To appear in The 30th International Symposium on Computer Architecture, June 9–11, 2003

Parallelism in the Front-End

Paramjit S. Oberoi and Gurindar S. Sohi
Computer Sciences Department, University of Wisconsin–Madison
`{param,sohi}@cs.wisc.edu`

Abstract

As processor back-ends get more aggressive, front-ends will have to scale as well. Although the back-ends of superscalar processors have continued to become more parallel, the front-ends remain sequential. This paper describes techniques for fetching and renaming multiple non-contiguous portions of the dynamic instruction stream in parallel using multiple fetch and rename units. It demonstrates that parallel front-ends are a viable alternative to high-performance sequential front-ends.

Compared with an equivalently-sized trace cache, our technique increases cache bandwidth utilization by 17%, front-end throughput by 20%, and performance by 5%. Parallelism also enhances latency tolerance: a parallel front-end loses only 6% performance as the cache size is decreased from 128 KB to 8 KB, compared with a 50–65% performance loss for sequential fetch mechanisms.

1 Introduction

Increasing the exploitation of parallelism, especially instruction-level parallelism, has been the focus of architectural techniques for several decades. Starting with serial, in-order operation of all stages in instruction processing, processors have gradually become increasingly parallel in different stages. The first step toward more parallelism was to increase the width of execution, i.e., processing multiple operations at the same time, but in program order. The next step was to remove the artificial constraints due to serial processing and perform out-of-order processing, further increasing parallelism.

Despite the use of parallel processing techniques in the back-end, the front-end stages of the processing pipeline—instruction fetching, decoding, and renaming—have remained sequential processes. Increasing parallelism in the back-end has placed increasing demands on the front-end, and processor architects have responded by increasing the width of sequential front-ends. We believe that the brute force solution of increasing the width of the front-end pipeline stages while retaining their sequential nature is not the preferred approach for future processors. Accordingly, we propose techniques to parallelize the front-end of the processing pipeline. The techniques that we propose are able to achieve better front-end performance and, in most cases, better or equivalent overall performance, than known high-performance sequential front-ends.

In Section 2 we discuss the limitations of sequential front-ends and introduce parallel front-ends. The next two sections describe our proposed parallel front-end in detail: Section 3 describes the parallel fetch unit and Section 4 describes the parallel rename unit. Section 5 presents an evaluation of our proposal, and Section 6 concludes the paper.

2 Sequential and Parallel Front-Ends

We use the term *front-end* to denote the mechanism(s) responsible for supplying instructions to the execution units (the *back-end*). The front-end includes the fetch unit, the rename unit, and other support structures (e.g., a branch predictor). The aim of a high-performance front-end is to keep the later stages of the processing pipeline busy by providing them with a sufficient number of instructions every cycle. We start our discussion of the front-end with the fetch unit since it is the earliest part of the pipeline. The discussion of renaming is postponed until Section 4, at which point we will be able to discuss the implications of a parallel fetch unit on the rename unit.

A sequential fetch unit relies on being able to fetch long contiguous sequences of instructions every cycle. Increasing the throughput of a sequential fetch unit therefore requires increasing the length of the contiguous instruction sequences fetched each cycle. As the required length becomes longer, it becomes progressively more difficult to achieve. Section 2.1 discusses these difficulties in more detail.

To overcome these difficulties, we propose that higher fetch throughput be achieved by using parallelism: several sequential fetch units fetching different fragments of the program in parallel, rather than a single fetch unit trying to fetch longer sequences of instructions. Section 2.2

describes parallel fetch and the advantages of parallel fetch over sequential fetch.

2.1 Limitations of Sequential Fetch

Most limitations of sequential fetch mechanisms result from the fact that they are designed to fetch instructions that are stored in consecutive memory locations (i.e., stored sequentially), even though the arbitrary control flow structure of programs generally cannot be mapped onto a static sequential storage order. Thus, to improve fetch throughput, the mechanism must fetch a large number of instructions that are not consecutive in the static program representation. This is accomplished in one of several ways, used either in isolation or in combination.

The first way is to rearrange the static code so that basic blocks, and therefore instructions, that are likely to be consecutive in the dynamic program are also consecutive in the static program [2, 5, 18]. This rearrangement may be done statically, for example at link time [12], or dynamically [1]. Since this approach is not always successful, especially as the demands on fetch bandwidth are increased, other approaches have been necessary.

The second way is to design hardware that can read multiple cache lines simultaneously and thus fetch instructions that are contiguous in dynamic program order but not in the static program. A collapsing buffer [7] is an example of this approach. Studies have shown that this approach is also unable to deliver high fetch throughput [20].

A third way is to observe the dynamic execution order as the program executes and cache instructions in their dynamic execution order. A trace cache [14, 16, 20] is an example of this approach. Accessing a single entry in a trace cache returns multiple instructions that were not necessarily contiguous in the static program, thereby allowing a sequential fetch unit to achieve a high fetch throughput. This approach uses additional storage resources that might be put to more productive use if other approaches to achieve high fetch throughput were possible. Furthermore, it makes inefficient use of storage resources due to fragmentation and duplication [17].

Sequential fetch mechanisms, like in-order issue mechanisms, are also susceptible to stalls. A stall condition like an instruction cache miss prevents any further fetch activity until the stall ends. Increasing stall latencies increase the time in which no fetch activity occurs.

Many enhancements to these mechanisms have been proposed [9, 10, 15, 19, 22], but each of these inherits some problems from the sequential fetch mechanism upon which it is based.

2.2 Parallel Fetch

A parallel fetch unit achieves higher fetch throughput by fetching multiple (possibly discontinuous) instruction blocks in parallel, rather than increasing the width of individual blocks. Parallelism enables higher fetch throughput without being subject to the limitations of sequential fetch mechanisms.

Fetching multiple discontiguous blocks of instructions every cycle requires predicting multiple points in the upcoming dynamic instruction stream and fetching instructions from each of those points in parallel. Thus, instead of a single program counter (PC), there are multiple PCs, each representing the start of a fragment of the dynamic instruction stream. Instructions from each of these fragments are fetched concurrently using multiple *sequencers*. A sequencer is a mechanism that sequences through instructions in program order (like a sequential fetch unit).

The basic idea is very similar to instruction fetch in Multiscalar [3, 21]: Multiscalar divides the sequential instruction stream into tasks and assigns each task to a processing element. All processing elements fetch and execute the assigned task in parallel. However, parallel fetch in Multiscalar is an artifact of a fully clustered microarchitecture. The technique we are proposing is completely general and makes no assumptions about the back-end.

The net throughput of a parallel fetch unit is the aggregate throughput of all the sequencers, rather than being constrained by the throughput of a single sequencer. The maximum achievable throughput is still limited by the instruction cache bandwidth, but unlike a sequential fetch unit, the available bandwidth can be better utilized since fetch can be reordered to accommodate constraints of the instruction cache (e.g., misses and bank conflicts).

Parallelism in the fetch unit, like parallelism in other parts of the processor, also increases latency tolerance. If one of the sequencers experiences a cache miss, others can still continue fetching the fragments assigned to them. Thus, the cache miss can be overlapped with the fetch of other useful instructions, or with other cache misses. In addition, the cache-missing sequencer could fetch a different fragment while the miss is serviced and later return to the original fragment.

Finally, a parallel fetch unit has various benefits not related to performance. Building an instruction delivery mechanism out of replicated hardware simplifies its design and hence makes verification easier. Use of narrower sequencers may simplify their circuitry. It also makes the fetch unit more flexible: parallel fetch units easily lend themselves to fetching multiple threads, fetching down both paths of frequently mispredicted branches, fetching instructions from reconvergent points, etc.

The Alpha EV8 processor design included a fetch unit capable of a limited degree of parallelism [8]. It could fetch two discontiguous cache blocks simultaneously, like

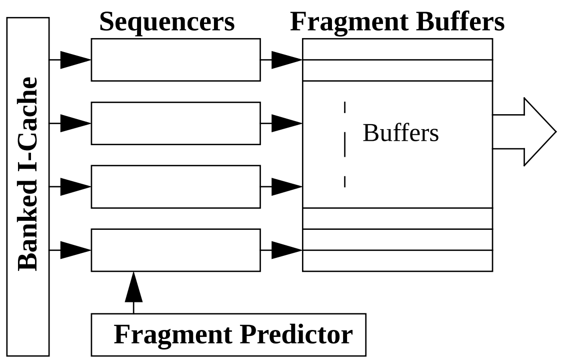

Figure 1. Parallel Fetch Unit

a collapsing buffer. The blocks could be from two different threads, but each thread was fetched in-order.

3 Parallel Fetch using Multiple Sequencers

A sequential fetch unit contains a single sequencer which fetches one or more lines from the instruction cache every cycle depending on the design of the cache and the sequencer. The required instructions are extracted from the fetched cache lines, and the output from the fetch unit is a block of instructions in program order.

Figure 1 illustrates a parallel fetch unit based on the design we proposed earlier [13]. We refer the reader to the original paper for a complete description, but a short overview of the design follows. It consists of multiple sequencers that write the fetched instructions into an array of fragment buffers. These buffers provide temporary storage until instructions can be merged into the in-order instruction stream. A *fragment predictor* predicts control flow on the granularity of fragments, and each predicted fragment is assigned a fragment buffer. Sequencers fetch multiple fragments into the corresponding fragment buffers in parallel. The instruction cache is banked so that it can handle multiple requests simultaneously (barring bank conflicts).

Instructions are read out of fragment buffers in oldest-fragment-first order, i.e., program order. Since instructions exit the fetch unit in program order, no changes are required to any other stage of the processor pipeline except support for training the fragment predictor and recovering its state on mispredictions.

3.1 Fragment Selection and Prediction

A program fragment is a portion of the dynamic instruction stream. The entire dynamic execution stream of the program can be obtained by concatenating all fragments. This is similar to the idea of *traces* [20] or *tasks* [21], except that fragments are completely general, whereas the other terms make assumptions about the nature of fragments or about how they are processed.

Conceptually, a fragment predictor only needs to predict fragment boundaries. Intra-fragment control flow can be predicted by each sequencer using a local mechanism. A variety of fragment predictor designs are possible, but in this paper we use the trace predictor proposed by Jacobson, Rotenberg, and Smith [11]. Since the trace predictor predicts trace addresses as well as branch directions for all branches in the trace, local branch predictors are not required.

The heuristics used to split the instruction stream into fragments are fairly similar to commonly used trace selection heuristics: fragments are terminated at all indirect branches, at any conditional branch after the eighth instruction, or at the sixteenth instruction. These heuristics are discussed in more detail in our prior work [13].

3.2 Fragment Buffers

Fragment buffers are FIFO queues of instructions large enough to store an entire fragment. In addition to instructions, they store other information relating to the fragment: its starting address, the current PC, and branch predictions from the fragment predictor. As instructions are fetched into a fragment buffer, the PC is updated to reflect the next instruction to be fetched. When the entire fragment has been fetched, a flag is set indicating that the buffer is complete.

Once all instructions have been read from a buffer by the next pipeline stage, the buffer is marked unused, but its instructions are not discarded. If the same fragment is encountered again before its buffer has been reallocated, the instructions are reused instead of being fetched again from the instruction cache. Depending on the benchmark, 20–70% of fragments can be reused with just 16 fragment buffers [13].

Thus, the fragment buffers act like a very small trace cache, and the sequencers act like a prefetch/fill mechanism. A large trace cache can exploit most of the locality in the instruction stream but typically has a relatively slow sequential fill mechanism. The fragment buffers, on the other hand, can exploit only a fraction of the locality, but have a powerful parallel fill mechanism. Depending on design constraints, a fetch mechanism could lie anywhere on this spectrum. A complete study of this design space is beyond the scope of the paper.

3.3 Performance Intuition

The time taken by this mechanism to construct an individual fragment is typically *more* than the time that a sequential fetch mechanism would take because (1) the individual sequencers are not as wide as the monolithic fetch unit that they are replacing, and (2) access to the instruction cache is shared among multiple sequencers. However, since the fetch rate is higher than the commit rate, sequencers are usually fetching fragments far ahead of the back-end. Therefore, when the rename stage begins renaming instructions from a fragment, usually the entire fragment has already been fetched. Consequently, this mechanism operates like a just-in-time fragment/trace constructor, giving the illusion of a large trace-cache-like mechanism to the rest of the pipeline. Our simulations indicate that 84% of fragments are completely constructed before they are sent to the rename stage—as compared to an average trace cache hit rate of 87%.

Using parallelism to overcome the higher latency of constructing individual fragments also makes this mechanism more latency tolerant than sequential fetch. Since the fetch of different fragments is overlapped, a cache miss can be hidden behind the fetch of instructions from other fragments, or the latency of multiple cache misses can be overlapped.

Finally, since this mechanism uses a conventional instruction cache instead of a trace cache, it is able to utilize cache space more effectively. Therefore, programs with large working sets are likely to perform better—provided that the fragment predictor isn't overwhelmed by the code size as well. Although performance of both the predictor and the cache suffers as the code size grows, a predictor can usually perform acceptably over a greater range of code sizes than a cache since the space occupied by information about a fragment in the predictor is a small fraction of the space occupied by the fragment in a cache. For the same reason, it is easier to resize a predictor to handle larger programs than it is to resize a cache.

3.4 Limitations of Multiple Sequencers

Relying on parallelism for higher throughput has the result that during the time when parallelism is low, the throughput is also low. For parallel fetch, this occurs immediately after fetch is redirected due to a control misprediction. It takes a few cycles for all sequencers to become active again since the fragment predictor makes only one prediction every cycle.

The problem is further exacerbated if the parallel fetch unit is feeding a sequential decode and rename stage. In this case the effective throughput is limited to the throughput of a single sequencer since all the instructions fetched by the first sequencer must be renamed before any of the instructions fetched by other sequencers, even though they may be fetched earlier. A similar problem occurs when a sequencer encounters a cache miss. Later fragments are fetched into fragment buffers before the cache-missing fragment, but instructions from later fragments cannot be forwarded to the later stages of the pipeline until the cache miss is serviced.

Thus, although parallel fetch is able to maintain a high fetch rate at most times, and is able to reach its steady state fetch rate soon after a fetch redirect, sequential renaming of instructions exposes latencies that could otherwise have been hidden by parallelism.

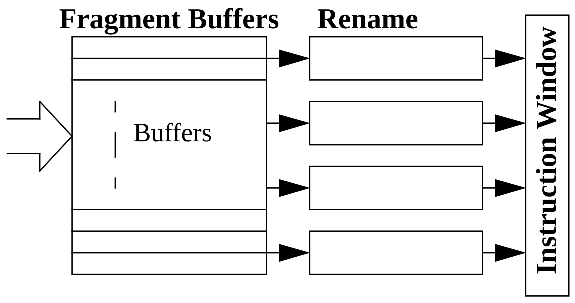

Figure 2. Parallel Renaming

4 Parallel Renaming

The cause of the limitations described in Section 3.4 is that a parallel pipeline stage is feeding a serial pipeline stage; therefore, the instruction stream must be serialized. For some stages in the pipeline, such as the commit stage, this may be unavoidable; but could this loss in performance be avoided for intermediate pipeline stages?

Serialization can be avoided if the rename stage can be built in a parallel fashion as well, as shown in Figure 2. A single monolithic renamer is replaced by multiple identical renamers. Each individual renamer renames a single fragment—just like each individual sequencer fetches a fragment—and all renamers operate concurrently to rename multiple fragments in parallel.

In addition to avoiding serialization of the instruction stream, a renaming unit composed of smaller replicated rename units may allow higher clock rates since small, loosely-coupled structures can typically be clocked faster than a large monolithic structure, and the critical path of a renaming unit is shorter if the number of instructions to be simultaneously renamed is smaller.

The main issue in building such a rename unit is ensuring that the consumer instruction in a RAW dependence pair gets renamed correctly. Figure 3 illustrates the problem with two example program fragments. Instruction I2 from fragment 2 uses the mapping created for logical register R1 when instruction I1 is renamed. A sequential renamer always renames I1 before I2, so this mapping is always available when I2 is renamed; however, if the two fragments were renamed in parallel, I2 may be renamed before I1. Since I2 cannot be renamed until the mapping created by I1 is available, a parallel renaming mechanism must do one of these two things [22]: (1) delay renaming I2 until I1 has been renamed, or (2) rename I2 speculatively and ensure that I1 maps its output to the predicted register.

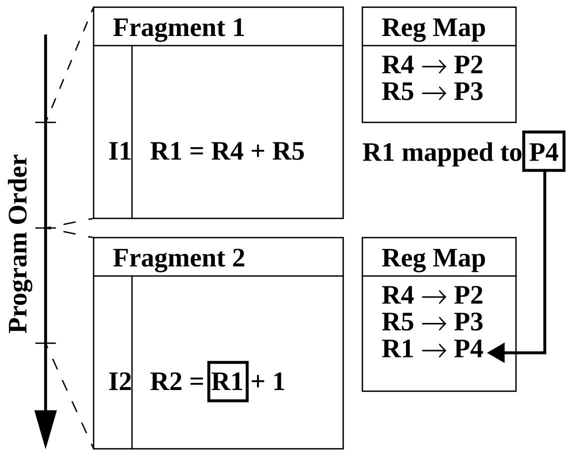

Figure 3. Difficulty with Parallel Renaming: I2 can not be renamed unless the latest mapping of R1 (produced when I1 is renamed) is known.

The first solution is similar to that used by Multiscalar [3, 21]. When a fragment is renamed, the hardware determines which register mappings are not yet available (either via a predictor, or using compiler information). The corresponding instructions are delayed until the missing mappings are available. This solution requires the renamers to exchange information about the register map table as they rename instructions.

The second solution is similar to that used by Skipper [6]. It is based on the observation that renaming an instruction and creating the corresponding register mapping do not necessarily have to be done at the same time. Before starting to rename each fragment, the hardware determines (speculatively or otherwise) which new mappings will be created by that fragment. Future fragments can use these mappings to rename instructions correctly. This allows multiple fragments to be renamed in parallel, but the process of creating new mappings must be performed serially for each fragment. The serialization is not a serious performance limiter since the process of creating these mappings only involves making a copy of the renaming table and allocating a group of physical registers. Some of this serialization could also be removed by having some conventions that restrict which physical registers the logical registers of a fragment can be mapped to.

The first solution removes serialization completely, but is more complex since it requires delaying instructions and communication between renamers. Moreover, delaying instructions increases the time fragments spend in the fragment buffer, which limits the ability of the fetch unit to look ahead in the instruction stream.

The second solution does not completely eliminate serialization, but it is simpler since the operation of individual renamers is largely unchanged and a renamer communicates only at the beginning of its renaming process. Since the fragment predictor limits the maximum throughput of this mechanism to one fragment per cycle anyway, the second solution has the benefit of lower latency without any significant performance loss. The rest of this section describes our proposed implementation of the second solution in greater detail.

4.1 Live-out Prediction

Two predictions are made for each fragment: (1) the logical registers written by that fragment, and (2) the instructions that write the live-out values seen by future fragments (i.e., for each register, the last instruction which writes a value to that register). Predicting this is relatively straightforward: the first time a fragment is seen, the live-outs are recorded in a table, and that table is used to make predictions later.

The live-out registers are stored as a bitmap containing one bit for each register, with a 1 indicating that the corresponding register is a live-out. The instructions corresponding to the last writes are stored as a bitmap containing one bit for each instruction in the fragment, with the n^{th} bit indicating if the n^{th} instruction in the fragment is a last write. In our example implementation, the predictor has a 4-bit tag to detect aliasing and it is indexed by a hash of the address and predicted branch directions of the fragment.

4.2 Fragment Renaming

In addition to the live-outs, the length of each fragment is also predicted. Every cycle, free reorder buffer entries are allocated to the oldest fragment which has not yet been allocated entries for all its instructions. Instructions from later fragments can be written into the reorder buffer before earlier fragments have been fetched completely.

Fragments are renamed in two phases. In phase 1, all the predicted live-outs of the fragment are allocated new physical registers. A copy of the register map table with the newly allocated registers is sent to the next renamer. In phase 2, instructions in the fragment are renamed sequentially. The physical register allocated in phase 1 is used for the result when renaming an instruction corresponding to a live-out value; otherwise, a new physical register is allocated.

Phase 1 is performed one fragment at a time, in program order; phase 2 is performed in parallel for multiple fragments. Phase 2 is performed only after the fragment has been allocated space in the reorder buffer.

4.3 Mispredictions

There are two live-out misprediction scenarios: (1) under-predicting the live-outs, and (2) over-predicting the live-outs. Since two predictions are made per fragment to determine live-outs (registers and instructions), we need to detect a total of four misprediction conditions: (1) A write to a register that was not predicted to be a live-out; (2) No writes to a register that was predicted to be a live-out; (3) A write to a live-out register after the predicted last write; (4) No instruction predicted to be the last write of a live-out register.

Condition 4 supersedes condition 2. Conditions 1 & 3 can be easily detected during the rename process, and condition 4 can be detected after all instructions in a fragment have been renamed. On a misprediction, all future fragments are squashed. Alternatively, it is possible to selectively re-execute the incorrectly renamed instructions, but the extra expense may not be justified if the misprediction rates are sufficiently low.

In addition, another source of misprediction is the fragment length. Overpredicting the fragment length is safe—it only wastes resources. Underprediction is handled by squashing all future fragments.

4.4 Parallel Renaming with Sequential Fetch

The renaming scheme proposed above only depends on the existence of a set of fragment buffers so that multiple fragments can be read out and renamed in parallel. The details of how the fragment buffers are filled do not affect parallel renaming. For example, even if the fetch mechanism was a trace cache which placed one trace every cycle into a free fragment buffer, the parallel renaming mechanism described could be used without any changes.

5 Evaluation

We modelled three different front-ends—a conventional sequential mechanism, a trace cache, and the parallel front end described in this paper—using an execution driven simulator based on the SimpleScalar toolkit [4]. Only the system call emulation and the instruction definitions were taken from SimpleScalar; the out-of-order timing model was rewritten entirely. Since improving the front-end is only useful if it is a bottleneck, we simulate a 16-wide out-of-order superscalar processor with abundant functional units and large caches. Table 1 describes the simulated processor in detail.

The conventional sequential front-end is labelled **W16** in the rest of the paper. **W16** fetches at most 16 instructions sequentially starting at a given PC until it encounters a taken branch or a cache-line boundary. We assume that there is no restriction on the number of branch predictions

Table 1: Simulation Parameters

Width	Fetch, decode and commit at most 16 instructions per cycle
Functional Units	16 Int adders, 4 Int multipliers, 4 FP adders, 1 FP multiplier, 4 load/store units.
Window	256 entry instruction window
L1 Caches (Instr. & Data)	64 KB, 2-way set-associative, 1 cycle access time, 64 byte blocks (16 instructions per cache block)
L2 Cache (Unified)	1 MB, 4-way set-associative, 10 cycle access time, 128 byte blocks
Memory	100 cycle access time
Trace & Fragment Predictor	DOLC [11], 64K entry primary table, 16K entry secondary table, D=9, O=4, L=7, C=9
Parallel Fetch and Rename	16 fragment buffers, 16 instructions each (1 KB). 2-way 4K entry live-out predictor (84 bits per entry, 42 KB)

in a cycle, i.e., fetch can proceed past any number of not-taken branches in a cycle. The cache can supply only one cache line every cycle, so fetch must stop at cache-line boundaries. Fetch stops at taken branches regardless of whether the target is in the same cache line. The L1 instruction cache size is 64 KB. `NOP` instructions are eliminated very early in the pipeline and do not count towards the number of instructions fetched, renamed, or committed. Branches are predicted using a trace predictor.

TC represents a 2-way set associative trace cache with a maximum trace size of 16 instructions. On a cache hit, the trace cache can supply an entire trace in a single cycle. On a miss, instructions are fetched using the **W16** mechanism. The processor contains an L1 instruction cache in addition to a trace cache, and space is divided equally between the instruction cache and the trace cache[1]. We simulate two trace cache configurations: (1) **TC** denotes a 32 KB trace cache and a 32 KB instruction cache, and (2) $\mathbf{TC_{2x}}$ denotes a 64 KB trace cache and a 64 KB instruction cache. $\mathbf{TC_{2x}}$ uses double the amount of L1 instruction storage as **W16**. As in the case of **W16**, `NOP` instructions are not counted towards trace size.

PF represents the parallel fetch mechanism based on multiple sequencers described in Section 3. It contains 16 fragment buffers of 16 instructions each. Two different **PF** configurations are simulated: **PF-2x8w** consists of 2 sequencers, 8-wide each; and **PF-4x4w** consists of 4 sequencers, 4-wide each. The aggregate width of the front end is 16 in each case. Each individual sequencer is identical to **W16** except for its width. The aggregate size of the

1.This combination of an instruction cache and a trace cache performs better than allocating the entire L1 cache space to a trace cache.

Table 2: Benchmark Characteristics

Benchmark	Input	Avg Frag Size (instructions)
bzip2	test	12.79
crafty	test	11.99
eon	train (cook)	10.98
gap	test	10.69
gcc	test	11.15
gzip	test	12.06
mcf	train	9.04
parser	test	10.35
perl	train (diffmail)	11.32
twolf	train	12.16
vortex	test	11.20
vpr	train (place)	12.33

L1 cache is the same as in **W16** (64 KB), and the cache is split into 16 banks. The fragment predictor and fragment selection heuristics are identical to **TC** to allow an unbiased comparison.

Finally, **PR** denotes the **PF** mechanism coupled with the parallel renaming mechanism described in Section 4. **PR-2x8w** denotes 2 sequencers, 8-wide each, coupled with 2 renamers, also 8-wide each; similarly, **PR-4x4w** denotes 4 sequencers, 4-wide each, coupled with 4 renamers, 4-wide each. The live-out predictor contains 4K entries, and is 2-way set associative. Perfect fragment length prediction is assumed.

PF and **PR** have a slight storage advantage over **W16** and **TC** since the total L1 storage available to them is the size of the cache plus the size of the fragment buffers ($16 \times 16 \times 4 = 1$ KB). We decided not to correct this discrepancy since it would require simulating a 63 KB L1 cache—which is neither practical for simulation, nor meaningful for a real machine. We do not expect this to skew the conclusions significantly since the fragment buffers increase the total available L1 storage by only 1.6%.

All benchmarks were taken from the SPEC CPU 2000 benchmark suite and were compiled with ‘peak’ settings using the Compaq Alpha compiler. We report results only for the twelve integer benchmarks. Floating point benchmarks were omitted since they are either memory limited or have very simple control flow, with the result that all front ends perform equivalently on them. Excluding them prevents dilution of differences between the schemes.

All programs were simulated for the first one billion instructions. Test inputs were used, except for the benchmarks `eon`, `mcf`, `perl`, `twolf`, and `vpr`, since their test run was shorter than a billion instructions—train inputs were used for these. Table 2 lists the benchmarks, the inputs used, and the average fragment length.

The results are organized as follows: in Section 5.1 we study the efficiency with which various fetch mechanisms utilize the available cache bandwidth, and the net throughput they achieve. Following that, we study our parallel renaming mechanism: Sections 5.2 and 5.3 evaluate the parallel rename unit and the live-out predictor respectively. Section 5.4 compares the overall performance of various schemes. Finally, Section 5.5 and Section 5.6 study the sensitivity of performance to the amount of L1 instruction storage and the trace/fragment predictor size.

5.1 Parallel Fetch

All three fetch mechanisms—**W16**, **TC**, and **PF**—have identical maximum throughput: each can fetch at most 16 instructions every cycle. However, each fetch mechanism has different limitations on how efficiently this available bandwidth can be utilized. The number of instructions fetched per cycle is not an accurate measure of bandwidth utilization since, in addition to being affected by the fetch unit, it is also affected by the IPC of the program. At times of low IPC the fetch unit will be stalled, and thus unable to utilize the available bandwidth. Therefore, measuring the instructions fetched per cycle may hide limitations of the fetch unit.

The ratio of the number of instructions fetched to the number of cycles in which the fetch unit was not stalled is not accurate either: a parallel fetch unit may be partially stalled sometimes (i.e., some sequencers may be stalled, but not all), and therefore this metric discriminates against a parallel fetch unit.

We use the notion of *fetch slots* to abstract away the back-end from the fetch unit. Each cycle that a sequencer is active, there is a potential maximum number of instructions it can fetch. That is the total number of fetch slots. Thus **W16** and **TC** have either 0 or 16 fetch slots every cycle, whereas **PF** has a varying number of fetch slots depending on the number of sequencers that are active. To measure the efficiency of each fetch mechanism, we determine the ratio of the number of instructions fetched to the total number of fetch slots, i.e., the *fetch slot utilization*.

Figure 4 shows the fetch slot utilization of different fetch mechanisms. Each bar in the graph represents the harmonic mean across all benchmarks. As expected, **W16** does not perform well. It is able to utilize only 40% of the available slots. A trace cache increases the average utilization to about 60%—a little lower than the ratio between average and maximum trace size (Table 2). **PF-2x8w** achieves about 70% utilization on average—17% more than **TC** and **TC$_{2x}$**. **PF-4x4w** further increases utilization, since narrower sequencers lead to fewer wasted slots, achieving 80% utilization on average.

Figure 5 shows the average number of instructions fetched and renamed every cycle by each mechanism,

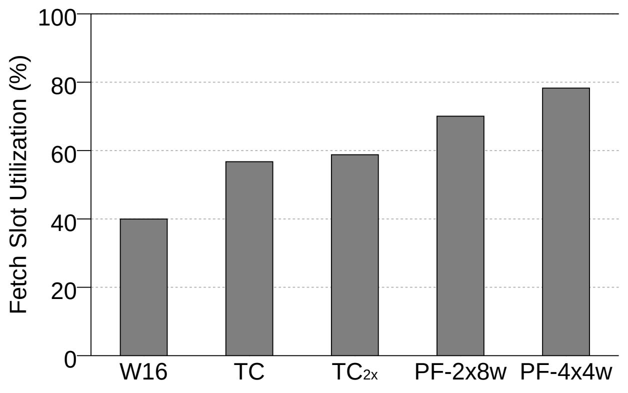

Figure 4. Fetch Slot Utilization

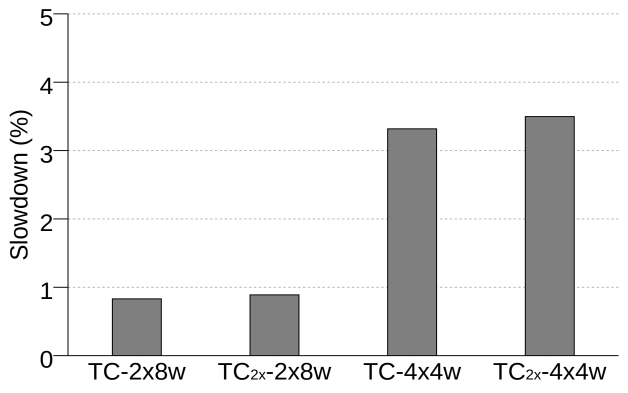

Figure 6. Monolithic versus Parallel Renaming

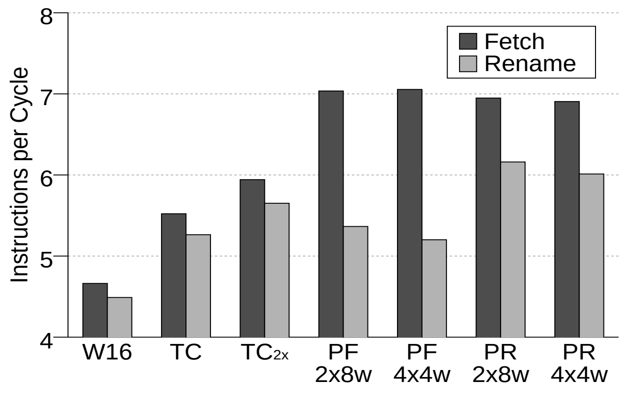

Figure 5. Fetch and Rename Throughput

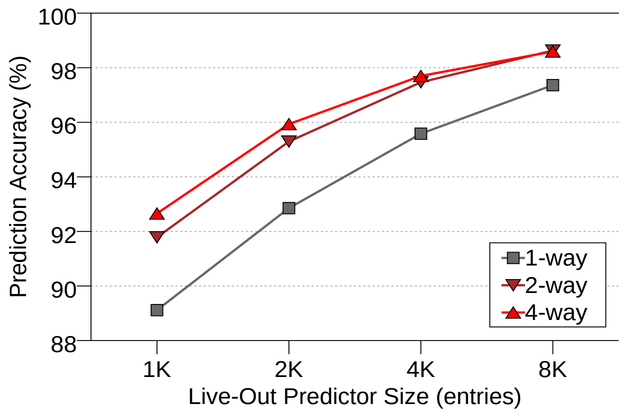

Figure 7. Live-Out Prediction Accuracy

including wrong-path instructions. **PF** is able to sustain an average fetch rate of 7 instructions per cycle—about 20% higher than the trace cache, and 49% higher than **W16**.

Of course, high fetch efficiency and/or rate may not directly translate to higher performance since the fetch unit may be fetching many wrong-path instructions. However, it is important to realize that there are two problems in instruction fetch: (1) *what to fetch*, and (2) *how to fetch*. Parallel fetch is aimed at the second problem, and the above results show that it successfully achieves this aim.

5.2 Parallel Renaming

As discussed in Section 3.4, the high fetch rate of **PF** does not necessarily lead to a high rename rate, since the instruction stream could be serialized at the rename unit. However, the rename rate has a more direct impact on performance than the fetch rate. High IPC can be obtained only if the back-end has enough ready instructions, and increasing the rename rate directly increases the number of instructions available for execution.

The light gray bars in Figure 5 show the average number of instructions renamed each cycle by the mechanisms under study. For sequential fetch mechanisms, the rename rate is similar to the fetch rate; a little lower, since on branch mispredictions some fetched instructions are discarded before they reach the rename stage. However, the rename rate of **PF** is much lower than its fetch rate, indicating that serializing the instruction stream at the rename stage severely impacts the front-end throughput.

PR increases the rename rate of **PF** by 13% on average. However, there is still a significant gap between the fetch rate and the rename rate of **PR** that is larger than the corresponding gap for **W16** and **TC**. This gap exists because the number of instructions discarded due to mispredictions by a parallel fetch unit is higher than by sequential fetch schemes. A parallel fetch unit buffers many more instructions in the fetch stage, and is required to predict control flow much further into the future.

As described in Section 4.4, a sequential fetch unit can be combined with a parallel renaming unit. Note that parallel renaming, while adding performance to parallel fetch, is not a performance enhancing technique by itself. It may

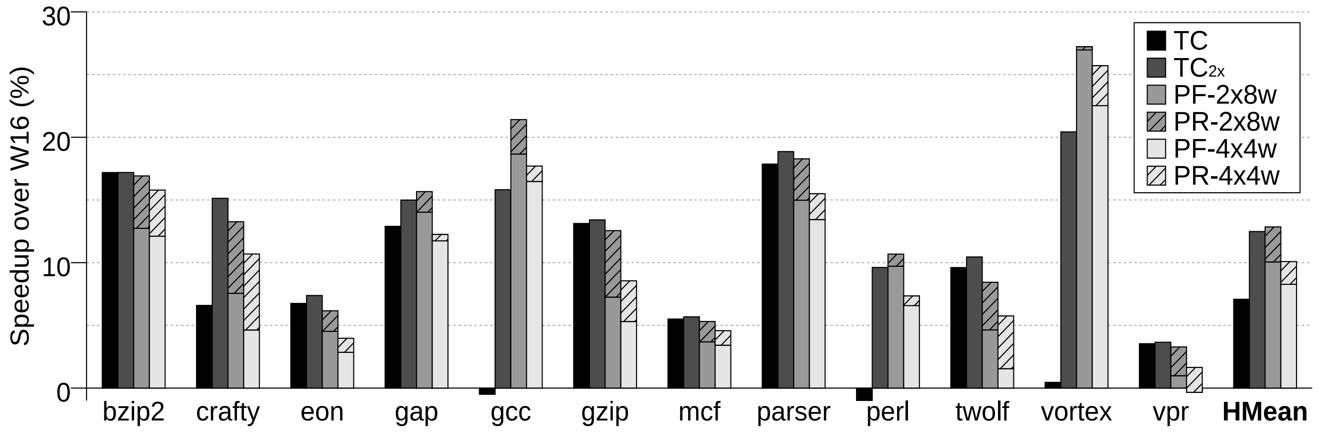

Figure 8. Performance

simplify rename, however, and thus enable faster circuit implementations. Figure 6 shows the performance penalty of using a parallel renaming unit with a trace cache fetch mechanism. Two parallel renamers are studied: (1) 2x8w—two 8-wide renamers operating in parallel, and (2) 4x4w—four 4-wide renamers operating in parallel.

A 2x8w renaming unit performs within 1% of a monolithic renaming unit on average. A 4x4w renamer suffers a higher penalty of about 3.5%. Only one-third of this slowdown is due to live-out mispredictions; the rest is caused by instructions being renamed before their sources. Our simulations indicate that 4–12% of dynamic instructions are renamed before the instructions producing the corresponding sources when a 4x4w renamer is used.

5.3 Live-Out Predictor

Figure 7 shows the prediction accuracy of the live-out predictor for a range of sizes and associativities. The predictor is clearly space-limited and benefits substantially as the number of entries is increased. Increasing the associativity to two increases accuracy, but a further increase does not help much. In this paper, we use a 2-way 4K entry predictor (42 KB) which achieves 98% accuracy on average.

The number of live-outs per fragment is typically small (4–6 registers), so a more complex encoding could significantly reduce the storage requirements of the predictor.

5.4 Overall Performance

Figure 8 shows the performance of different front-ends over all benchmarks. The Y-axis indicates the percent speedup over **W16**. The four bars in each cluster represent **TC**, **TC_{2x}**, **PR-2x8w**, and **PR-4x4w** respectively. The lower section of last two bars indicates the performance of the corresponding parallel fetch configuration, and the upper section shows the benefit due to parallel renaming.

As noted earlier, **TC_{2x}** is identical to **TC**, except that total L1 instruction storage is doubled from 64K to 128K. The difference between the **TC** and **TC_{2x}** bars is therefore the benefit due to a larger cache. This difference is small in most cases, indicating that the working sets of most benchmarks fit in 64KB of L1 cache space. **PR-2x8w** performs within 2% of both **TC** and **TC_{2x}** on these benchmarks. On the four benchmarks that gain significantly from doubling the L1 cache (`crafty`, `gcc`, `perl`, and `vortex`), **PR-2x8w** performs 10–20% better than **TC**.

On average, **PR-2x8w** performs equivalently to **TC_{2x}** with just half the cache space and 5% better than **TC** with a similar amount of space. **PR-4x4w** performs 3% better than **TC** on average but a little worse than **TC_{2x}**. Out-of-order renaming increases performance of the parallel fetch unit by 0–6% depending on the benchmark. These results represent a 10–13% average speedup over the base **W16** configuration, indicating the importance of a high performance fetch mechanism when using an aggressive back-end.

PR-4x4w performs 3% worse than **PR-2x8w** on average since it looks further into the future, and thus is more likely to fetch down mispredicted paths. In addition, it takes longer to recover from mispredictions since it takes at least four cycles for all four sequencers to become active, rather than two cycles in the case of **PR-2x8w**. Finally, as described in Section 5.2, greater parallelism in the renaming stage causes instructions to be renamed in suboptimal order. Thus, better control prediction and more intelligent parallel renaming would be necessary to achieve the advantage of four sequencers over two.

5.5 Sensitivity to Cache Size

Results presented in the Section 5.4 already indicate that a parallel fetch unit is more suitable than a trace cache for workloads with large code footprints. In this section,

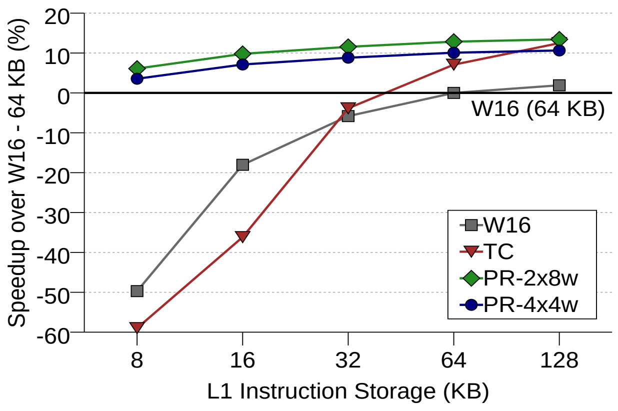

Figure 9. Sensitivity to Cache Size

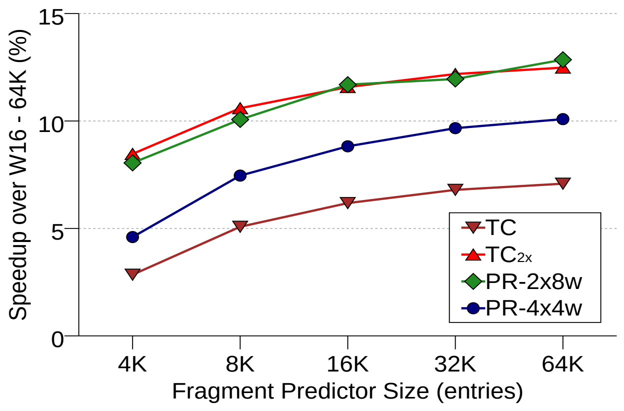

Figure 10. Sensitivity to Fragment Predictor Size

we take this further and show that a parallel fetch unit provides robust performance over a wide range of L1 instruction cache miss rates.

Figure 9 shows the performance of the fetch mechanisms under study over a range of cache sizes, and thus a range of cache miss rates. The X-axis indicates the total L1 instruction storage (instruction cache + trace cache) and the Y-axis indicates speedup over **W16** with a 64 KB cache. Since **TC** and **TC$_{2x}$** differ only in cache size, **TC$_{2x}$** is not shown on the graph.

The line representing **TC** has the highest slope of all, indicating that a trace cache loses performance most rapidly as the number of cache misses increase. On the other hand, **PR-2x8w** and **PR-4x4w** lose only about 6% performance as the cache size is reduced by a factor of sixteen from 128 KB to 8 KB. For small cache sizes, sequential fetch mechanisms are 50–62% slower than **PR**. **PF**, not shown on the graph to reduce clutter, has slope similar to **PR**, but slightly lower performance.

Two factors contribute to this resilience: (1) the ability to continue fetching and executing instructions beyond a cache miss, and (2) the ability to overlap multiple cache misses with each other. We expect that tolerance to cache miss rates will become increasingly important in the future as technology constraints make it harder to design large structures, and as program sizes become larger. Small caches are also attractive since they can be clocked faster, and parallel fetch allows the cache size to be reduced with little impact on performance.

5.6 Sensitivity to Trace/Fragment Predictor Size

Figure 10 shows the sensitivity of performance to trace/ fragment predictor size. The X-axis indicates the number of entries in the primary table; the number of entries in the secondary table is one fourth of that in all cases[1]. The Y-axis indicates the speedup over **W16** with a default sized predictor (64K entries).

We see that all mechanisms gain about 1.25% performance on average when the predictor size is doubled. For a trace cache, this is a significantly smaller increase as compared to the benefit of doubling the cache size. For parallel fetch, however, this is similar to the benefit from doubling the instruction cache size. This suggests that a parallel fetch mechanism should have a large predictor, even at the expense of a smaller instruction cache, since doubling the predictor may be less expensive in terms of chip area than doubling the cache.

6 Conclusions

Sequential front-ends are limited in the throughput they can achieve since they are designed to fetch instructions from contiguous memory locations, but the control flow structure of many programs cannot be mapped onto a sequential storage order. Therefore, we propose that fetch throughput be increased not by widening sequential front-ends, but by building parallel front-ends—front-ends composed of multiple sequential fetch and rename units operating in parallel. We described a possible implementation of such a parallel front-end and qualitatively discussed its characteristics. In particular, we discussed why a parallel fetch unit cannot be used effectively unless coupled with a parallel rename unit, and described ways in which a parallel rename unit could be built.

We found that a parallel front-end is able to achieve higher throughput than a trace cache, and in most cases better or equivalent overall performance as well. As a result of being parallel, the proposed front end is able to tolerate cache miss latency better than sequential front ends, and thus provide good performance even on programs with a high L1 instruction cache miss rate.

1.The reader is referred to prior work [11] for a detailed description of the trace/fragment predictor.

Since the objective of this paper was to compare a parallel front-end with a high-performance sequential front-end, we chose to make fragments identical to traces. This enabled us to directly compare our scheme to a trace cache. However, this mechanism has fewer restrictions on fragment selection than a trace cache has on trace selection. Fragments can be longer and can have a larger variance in size without affecting cache storage efficiency. They can contain intra-fragment control flow, unlike traces. Further research on fragment selection and prediction is necessary to fully exploit the potential of parallel front-ends.

7 Acknowledgements

We would like to thank Adam Butts, Philip Wells, Manoj Plakal, and Allison Holloway for commenting on drafts of this paper. The comments of the anonymous reviewers have also helped improve the quality of this paper. This work was supported in part by National Science Foundation grants CCR-9900584 and EIA-0071924, donations from Intel and Sun Microsystems, and the University of Wisconsin Graduate School.

8 References

[1] V. Bala, E. Duesterwald, and S. Banerjia. Transparent Dynamic Optimization. Technical Report HPL-1999-77, Hewlett Packard Labs, June 1999.

[2] T. Ball and J. R. Larus. Branch Prediction For Free. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 300–313, Albuquerque, New Mexico, June 23–25, 1993.

[3] S. Breach. *Design and Evaluation of a Multiscalar Processor.* Ph.D. thesis, University of Wisconsin-Madison, 1998.

[4] D. C. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin-Madison, Jun. 1997.

[5] B. Calder and D. Grunwald. Reducing Branch Costs via Branch Alignment. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 242–251, San Jose, California, October 4–7, 1994.

[6] 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*, Austin, Texas, Dec. 2–5, 2001.

[7] T. M. Conte, K. N. Menezes, P. M. Mills, and B. A. Patel. Optimization of Instruction Fetch Mechanisms for High Issue Rates. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 333–344, Santa Margherita Ligure, Italy, June 22–24, 1995.

[8] J. Emer. EV8: The Post–Ultimate Alpha. Keynote Address, 10th International Conference on Parallel Architectures and Compilation Techniques, 2001.

[9] M. Franklin and M. Smotherman. A Fill-Unit Approach to Multiple Instruction Issue. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 162–171, November 30–December 2, 1994.

[10] D. H. Friendly, S. J. Patel, and Y. N. Patt. Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 173–181, Dallas, Texas, November 30–December 2, 1998.

[11] Q. Jacobson, E. Rotenberg, and J. E. Smith. Path-Based Next Trace Prediction. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 14–23, Dec. 1–3, 1997.

[12] R. Muth, S. Debray, S. Watterson, and K. de Bosschere. ALTO: A Link-Time Optimizer for the DEC Alpha. Technical Report TR98-14, University of Arizona, September 1998.

[13] P. S. Oberoi and G. S. Sohi. Out-of-Order Instruction Fetch using Multiple Sequencers. In *Proceedings of the 2002 International Conference on Parallel Processing*, pages 14–23, Vancouver, Canada, August 18–21, 2002.

[14] S. J. Patel, D. H. Friendly, and Y. N. Patt. Critical Issues Regarding the Trace Cache Fetch Mechanism. Technical Report CSE-TR-335-97, Department of Electrical Engineering and Computer Science, University of Michigan, May 1997.

[15] 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 33rd Annual International Symposium on Microarchitecture*, pages 303–313, Monterey, California, December 10–13, 2000.

[16] A. Peleg and U. Weiser. Dynamic Flow Instruction Cache Memory Organized Around Trace Segments Independent of Virtual Address Line. US Patent 5,381,533, March 30, 1994.

[17] M. Postiff, G. Tyson, and T. Mudge. Performance Limits of Trace Caches. *Journal of Instruction-Level Parallelism*, 1, August 1998.

[18] A. Ramirez, J-L. Larriba-Pey, C. Navarro, J. Torrellas, and M. Valero. Software Trace Cache. In *Proceedings of the 1999 international conference on Supercomputing*, pages 119–126, Rhodes, Greece, 1999.

[19] A. Ramirez, O. J. Santana, J. L. Larriba-Pey, and M. Valero. Fetching Instruction Streams. In *Proceedings of the 35rd Annual International Symposium on Microarchitecture*, Istanbul, Turkey, November 18–22, 2002.

[20] E. Rotenberg, S. Bennett, and J. E. Smith. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 24–34, Paris, France, Dec. 2–4, 1996.

[21] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *Proc. 22nd International Symposium on Computer Architecture*, pages 414–425, Jun. 1995.

[22] 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, Dec. 1–3, 1997.