrability annotations.

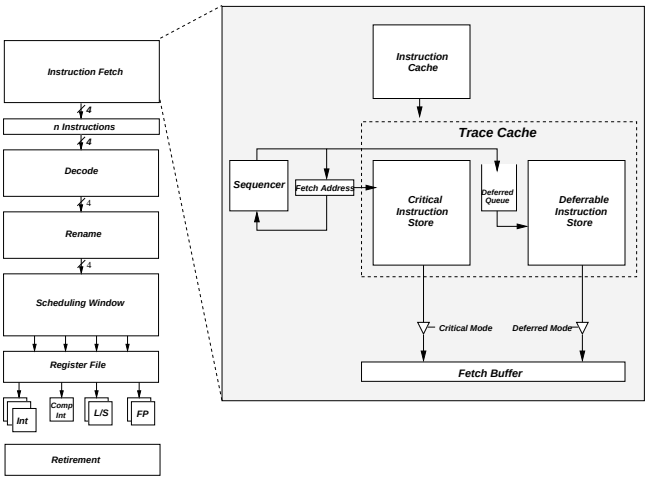

Figure 5. Fetch mechanism support for instruction deferral.

Each fetch address is presented to the critical partition of the trace cache on each fetch initiation. A hit in the critical partition causes the successful request to be enqueued into a Deferred Queue for later fetch. Fetch requests that miss in the critical partition are submitted to the supporting instruction cache and are *not* enqueued.

In our study, we position the trace fill unit to collect instructions at retirement time; the fill unit can also be positioned to collect instructions at fetch time, as is the case with the Pentium 4. The decision affects the results only indirectly: the Pentium 4 may cache speculated instruction sequences unnecessarily, and our approach may unnecessarily delay the caching of instructions. Like the Pentium 4 trace cache, our trace cache contains traces that can span multiple cache lines. Our traces contain up to 256 instructions.

While various control strategies between critical and deferred fetch are possible, we use a fairly simplistic one that gives priority to critical instructions. Fetch cycles that are empty (or partially empty) because of a cache or BTB miss or branch recovery, or because of the turn-around time between a trace cache and icache access, can be populated by deferred instructions. Furthermore, we use a simplistic branch confidence mechanism that selects deferred instructions after each indirect (and non-return) branch instruction. That is, we assume that prediction of any indirect branch other than a return instruction is of low-confidence, causing the deferred stream to be selected until the target address calculation completes.

3.1.2 Rename

The majority of the complexity involved with instruction deferral involves maintaining consistent architectural state while allowing maximum flexibility in deferring instructions. As noted at the beginning of this section, issues arise from renaming instructions out-of-order.

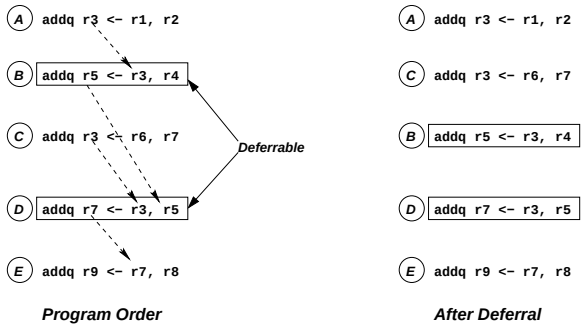

Figure 6. Renaming extensions required to support instruction deferral.

A simple example serves to illustrate the potential problems. Figure 6 shows five instructions before and after reordering, giving rise to several renaming scenarios associated with instruction deferral. Of the five instructions in the example, only B and D are deferrable. With instruction deferral, B and D may be fetched and renamed many cycles after their original positions in program order.

Three cases are of interest. For the first two, assume that the actual fetch order is the one shown on the right-hand side of the figure. Given this ordering (or any other allowed), the instruction deferral mechanism must establish dependencies as if the processing happened in program order.

The first case lies in the register dependence between instructions A and B. B must obtain the proper physical register assignment for R3, as established by A, even though the mapping of R3 in the Register Alias Table has been reassigned by instruction C.

A second case occurs in the dependence between instructions B and D through R5, which must also be maintained. Both instructions are deferred. When instruction D is renamed, it must obtain its source operand tag from the deferred version of R5, whereas D's version of R3 must be that produced by the critical instruction C.

Finally, consider instruction E, which uses a value produced by a deferred instruction. If the deferred instruction has not yet been fetched, this situation forces the instruction deferral mechanism to drain the deferred queue in order to

resynchronize the streams. In effect, fetch of instruction E causes instructions B and D to be demand-fetched.

To handle these three scenarios, we must provide each deferred instruction with a copy of the register mapping of its source operands as if renaming occurs in program order. That is, instruction B requires the register mapping of R3 as if it preceded instruction C (and not instruction D, as it might in deferred order.)

We address these problems by leveraging the notion of of atomic traces, or *frames* [6]. These atomic traces contain no side exits, and therefore are, in effect, similar to basic blocks. Each trace, when it is constructed, is analyzed and all internal communication is renamed using a technique similar to the one described by Vajapeyam and Mitra [9]. That is, each register source operand for each instruction within the trace is tagged to easily identify the producer. If the source operand is a live-in, the tag is the register identifier itself. If the particular instruction is deferred, however, and the source operand is a live-in, an extra level of indirection is required to find the correct source tag. Each trace is given a header that identifies all register live-ins required by deferred instructions. This header in effect establishes an indirect mapping between register live-ins to the tag encoded within the deferred instruction.

Coupled with this mechanism is a deferred bit associated with each entry for each architectural register in the Register Alias Table. This bit identifies whether a register is currently owned by the critical stream or the deferred stream. The setting of these bits must happen in program order—each trace header is processed by rename prior to the renaming of instructions within the trace. The trace header information is used to set the deferred bits of registers that are produced by deferred instructions. A subsequent critical instruction can overwrite the register, clearing the deferred bit. A critical instruction that reads a source register with the deferred bit set causes the machine to synchronize the deferred and critical streams by draining the deferred queue.

A final issue is that of memory-based dependencies between critical and deferred instructions. We adopt a simplistic policy by detecting violations at retirement (retirement is in-order) using the standard memory conflict mechanism–which in our case is an associative store buffer. If a violation is detected due to instruction deferral, the processor pipeline is flushed, and as a precautionary measure to prevent the situation from reoccurring (it can be a potential live-lock situation), the trace containing the delinquent deferred instruction is invalidated in the trace cache.

3.1.3 Retirement

To further simplify machine design, we assume in-order retirement of all instructions. Completed critical instructions must wait in the retirement queue until all previous deferred (and critical) instructions have been fetched, executed, and completed without error before they can retire. Exceptions and memory ordering violations are all checked at retirement. While in-order is not necessarily a requirement for instruction deferral, and one can devise mechanisms for out-of-order retirement, in-order retirement is not a significant constraint if enough in-flight physical register state is available.

To facilitate in-order retirement, each trace header must provide an indication of the position of each deferred instruction in order for the retirement logic to recreate the retirement order. In our case, because we use frames (which are more constrained variant of traces) and each frame is a single retirement entity, each frame header need only record the *number* of instructions in the frame and not the actual order. Once all instructions in a frame have retired, the entire frame retires. Any exceptional event causes the entire frame to be discarded.

3.1.4 Pipeline Flushing

Branch misprediction events cause the deferred stream to be resynchronized with the critical stream. That is, all fetch targets in the Deferred Queue (in Fetch) that are subsequent to the mispredicted branch are flushed. All other fetch targets are drained from the queue and executed.

4 Methodology

Our simulation framework is built upon the Alpha instruction-level simulator provided as the core of the SimpleScalar 3.0 tool set. Using the instruction simulator, we created a timing model of a superscalar processor that is capable of cycle-level simulation, including wrong path effects. This timing model serves as both the model from which slack calculation is performed (using the reverse trace analysis technique described in Section 2) and experimental model upon which all performance estimation is performed.

4.1 Benchmarks

We evaluate instruction deferral using eight of the twelve SPECINT 2000 benchmarks. The benchmarks were run to completion on all benchmarks (approximately 200M instructions per benchmark) using a set of profile input sets. Recall that a profiling pass is required to estimate static slack for each static instruction. Performance data for both the profile input set and another input set are presented in Section 5.

4.2 Machine model

We evaluate the instruction deferral mechanism in the context of deeply-pipelined 4-wide and 8-wide superscalar processor configurations. The specifics of the configurations are provided in Table 1. A trace cache miss results in a one cycle delay to access the supporting instruction cache. All NOP's occurring in the benchmarks consume no bandwidth whatsoever in any configuration; they are fetched and decoded for free. To first order, NOP's have no effect on the results in the next section.

Fetch	64KB Trace Cache, 4KB ICache
Sequencer	15-bit gshare, 1K-entry BTB
Pipeline	12 cycles (min) for BR res
Inst Window	512 instructions
L1 DCache	64KB, 2 cycles
L2 Cache	1MB, 10 cycles
Memory	50 cycles
4-wide ExeUnits	4 IALUs, 2 IMULs, 2 FLTs, 2 LdSt
8-wide ExeUnits	7 IALUs, 2 IMULs, 2 FLTs, 2 LdSt

Table 1. Configuration of Superscalar Processor Core.

With instruction deferral, the trace cache is physically split into a 48KB critical partition and a 16KB deferred partition. Also, the controlling policy for selecting between critical and deferred instructions is simplistic. As described in Section 3, critical instructions are always given priority over deferred instructions unless an unused fetch slot is encountered (e.g., due to a cache miss or branch recovery, etc.) or a deferred queue draining scenario is encountered. A maximum of ten fetch targets can be enqueued in the deferred queue.

5 Experimental Results

In this section, we evaluate various performance characteristics of instruction deferral. For the majority of experiments in this section, our tests use the profile input sets. However, as demonstrated later, the notion of static slack is a fairly robust metric across input sets.

5.1 Relative Performance

For the first experiment, we evaluate the impact of using deferral on the 4-wide and 8-wide baseline models. Figure 7 is a plot of the improvement in the number of instructions retired per cycle (IPC) when instruction deferral is added to each configuration. That is, the left bar for each benchmark represents the percentage improvement provided by deferral over the 4-wide baseline (4-wide vs. 4-wide+deferral), and the right bar represents improvement provided by deferral over the 8-wide (8-wide vs. 8-wide+deferral). The IPC for the baseline is provided below the corresponding bar. The IPC increases by an average of 11% across benchmarks on the 4-wide configuration and by 6% on the 8-wide configuration. In the instruction deferral case, all instructions with a static slack of five or more cycles are marked as candidates for deferral.

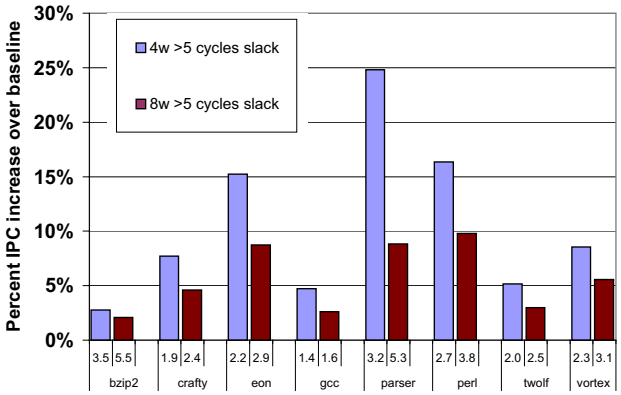

Figure 7. The effects of Instruction Deferral. The performance is provided for a 4-wide and an 8-wide configuration. The baseline is a 4-wide configuration.

Instruction deferral derives its benefits from two main sources:

- Critical instructions are fetched earlier than they are without deferral. Since deferred instructions only consume spare bandwidth, a deferred instruction rarely displaces a critical instruction in the fetch stream. Deferral enables critical instructions to potentially execute earlier. The realization of this potential is demonstrated by the drop in performance benefit between the 4-wide configuration of instruction deferral and the 8-wide. The 8-wide fetch mechanism delivers enough bandwidth such that fetching critical instructions earlier is of less benefit.
- Deferred instructions are used to fill fetch holes. Fetch holes arise from trace cache and instruction cache misses, partial fetches, BTB misses, and potentially branch mispredictions. Instruction deferral can be used to fill these holes with useful work. While avoiding cycles on an incorrect execution path is a profitable means for using deferral, branches likely to be mispredicted can not always be identified. Our scheme could potentially benefit from the use of a confidence predictor [5]. For our baseline models, because of the use of

a trace cache, a cycle of fetch is lost when switching between the trace cache and icache. This lost cycle is filled with deferred instructions when possible.

5.2 Average Deferral Distance

Recall that our simplistic selection mechanism heavily favors critical instructions. Deferred instructions are only fetched if spare bandwidth is available. Because of this policy, deferred instructions are often deferred by a significant number of cycles. Table 2 provides the median distance in *number of instructions* that a deferred instruction is deferred.

	Median deferral distance
bzip2	50
crafty	78
eon	104
gcc	31
parser	245
perl	101
twolf	45
vortex	86

Table 2. Median distance (in number of instructions) that a deferred instruction is deferred, as measured on the 4-wide configuration.

The deferral distance averages 90 instructions, corresponding to approximately 20 cycles of deferral. Even though the threshold for deferral was a static slack of five of more cycles, the number of synchronizing events (for example, caused by a critical instruction requiring a deferred value) is fairly small, on the order of one event per 100 cycles. The cost of a synchronizing event is not devastating—a stream of deferred instructions is spooled out from the deferred partition of the trace cache whenever a synchronization is required, with a few cycles of overhead to detect (in rename) that a critical instruction required a deferred value.

5.3 Using Different Slack Thresholds

Figure 8 plots the percentage increase in performance of the 4-wide+deferral configuration over the basic 4-wide as the deferral threshold is varied between 5, 10, and 20 cycles. That is, the profiler only marks an instruction as deferrable if its static slack is above the threshold. The results indicate that as the threshold is increased, the performance declines slightly. The primary reason for this decrease is that fewer instructions are categorized as deferrable. On average, 17% fewer dynamic instructions are marked as deferrable when the threshold is increased from 5 to 20. The benchmark eon behaves differently. One reason for this is that eon exhibits a higher incidence of synchronization events; a higher slack threshold reduces the synchronization frequency.

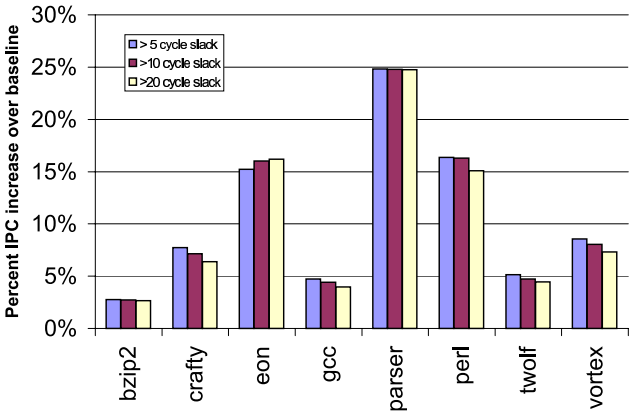

Figure 8. The performance impact of using static slack thresholds of 5, 10, and 20 cycles.

5.4 Evaluating the Brittleness of the Profile

Figure 9 provides a comparison between the performance benefits of deferral when the profile input set and the execution input set are the same and when they are different. The figure plots the performance of the 4-wide+deferral configuration when a *different* input set than the profile set is used for the measurement. The baselines for the two cases are different: that is, both the baseline and the 4-wide+deferral use the same input set for the measurement. The results for the alternate input sets are quite similar to, and in several cases better than, execution of the profile input set. The correlation demonstrates that our measurement of static slack is fairly robust for profiling. Static slack provides an estimate of global slack across all possible paths, and our profile-based estimation measures it across all dynamically observed paths.

5.5 Deferring Candidate Instructions

In all previous measurements, the only instructions that can be deferred are those that originate in the trace cache. Instructions that are fetched from the instruction cache cannot be deferred. Therefore, a degradation in potential performance arises from trace cache misses. In Figure 10, this effect is alleviated by allowing all instructions, regardless of their fetch origin, to be deferred. The results are shown for the 4-wide configuration with a deferral threshold of ten cycles. As might be expected, certain benchmarks exhibit

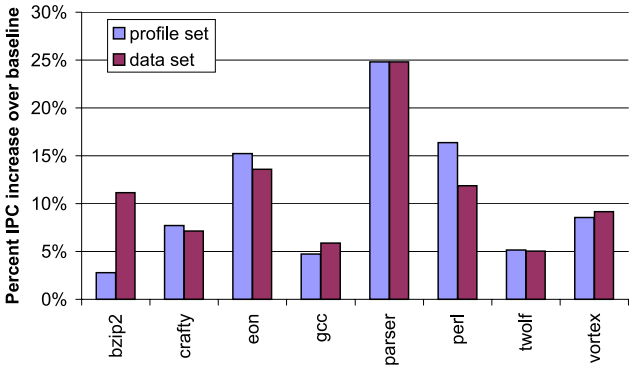

Figure 9. Performance increase for an input set other than the profile input set. Results are shown for the 4-wide architecture with a five cycle static slack threshold.

a high miss rate in the trace cache, and thus lower the observed potential performance of instruction deferral.

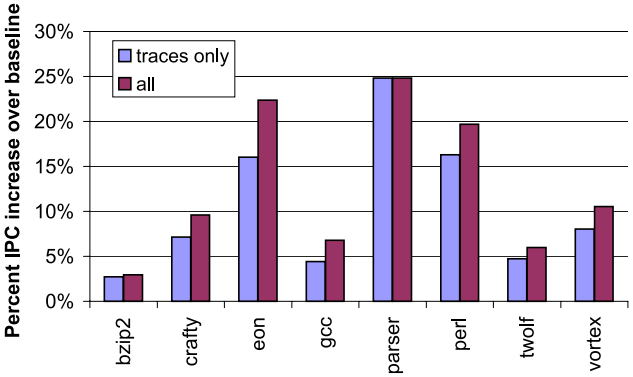

Figure 10. The effect of a more aggressive deferral mechanism in which instructions fetched from the instruction cache can also be deferred. The results for the trace-cache-only approach are shown for comparison.

6 Related Work

The most closely related work to instruction deferral is that of out-of-order fetch [8] and decoupled fetch [7]. Out-of-order fetch attempts to reorder instruction delivery by fetching and processing instructions beyond an icache miss using a non-blocking instruction cache. Decoupled fetch uses a queue of fetch addresses provided by a sequencer running ahead of the fetch point. Like deferral, both mechanisms attempt to fill vacant fetch slots generated by icache misses. Instruction deferral has the added advantage that it allows critical instructions to be moved upwards in the fetch stream, enabling them to be processed earlier. Fetch deferral bears similarity to hardware-initiated fetch of control independent code [1, 2]. In such schemes, the fetch stream need not follow program order but can jump ahead to control-independent fetch points.

The renaming mechanisms that enable fetch deferral are similar to those required any scheme that performs hardware-centric out-of-order renaming. In particular, out-of-order fetch [8] and control-indepedent fetch [2] require a mechanism to reconstruct original program dependencies in light discontinuities in the fetch stream.

There is a correspondence between this work and previous work on slack-based instruction scheduling in compilers [4]. As mentioned previously, the dynamic approach described here provides more flexibility in how late an instruction can be fetched. A slack-based scheduler in a compiler can defer an instruction until the end of a region, but must replicate the instruction along all potential target paths if the instruction is to be deferred further.

The use of slack to control various processor selection mechanisms was explored by Fields et al [3]. In their work, they explored the use of dynamically-measured instruction slack to control (and defer) instruction execution. Execution deferral is similar in extent to the fetch deferral explored here. In this paper, all instructions have equal priority once they have been fetched.

7 Conclusions

Instruction fetch deferral is a novel mechanism for optimizing the fetch order in which an application is processed. Using static slack as the deferral criterion, we identify computation dataflow clumps, or streams, whose results are not needed for some time in the future, and can therefore be deferred for later instruction fetch. We demonstrate that dataflow sections with large static slack have interesting properties that make them amenable to decoupled, or threaded, processing. In particular, such instructions constitute over 30% of the dynamic instruction stream and produce relatively few register values that are needed by more critical, slackless, instructions.

We describe an instruction deferral mechanism constructed atop a trace cache-based superscalar processor. The use of a trace cache helps to address two of the critical hurdles associated with instruction deferral. First, critical instructions are cached, in trace form (actually, frame form), separately from deferred instructions, enabling critical instructions to be fetched independently of those that are deferred. Second, trace headers help to maintain consistent architectural state despite the out-of-order renaming associated with instruction deferral. Trace headers are processed

in program order and are used to collect register mappings for subsequent deferred instructions and to inform the renamer of register writes by the deferred stream.

Using a timing simulator, we estimate that an instruction deferral mechanism boosts the performance of a 4-wide processor by an average of 11% on eight of the SPECINT2000 benchmarks. The benefit on an 8-wide processor is approximately 6%. The majority of the benefit of deferral comes from two sources: (1) critical instructions are potentially fetched and processed earlier, and (2) otherwise unused fetch bandwidth (due, for example, to cache misses) is filled with useful work. In essence, instruction deferral discovers cache misses cycles earlier. We also provide some evidence on the robustness of our profile-derived static slack estimations. Measurements of instruction deferral performance benefit taken on different input sets yielded similar results to those taken on the profile set.

We present one hardware-centric framework for optimizing the instruction fetch via slack-based deferral. Other frameworks are possible, including holistic approaches that alleviate the renaming complexity via an integrated hardware-software approach.

8 Acknowledgements

We thank the other members of the Advanced Computing Systems group, as well as David Kaeli and Antonio Gonzalez, for providing feedback during various stages of this work. This work was funded in part by NSF CAREER grants NSF-CCR-00-92740 and NSF-ACI-99-84492, and gracious support from AMD, Intel, and Sun.

References

[1] H. Akkary and M. A. Driscoll. A dynamic multithreading processor. In *Proceedings of the 31th Annual International Symposium on Microarchitecture*, 1998.

[2] C.-Y. Cher and T. N. Vijaykumar. Skipper: A Microarchitecture for Exploiting Control-Flow Independence. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, 2001.

[3] B. Fields, R. Bodik, and M. Hill. Slack: Maximizing performance using technological constraits. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, 2002.

[4] P. Gibbons and S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of ACM SIGPLAN Symposium on Compiler Construction*, pages 11–16, 1986.

[5] E. Jacobsen, E. Rotenberg, and J. E. Smith. Assigning confidence to conditional branch predictions. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 142–152, 1996.

[6] S. J. Patel, T. Tung, S. Bose, and M. M. Crum. Increasing the size of atomic instruction blocks using control flow assertions. In *Proceedings of the 33th Annual International Symposium on Microarchitecture*, 2000.

[7] G. Reinman, T. Austin, and B. Calder. A scalable front-end architecture for fast instruction delivery. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, 1999.

[8] J. Stark, P. Racunas, and Y. N. Patt. Reducing the performance impact of instruction cache misses by writing instructions into the reservation stations out-of-order. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 34 – 43, 1997.

[9] S. Vajapeyam and T. Mitra. Improving superscalar instruction dispatch and issue by exploiting dynamic code sequences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 1–12, 1997.