Out-of-Order Front-Ends

Paramjit S. Oberoi
Computer Sciences Department
University of Wisconsin–Madison
param@cs.wisc.edu

Preliminary Proposal

1 Introduction

Achieving high performance requires processor architectures capable of exposing and exploiting large amounts of instruction-level parallelism (ILP), while allowing a high clock rate. Over the last few decades, processors back-ends have become increasingly successful at exploiting ILP (especially by using out-of-order processing techniques) and higher clock rates have been achieved by using decoupled and/or clustered designs. However, front-ends continue to be both in-order and monolithic. I propose to investigate high-performance front-ends suitable for future processors that employ the same techniques that have been successful in the back-end: out-of-order processing, decoupled operation, and clustering.

I begin by discussing the challenges in building a high-performance front-end and describe why existing solutions are unsuitable for future processors (Section 1.1 and Section 1.2). Following that, I discuss the proposed solution: out-of-order front-ends (Section 1.3), and present an outline of the proposed research (Section 1.4).

1.1 High-performance Front Ends

The difficulty in fetching instructions at a high rate arises from the fact that it is impossible to map the control flow of a complex program onto a sequential storage order. For this reason, most programs contain frequent branch instructions, and a significant fraction of the branches result in transfers of control from one part of the program to another. Therefore, fetching a large block of instructions contiguous in the dynamic instruction stream involves fetching from different areas of memory, which is difficult in a single cycle.

In addition, simple fetch units work by fetching a single block from the instruction cache, and then extracting the required instructions from it. Their maximum throughput is therefore limited by two factors: (1) distance between taken branches, and (2) number of useful instructions in a cache line. Taken branches and cache line boundaries introduce *discontinuities* in the instruction stream since instructions cannot easily be fetched across these boundaries in the same cycle. Typical programs contain one taken branch every eight instructions on average [13, 14, 44]. The number of instructions between cache-line boundaries is similar: L1 instruction cache lines contain 16 instructions on the Alpha 21264 [18]; therefore, a cache-line boundary will be encountered every eight instructions on average.

A front-end must not be limited by these discontinuities if it is to achieve high throughput. Various techniques have been proposed to transcend this limit. These techniques can be classified into two groups: restructuring instruction storage to remove discontinuities, and restructuring the fetch mechanism to handle discontinuities.

1.1.1 Removing Discontinuities

One way of handling discontinuities is to simply remove them, i.e., to ensure that the dynamic instruction stream is laid out sequentially in memory and covers as few cache lines as possible [2, 6, 14, 19, 26, 28, 29]. This rearrangement of instructions can be carried out in a variety of ways, both at compile time and run time, and possibly, although not necessarily, involving modification of the static program binary.

The most straightforward implementation of this approach is a modification to the code generation and linking phases of the compiler. The compiler determines probable directions of all conditional branches (using heuristics, profile information, or programmer hints), and then attempts to lay out code to minimize the occurrence of taken branches. In addition, some unconditional branches and function calls can be avoided by duplicating code, and unpredictable conditional branches can be eliminated using predication. Information about cache sizes and associativities can be used to partially realize the benefits of hardware mechanisms like a trace cache [29]. These techniques can also be implemented in an offline binary rewriting tool [20]. The static optimization approach is beneficial to some extent, but not effective enough to solve the front-end throughput problem by itself [14, 29, 44].

The effectiveness of code reordering optimizations can be improved by carrying them out at run-time, with the additional information available on the actual program behavior, and with the possibility of tailoring the optimization to the phase behavior of the program. This can be achieved using both software and hardware techniques.

A trace cache [23, 25, 34] is an example of a transparent micro-architectural technique for optimizing code layout dynamically at run time. A trace cache observes the dynamic execution of the program, and caches instructions in their dynamic execution order. Accessing a single entry in a trace cache returns a sequence of instructions that may be discontiguous in the static program. Moreover, as the program behavior changes, the trace cache adapts by evicting unused traces and constructing more relevant traces. However, this approach makes inefficient use of storage resources available to it due to fragmentation and duplication [27, 30]. Efficient use of cache space is important for future designs since trends indicate that (1) program working sets are becoming larger, and (2) technology constraints are making it harder to build large on-chip structures. Techniques have been proposed to improve the cache space utilization of the trace cache [3, 30]. However, although these techniques reduce replication to some degree, fragmentation still remains a problem.

Code layout optimization can also be performed dynamically by software, in a virtual machine, for example [1, 10]. However, software techniques inevitably lead to higher overhead which can diminish the advantages of the optimizations performed. In addition, just like a trace cache, such techniques do not work well for programs with large code sizes.

1.1.2 Fetching Across Discontinuities

The other option is to design a fetch mechanism capable of fetching instructions across instruction stream discontinuities in the same cycle. This typically requires a banked instruction cache so that multiple cache blocks can be fetched concurrently. The collapsing buffer [9] is such a mechanism.

However, fetching across discontinuities tends not to perform very well since there are restrictions on the sets of cache blocks that can be fetched concurrently. These restrictions arise because (1) circuit complexity limits the num-

ber of different cache blocks that can be fetched and combined in a single cycle, and (2) bank conflicts limit which blocks may be fetched concurrently. The limit on the maximum number of blocks limits the maximum throughput, and even if fetching a large number of blocks is made possible, the larger the number of blocks, the more the chance of a conflict that will restrict the actual number of blocks fetched.

High-performance discontinuity-removing techniques like the trace cache typically outperform collapsing buffers, especially when sufficient instruction storage is available [34].

1.2 Sequential Front-Ends

It is this sequential nature of fetch mechanisms that leads to the requirement of fetching long *contiguous* blocks of instructions, and therefore makes discontinuities in the instruction stream performance limiters. Sequential fetch also has other consequences that make it undesirable. These are discussed below.

As processors get more aggressive, they need larger windows of dynamic instructions so that sufficient ILP is exposed. It is not inconceivable that processors would require windows of several hundred instructions in a few years. In some situations, it may be desirable to execute a few critical instructions towards the end of the instruction window before any of the prior instructions (for example, if the later instruction is a load or a branch). A sequential front-end would require fetching and renaming all prior instructions—possibly *hundreds* of prior instructions—before those critical instructions were available for execution.

Sequential components are also susceptible to stalls since any operation that cannot be completed quickly holds up all future operations. In the case of a front-end, instruction cache misses, TLB misses, and bank conflicts are examples of events that may unnecessarily delay future instructions. Techniques have been proposed to alleviate some of these problems [7, 32, 33, 36, 45], but it may be better to attack the root cause of the problem—sequential operation.

Sequential fetch may also be an inefficient way of providing instructions to an out-of-order back-end. Consider the trace cache: it is capable of fetching contiguous blocks of over ten instructions every cycle. However, it turns out that an out-of-order back-end executes instructions in consecutive traces more-or-less consecutively [21]. A fetch mechanism capable of interleaving fetch of instructions from different traces may be able to achieve performance equivalent to the trace cache despite a lower fetch rate, by choosing what to fetch more judiciously.

A limited out-of-order fetch technique [39, 40] has been proposed that alleviates the problem of stalls due to cache misses. It is able to successfully overlap cache miss latencies with useful future work. However, it focusses specifically on the problem of cache misses, and thus does not resolve the other two problems mentioned above—performance limitations due to discontinuities, and flexibility. The techniques I propose to study are likely to be significantly more powerful: the aim is to develop general out-of-order fetch and rename mechanisms without concentrating on any specific limitation of sequential front-ends. Such an approach is more likely to lead to front-end designs that solve a wider range of problems in a simpler as well as more effective fashion—rather than designs that are agglomerations of various different techniques, and thus are likely to be both more complex as well as less generally applicable.

1.3 Out-of-Order Front-Ends

There is nothing inherent in the process of fetching instructions that requires it to be strictly sequential—the same instructions are obtained regardless of the order in which they are fetched. Other parts of the front-end, namely control prediction and register renaming, do have the conceptual model of serial processing. Of course, instruction execution also has sequential semantics—but we have been able to relax the serial processing requirement for execution. So it so is not unreasonable to expect that we may be able to relax it for the front-end as well.

Enabling the front-end to process instructions out-of-order, i.e., in an order different from program order, makes it possible to leave behind the limitations of sequential front-ends. First, discontinuities no longer limit performance since the front-end is explicitly designed to fetch instructions non-sequentially. Moreover, the challenges facing a sequential front end that attempts to fetch across discontinuities—complexity and bank conflicts—become easier to deal with since (1) the fetch unit is no longer required to 'select' and 'collapse' instructions from multiple cache lines at once to produce a sequential instruction stream, and (2) bank conflicts can be reduced by changing the order in which instructions are fetched.

An out-of-order front-end is also more latency tolerant. Just as an out-of-order back-end can overlap long latency operations with other independent instructions, an out-of-order front-end can, for instance, overlap an instruction cache miss with the fetch of other instructions that may be in the cache, or overlap multiple instruction cache misses with each other.

Finally, an out-of-order front-end enables optimization of the order in which instructions are fetched. This can be used to achieve higher performance, for example by fetching critical instructions first; alternatively, it can allow enhancement of some other parameter like resource utilization or energy consumption.

1.4 Outline

In my dissertation, I propose to describe and characterize two different out-of-order front-ends:

1. **Parallel Fetch and Rename:** A front-end composed of several decoupled sequential front-ends operating in parallel. Out-of-order fetch is achieved by dividing the instruction stream into fragments and fetching multiple fragments in parallel. Both fetch and rename are parallelized. Research issues include fragment selection & prediction, fetch interleaving heuristics, out-of-order renaming techniques, and a detailed design space exploration.

2. **Block-Based Out-of-Order Fetch:** A front-end modelled along the lines of an out-of-order execution window. The fetch unit maintains a 'fetch window' comprising of a list of cache block addresses, and a scheduler selects a subset every cycle for fetching from the cache. This design allows greater flexibility than parallel fetch and rename, but may be more complex. The additional flexibility can be used to actively optimize the order in which instructions are processed for improving various characteristics of the processor (performance, resource consumption, etc.).

The rest of this document is organized as follows. Section 2 describes a parallel fetch mechanism, and Section 3 describes an associated parallel renaming mechanism. Section 4 presents a block-based out-of-order front-end.

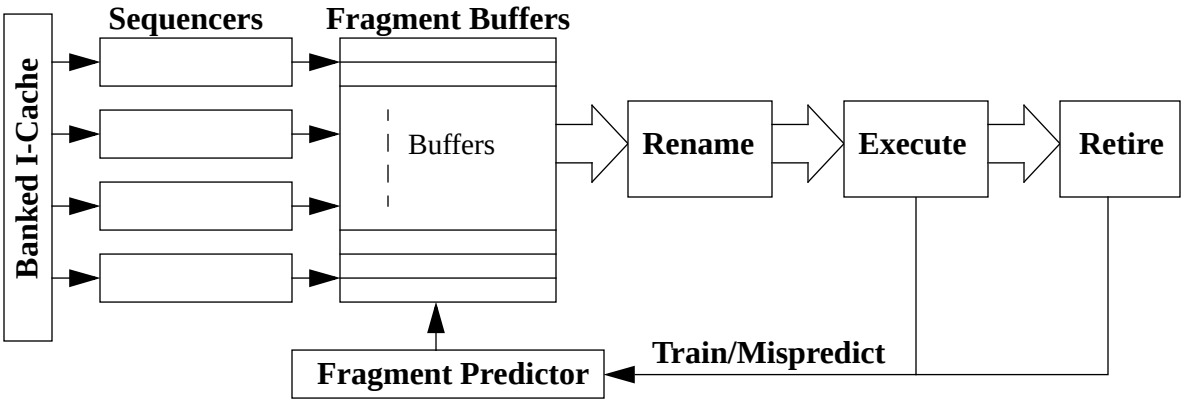

Figure 1. Parallel Fetch Unit

Section 5 discusses some possible heuristics for reordering fetch, assuming a block-based front-end. Finally, Section 6 concludes with a summary and a schedule.

2 Parallel Fetch

A straightforward way of achieving out-of-order fetch behavior is to parallelize fetch. The instruction stream is divided into fragments, and multiple fragments are fetched in parallel, each by a sequential fetch unit. Each fragment is simply an arbitrary contiguous portion of the dynamic instruction stream (fragments are discussed in more detail in Section 2.1.2). Each individual fragment is fetched in program order, but fetch of different fragments is decoupled so that the order in which instructions are fetched can vary. Instructions are first fetched into a "staging area" and then subsequently reassembled into the original sequential stream. The objective of a parallel fetch unit is to achieve high fetch throughput by fetching multiple (possibly discontiguous) instruction blocks in parallel, rather than fetching long contiguous instructions blocks.

This is similar to instruction fetch in Multiscalar [4, 37]: 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, out-of-order fetch in Multiscalar is an artifact of a fully clustered microarchitecture. The technique proposed here is completely general and makes no assumptions about the back-end.

Parallel fetch addresses the challenges outlined in the introduction as follows: (1) Discontinuities: Since throughput is achieved by fetching multiple blocks in parallel, individual blocks can be small—and therefore discontinuities are no longer a limiting factor. (2) Stalls and Bank Conflicts: Fetch of different fragments is decoupled from each other—thus a delay in the fetch of one fragment does not delay fetch of other fragments. In addition, many bank conflicts can be eliminated by reordering fetch.

2.1 Design Overview

I use the term *sequencer* to denote a mechanism that sequences through instructions in program order. A sequential fetch unit contains a single sequencer. Figure 1 illustrates a parallel fetch unit composed of multiple sequencers. The sequencers write the fetched instructions into an array of fragment buffers instead of directly forwarding them to

later stages in the pipeline. 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 sequencers fetch multiple fragments into the corresponding fragment buffers in parallel. The instruction cache is banked so that it can handle multiple (non-conflicting) requests simultaneously.

The net throughput of a this 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 instruction cache misses and bank conflicts.

2.1.1 Fragment Buffers

Fragment buffers are FIFO queues of instructions. In addition to instructions, they store other information relating to the fragment like its starting address, the current PC, and branch prediction information. 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. 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. Once all instructions have been read from a buffer by the next pipeline stage, the buffer is marked invalid so that it can be reused by subsequent fragments.

Various heuristics can be used to decide which predicted fragments to fetch in any cycle: oldest first, round robin, etc. Our first implementation of parallel fetch [21] used a combination of the two: the oldest fragment was always given first priority, but the others were serviced in a round-robin fashion (the current implementation uses oldest-first). Another design decision is the size of the FIFO queues: it may be smaller than the longest possible fragment, in which case it can be managed as a circular buffer. My dissertation will evaluate these choices in detail.

Another potential optimization is to not discard instructions in fragment buffers after they have been read; instead, the buffer is marked unused, and if the same fragment is encountered again before the buffer has been reallocated, the instructions are reused instead of being fetched again from the instruction cache. This makes the fragment buffers behave like a very small trace cache. 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. Preliminary results indicate that reuse has a small impact on performance, but a large impact on cache traffic; I plan on exploring the reuse/fetch trade-off in detail in the future. Fragment reuse also depends upon buffer allocation strategies. An LRU allocation scheme is used currently, but other methods may be more appropriate since the number of fragment buffers is much smaller than a typical cache.

2.1.2 Fragment selection

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* [34] or *tasks* [37], except

Table 1: Trace Characteristics of SPEC INT 2000 Benchmarks

Benchmark	Input	Number of Static Fragments	Average Fragment Size	Static Fragments responsible for 95% dynamic instructions
bzip2	test	1129	12.72	74
crafty	test	6846	12.14	939
eon	train (cook)	4565	10.88	317
gap	test	6491	10.76	770
gcc	test	35994	11.24	7126
gzip	test	1769	12.00	50
mcf	train	1032	9.66	143
parser	test	5578	10.16	563
perl	train (diffmail)	9743	11.62	636
twolf	train	4009	10.91	426
vortex	test	6400	11.22	633
vpr	train (place)	4093	11.98	298

that fragments are completely general whereas the other terms make assumptions about the nature of fragments or about how they are processed.

Good fragment selection involves balancing several contradictory requirements. First, the fragments must be reasonably long. At the same time, they should be terminated at the end of control structures like loops and functions to increase the prediction accuracy and decrease the number of overlapping fragments. A balance must be struck between reasonable fragment length and the amount of overlap between different fragments.

The fragment selection criteria used in the current implementation are as follows: fragments are terminated at all indirect branches (including calls and returns), at any unconditional branch after the eighth instruction, or at the sixteenth instruction. Table 2 lists characteristics of the fragments obtained.

These heuristics work well for both a parallel fetch unit as well as a trace cache, and therefore they can be used to make an unbiased comparison between the two schemes. However, a parallel fetch unit can be more flexible in its fragment selection than a trace cache: it can tolerate much larger variance in fragment sizes, and a larger overlap between fragments without wasting cache space. Fragments could also contain internal control flow, similar to Multiscalar tasks. Fragment selection could also be improved by taking the predictability of branches into account [15, 24]. Using unpredictable branches as fragment boundaries would make intra-fragment control flow predictable, enhancing the potential for fragment reuse. On the other hand, if fragments were allowed to have internal control flow, it may be beneficial to encapsulate unpredictable "hammocks" within fragments so that the effect of mispredictions could be limited to a single sequencer.

It is likely that good fragment selection is critical to obtaining good performance with this mechanism, and therefore I plan to explore fragment selection issues in detail as a part of my research.

2.1.3 Fragment prediction

Conceptually, a fragment predictor only needs to predict fragment boundaries. Intra-fragment control flow can be predicted by each sequencer using a local mechanism. Fragment prediction is a very similar problem to trace prediction, and path-based trace predictors [16] based on the Multiscalar DOLC [4] predictor can be used as fragment pre-

dictors as well. Path-based prediction has been shown to be effective in a variety of control prediction domains, viz. task prediction [4], trace prediction [16], stream prediction [31], and therefore it can be expected to be effective at fragment prediction as well. An additional advantage is that a path-based predictor predicts branch directions of all branches in a fragment as well, and therefore local branch predictors are not required at each sequencer to predict intra-fragment branches. However, if fragments are allowed to have internal control flow, then it may be useful to consider separating inter-fragment and intra-fragment prediction.

The trace predictor proposed by Jacobson, Rotenberg, and Smith [16] is used in the studies presented here. Since this predictor predicts addresses as well as branch directions for all branches in the trace, local branch predictors are not used. For the benchmarks studied, the average branch prediction accuracy is 95%. In the future, I plan on experimenting with other prediction mechanisms if required by the fragment selection methods that I explore.

2.1.4 Scalability

One attraction of using a clustered fetch unit is that it may be easy to increase the net throughput of the fetch unit by simply adding more sequencers to it. Scalability of the design proposed above is limited by the following issues:

1. **Prediction Accuracy:** More sequencers are useful only if the control predictor accurately predicts enough future traces to keep all sequencers busy.
2. **Prediction Bandwidth:** The average throughput of this design can never exceed one fragment per cycle since the fragment predictor predicts at most one fragment per cycle. This limitation could possibly be alleviated by making the predictor return multiple predictions per lookup.
3. **Reading Instructions out of Fragment Buffers:** The current design assumes that instructions from at most one fragment can be read out of the fragment buffers in one cycle. This limits the throughput to one fragment per cycle. Relaxing this constraint may make the associated circuits more complex.

It may also be possible to sidestep these problems by using a fragment selection technique that results in much longer fragments.

2.2 Preliminary Results

The parallel front-end proposed here was compared against two other front-ends: simple 16-wide sequential fetch, and a trace cache. These two were chosen since the first is the most commonly used mechanism, and the second is widely considered the state of the art in front-ends.

The conventional sequential 16-wide front-end is referred to as **W16** hereafter. It 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 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. `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.

Table 2: 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 Predictor	DOLC [16], 64K entry primary table, 16K entry secondary table (D=9, O=4, L=7, C=9)

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 (this combination of an instruction cache and a trace cache performs better than allocating the entire L1 cache space to a trace cache). Two trace cache configurations are simulated: (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.

Parallel fetch is denoted by the prefix **PF**: **PF-2x8w** consists of 2 sequencers, 8-wide each; and **PF-4x4w** consists of 4 sequencers, 4-wide each. Each individual sequencer is identical to **W16** except for its width. The aggregate width of the front end is 16 for all configurations being compared. The parallel fetch unit contains 16 fragment buffers of 16 instructions each. The parallel front-end has 1KB of additional storage due to the fragment buffers (16×16×4 bytes). The size of the instruction cache was not adjusted to account for this since that would have involved simulating a 63KB cache—which is neither meaningful for a real machine, nor practical for simulation. The additional storage represents an increase of only 1.6% over a 64KB instruction cache.

The results presented here were obtained using an execution-driven simulator based on the SimpleScalar toolkit [5]. The simulator developed by Craig Zilles [46] was used, with its front end completely rewritten for more accurate modelling. Since improving the front-end is only useful if it is a bottleneck, a 16-wide out-of-order superscalar processor with abundant functional units and large caches was simulated. Table 2 describes the base-case processor in detail.

All benchmarks were taken from the SPEC CPU 2000 benchmark suite and were compiled with 'peak' settings using the Compaq Alpha compiler. Results are only reported 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 1 lists the benchmarks and the inputs used.

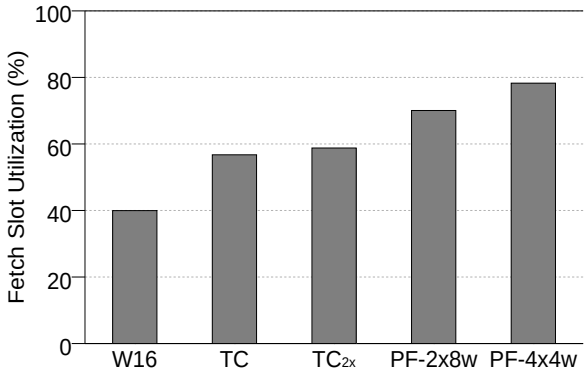

Figure 2. Fetch Slot Utilization

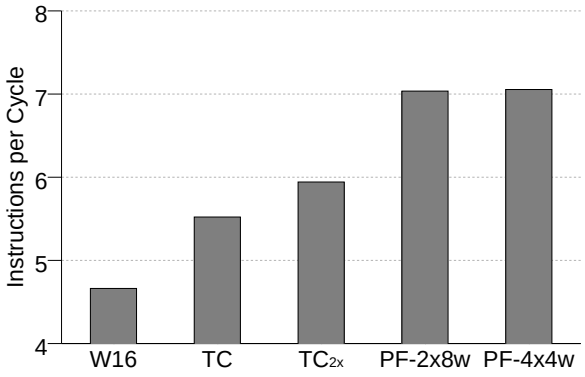

Figure 3. Average Fetch Rate

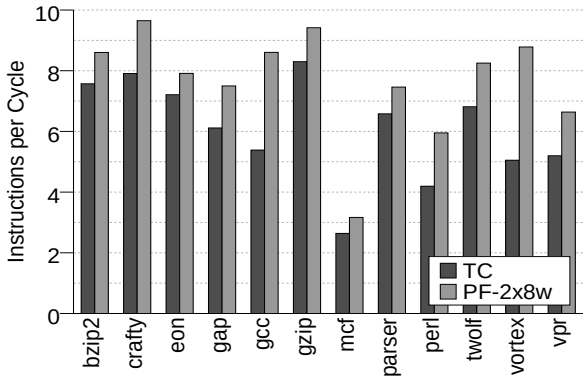

Figure 4. Fetch Rate

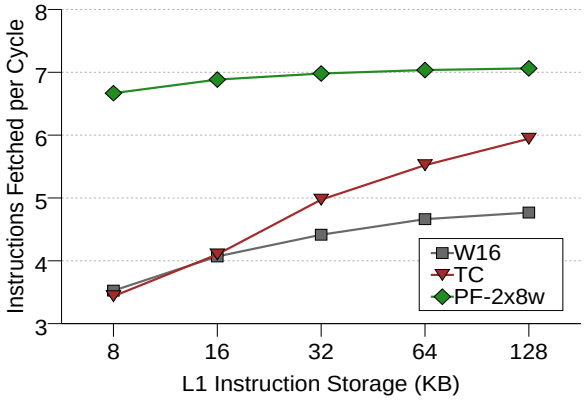

Figure 5. Sensitivity to Cache Size

2.2.1 Fetch Rate

The time taken by a parallel fetch unit 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, the aggregate fetch throughput is higher since fewer fetch opportunities are wasted due to instruction stream discontinuties. This can be quantified by using the notion of *fetch slots*. 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. For sequential fetch mechanisms, the number of fetch slots in a cycle is zero if fetch is stalled, or equal to the fetch width otherwise. For the parallel fetch mechanism described above, the number of fetch slots in a cycle is the sum of the widths of the sequencers that are active. The ratio of the total number of instructions fetched to the total number of fetch slots is the efficiency with which the fetch mechanism is able to utilize the available fetch width.

Figure 2 shows the fetch slot utilization and fetch rate of three front-end mechanisms. As expected, **W16** does not perform well. It is able to utilize only 40% of the available slots, underscoring the importance of a high-performance fetch mechanism when using an aggressive back-end. A trace cache increases the average utilization to about 60%—a little lower than the ratio between average and maximum trace size (see Table 1). **PF-2x8w** achieves about 70% uti-

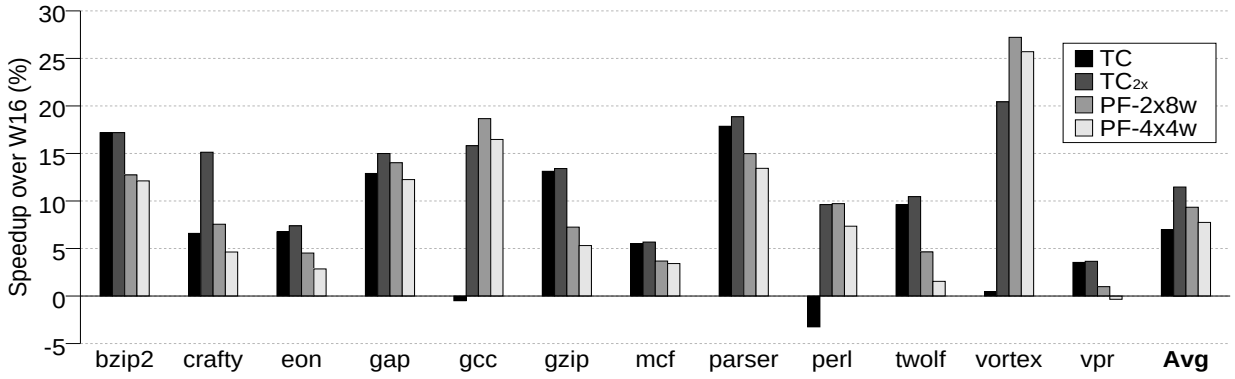

Figure 6. Performance

lization on average—17% more than **TC** and $\mathbf{TC_{2x}}$. **PF-4x4w** further increases utilization, since narrower sequencers lead to fewer wasted slots, achieving 80% utilization on average.

Figure 3 shows the average fetch rate for each scheme. The average fetch rate of **PF** is 7 instructions per cycle—about 20% higher than the trace cache, and 49% higher than **W16**. Figure 4 shows the fetch rate of **TC** and **PF-2x8w** for each benchmark. **PF-2x8w** has a higher fetch rate than **TC** in all cases, with the difference ranging from 20% to 80% depending on benchmark.

Using parallelism to achieve higher aggregate throughout also makes this mechanism tolerant to cache miss latencies. 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. In addition, use of a conventional instruction cache instead of a trace cache results in more effective utilization of cache space. Therefore, programs with large working sets are likely to perform better. Figure 5 shows the change in average fetch rate for different mechanisms as the size of the level-1 instruction storage is varied from 128 KB to 8 KB. The fetch rate of **PF-2x8w** decreases by less than 5% as the cache size is reduced by a factor of sixteen, whereas the fetch rate of **W16** and **TC** decreases by 27% and 42% respectively.

2.2.2 Performance

Figure 6 shows the overall performance of parallel fetch and trace cache normalized to performance of **W16**. On average, **PF-2x8w** and **PF-4x4w** perform 9.5% and 7.5% better than **W16** respectively—higher than **TC**, and only a little lower than $\mathbf{TC_{2x}}$ despite half the cache space.

As noted earlier, $\mathbf{TC_{2x}}$ is identical to **TC**, except that total L1 instruction storage is doubled from 64KB to 128KB. The difference between the **TC** and $\mathbf{TC_{2x}}$ bars is therefore the benefit due to a larger cache. This difference is large in four cases—`crafty`, `gcc`, `perl`, and `vortex`—indicating a large instruction working set. **PF** performs better than **TC** on these benchmarks, but on most other benchmarks **TC** outperforms **PF** despite the latter's higher fetch rate. This is due to serialization of the instruction stream at the rename stage.

To see why, consider the following two scenarios: (1) Immediately after fetch is redirected, the effective throughput is limited to throughput of a single sequencer since all instructions in the first fragment after the fetch redirect

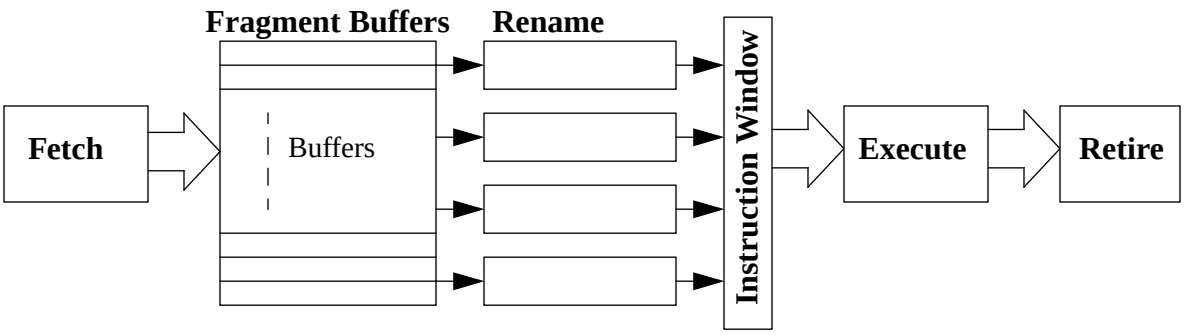

Figure 7. Parallel Renaming

must be renamed before any later instructions can be renamed, and (2) On a cache miss, instructions from future fragments may be fetched into fragment buffers before the cache-missing instructions, but none of those instructions can be renamed before the cache miss is serviced. In both these cases, useful instructions from future portions of the instruction stream are available, but they cannot be renamed and executed since all prior instructions have not been fetched. Thus, to achieve the potential benefits of a parallel fetch unit, it is necessary to be able to rename instructions out-of-order as well. This is the subject of Section 3.

2.3 Status

A parallel fetch mechanism using simple fragment selection heuristics and a path-based predictor has been implemented and preliminary performance and sensitivity studies have been carried out. This work was presented at the International Conference on Parallel Processing, 2002 [21].

Fragment selection and prediction is the main issue that I plan on exploring in the future. In addition, the dissertation will also contain a detailed study of the design space, especially relating to sizes of hardware structures and fragment reuse.

3 Out-of-Order Renaming

We saw in the last section that multiple sequencers are able to achieve a consistently high fetch rate even in the presence of cache misses. However, the high fetch rate is not directly translated into performance since a parallel instruction fetch stage was feeding a sequential rename stage. Serialization at the rename stage can be avoided if the rename stage can be built in a parallel fashion as well, as shown in Figure 7.

The difficulty in building a rename unit of this kind is, of course, dependencies between instructions. Unlike instruction fetch, where the same instructions are obtained regardless of the order they were fetched in, renaming depends on the order of instructions to ensure that logical register names get mapped to the correct physical registers. This is illustrated in Figure 8. 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.

Therefore, a parallel renaming mechanism must either delay renaming I2 until I1 has been renamed, or rename I2 speculatively and ensure that I1 maps its output to the predicted register [39]. Both these alternatives have been explored in the past by speculative parallelization architectures, since the same problem occurs in that domain as well. In addition, a parallel renaming mechanism needs a way of knowing when such a dependence exists without examining the instructions themselves, since I2 may be renamed before I1 is even fetched. The succeeding two sections discuss possible solutions to these problems.

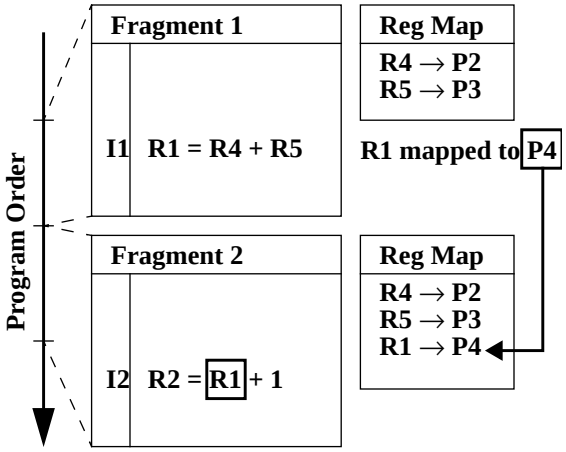

Figure 8. The Sequential Nature of Renaming

3.1 Determining Dependencies Correctly

A straightforward method to ensure that sources of all instructions are mapped to the correct physical registers is to delay renaming an instruction until its source instructions have been renamed. However, delaying instructions has disadvantages: the instruction spends more time in the fragment buffer, preventing the buffer from being used to fetch other future instructions, and thus reducing fetch throughput.

This delay can be avoided by using a technique proposed in Skipper [8]: the instruction waits until the its logical source registers have been assigned physical registers, but this assignment is done speculatively, without waiting for the actual source instructions to be fetched. The live-out registers of each fragment are predicted, and all live-outs are assigned physical registers as early as possible. Fragments later in the instruction stream use the pre-assigned mappings to rename instructions, and thus renaming can take place regardless of whether the actual instructions corresponding to previous fragments have been fetched.

One disadvantage of this technique is that the register assignments to fragment live-outs must happen serially, limiting the average front-end throughput to one fragment per cycle. This serialization can be avoided by using implicit conventions for mapping fragment live-outs to physical registers. This would restrict the flexibility of logical to physical register mapping, but may enable renaming fragments entirely in parallel. Similarly, if the register file is clustered, a different parallel renaming technique may be appropriate. I may investigate these issues if the serialization in the Skipper technique turns out to be an important limitation.

3.2 Live-out Prediction

The renaming mechanism described above requires that the *live-outs* of all fragments, i.e., the set of registers that instructions in a fragment write to, be known. It is trivial to obtain this information by examining the instructions themselves. However, we need a mechanism to determine this information before the instructions are available.

Note that in the absence of self-modifying code the set of instructions that comprise a fragment, and thus determine the live-outs of the fragment, never change. After a fragment has been observed once, its live-outs can be predicted very accurately by simply using a lookup table. Figure 9 shows the average live-out prediction accuracy for

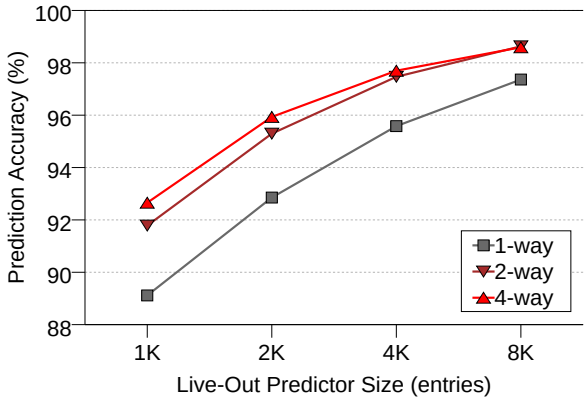

Figure 9. Live-Out Prediction Accuracy

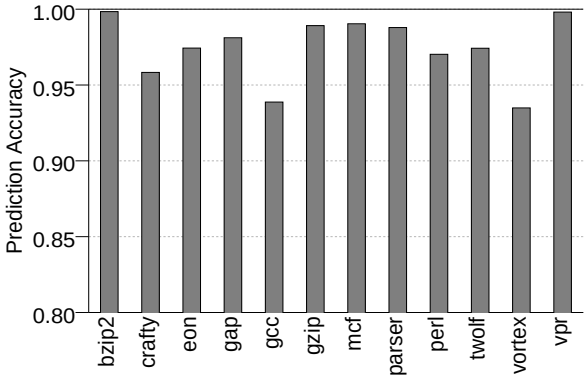

Figure 10. Accuracy of a 2-way 4K entry table

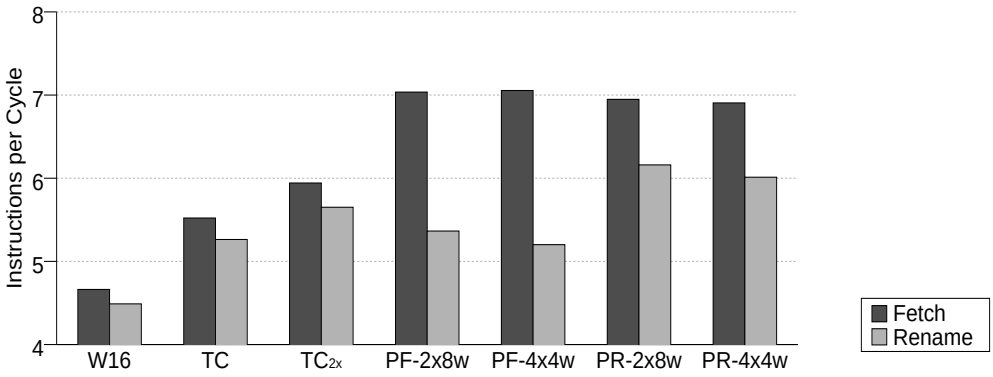

Figure 11. Fetch and Rename Rate

different table sizes and associativities. A 2-way, 4K entry predictor is assumed in the rest of this document, and Figure 10 shows its prediction accuracy for all benchmarks.

Live-outs are stored as a pair of bitmaps: one bitmap corresponding to logical registers, with ones indicating which registers are alive-outs; and one bitmap corresponding to instructions, with ones indicating which instructions produce live-out values. A 2-way 4K entry predictor occupies 42KB of storage with this encoding. The space requirement could be significantly reduced using a more complex encoding, since the actual number of live-outs per fragment is only 4–6 registers in most cases.

3.3 Preliminary Results

Figure 11 shows the average number of instructions fetched and renamed each cycle with and without parallel renaming. The bars marked **PF** correspond to sequential renaming (same as Figure 3), and the bars marked **PR** show the corresponding parallel renaming configurations. The width and number of renamers is identical to the width and number of sequencers. 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,

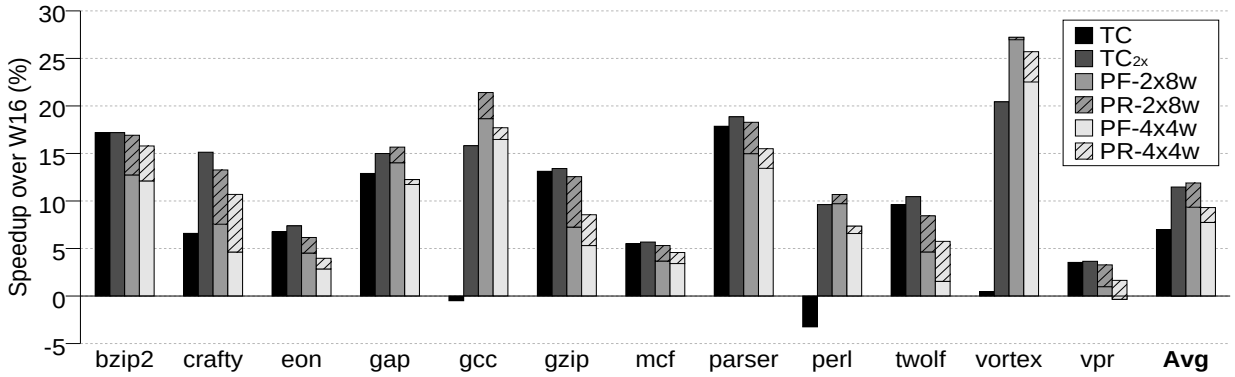

Figure 12. Performance

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.

Figure 12 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.

PR-2x8w performs within 2% of both **TC** and **TC_{2x}** on most benchmarks. On the four benchmarks whose instruction working sets are large (`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.

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**. Thus, better control prediction would be necessary to realize the advantages of four sequencers over two. Finally, renaming instructions in an order different from program order has overheads. The next section discusses this in more detail.

3.4 Out-of-Order Renaming with Sequential Front Ends

Nothing in the design of this renaming mechanism necessarily requires a parallel fetch unit, as long as a set of fragment buffers exist at the interface between the fetch and rename stage. 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

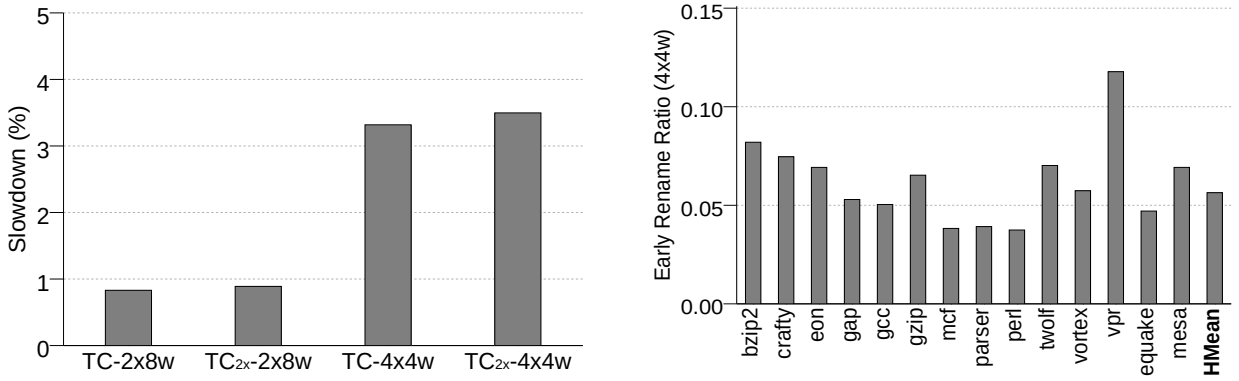

Figure 13. Monolithic versus Parallel Renaming

trace every cycle into a free fragment buffer, the parallel renaming mechanism described could be used without any changes. Parallel renaming is not a IPC enhancing technique for sequential fetch mechanisms—however, a parallel renamer may enable higher clock speeds than a wide monolithic renaming unit.

Figure 13 shows the performance penalty of using a parallel renaming unit with a trace cache fetch mechanism. The base case is a trace cache with a 16-wide monolithic renamer. Two parallel renamers are studied: (1) 2×8w—two 8-wide renamers operating in parallel, and (2) 4×4w—four 4-wide renamers operating in parallel. A 2×8w renaming unit performs within 1% of a monolithic renaming unit on average. A 4×4w renamer suffers a higher penalty of about 3.5%. Some of this penalty comes from live-out mispredictions; however, assuming a perfect live-out predictor only reduces the slowdown by 33%. A small fraction of the rest is due to the longer front-end pipeline. That still leaves a substantial portion of the slowdown unexplained.

The rest of the loss in performance is caused by instructions being renamed before their sources. Simulations indicate that 4–12% of dynamic instructions are renamed before the instructions producing the corresponding sources when a 4×4w renamer is used. Regardless of whether an instruction is renamed before its sources, it must wait for all source instructions to execute before it can be executed. Thus, renaming instructions too early diverts renaming resources from other instructions that could possibly have executed if they had been renamed instead. Early renaming also increases the pressure on the instruction window and register file.

This overhead could be reduced if the order of renaming instructions could be chosen more appropriately, possibly using the knowledge of data dependencies. Later portions of this document discuss ways in which this might be achieved.

3.5 Status

An out-of-order renaming mechanism has been implemented in conjunction with the parallel fetch mechanism described in Section 2. A paper based on this work was presented at the 30th Annual International Symposium on Computer Architecture, 2003 [22].

In the future I plan on investigating the relationship between out-of-order renaming and fragment selection, and exploring other techniques for ensuring correct preservation of dependencies if the current scheme turns out to have important limitations. I also intend to develop a more compact encoding for information in the live-out predictor.

4 General Out-of-Order Fetch

Sections 2 and 3 describe mechanisms that can fetch and rename fragments in parallel, and thus achieve a limited degree of out-of-order behavior in the front-end. However, each individual fragment must be fetched sequentially—the ability to reorder fetch comes only from interleaving different fragments. Thus the out-of-order-ness is, in some sense, undirected and uncontrolled. For instance, fetching and renaming instructions in an explicitly defined order in order to, say, alleviate the out-of-order renaming overhead discussed in Section 3.4 is difficult, if not impossible. In contrast, out-of-order back-ends are able to reorder operations on an instruction granularity, and sometimes sub-instruction granularity. In this section I propose a mechanism that allows more general reordering of operations in the front-end—it allows the fetch of individual cache blocks comprising the dynamic instruction stream to reordered arbitrarily, limited only by the size of the fetch window.

Coming back to the out-of-order execution analogy: in-order fetch corresponds to a simple in-order processor, fragment-based out-of-order fetch corresponds to a Multiscalar processor comprising in-order cores, and general out-of-order fetch corresponds to a dynamically scheduled superscalar processor.

4.1 Limitations of Fragment-based Out-of-Order Fetch

Most limitations of parallel fetch arise because fetch reordering is limited to interleaved operation of several sequential fetch engines. Some specific problems are listed below:

1. **Dependence on aggressive control flow prediction:** Since at most one block of instructions from a fragment can be fetched in any cycle, increasing parallelism requires predicting tens of traces ahead, which translates to hundreds of instructions. Predicting control flow that far into the future makes the technique vulnerable to mispredictions.

2. **Lost opportunity due to reordering restrictions:** Suppose a block encounters a cache miss, later blocks corresponding to the same fragment cannot be fetched instead; only blocks from other fragments can be fetched in its place. If no future fragments have been predicted yet, or if future fragments are waiting for the cache as well, it is impossible to overlap the cache miss with fetch of other blocks possibly present in the cache.

3. **Inability to fetch instructions close to each other quickly:** Since only fetch of instructions from different fragments is concurrent, instructions from the immediate future are often fetched after instructions that are far away. Although instructions that are far away are often more likely to be independent, in some cases it may be desirable to first fetch the instructions in the immediate future at a high rate—for example, immediately after a control misprediction.

4. **Renaming overhead:** As described in Section 3.4, there are overheads associated with renaming instructions out-of-order. This can possibly be addressed by being smarter about the order in which instructions are fetched instead of fetching them in an arbitrary order, but fragment-based out-of-order fetch makes such fine control difficult.

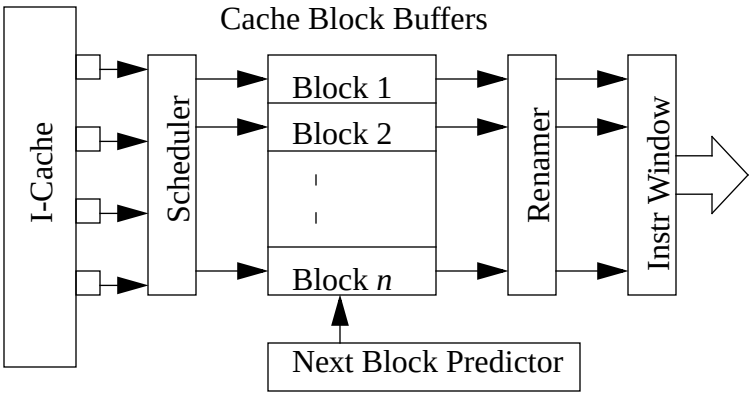

Figure 14. Front-End based on the OOO Execution Metaphor: Blocks to be fetched correspond to instructions in an OOO scheduling window. They are scheduled depending on the various structural hazards. Since it is impossible to fetch instructions from the cache at a granularity finer than blocks, this mechanism is maximally general in the degree of out-of-orderness.

5. **Duplicate fetch:** Since fragment boundaries do not correspond to cache block boundaries, many cache blocks are fetched twice: first for the end of one fragment, and then for the beginning of another. Also, if the same instructions are in two different fragments, the same cache block(s) are fetched twice since reuse only happens at the granularity of fragments.

These limitations exist because the front-end operates at the granularity of fragments, but this does not match the granularity at which instructions can be fetched from the cache/memory.

4.2 Block-based Out-of-Order Fetch

Ideally, we would like to be able to reorder fetch of individual instructions arbitrarily, just as an out-of-order superscalar processor can arbitrarily reorder execution. However, fetching at the granularity of individual instructions is wasteful since an instruction cache can only access storage at the granularity of cache blocks. If only one instruction is fetched, the excess will be discarded either by the fetch unit, or by the cache.

Thus, the smallest granularity at which it is useful to reorder fetch is at the granularity of cache blocks. Fetching the dynamic instruction stream involves fetching a corresponding series of cache blocks; and we would like to be able to arbitrarily reorder the fetch of those cache blocks. The front-end should be designed around the notion of interleaving fetch of cache blocks, and similarly, the control prediction mechanism should predict a stream of cache blocks and masks representing which instructions to select from the block, rather than depending on some artificial division of the instruction stream into traces or fragments.

The front-end can then be structured in a way similar to an out-of-order execution window: the various cache blocks to be fetched correspond to the instructions to be executed; cache ports correspond to execution units; and every cycle, a scheduler chooses a set of cache blocks to be fetched that cycle, depending on structural hazards and ordering heuristics.

4.3 Design

Figure 14 illustrates this design schematically. It is similar in spirit to the parallel fetch design proposed in Section 2, with some important changes. The fragment buffers are replaced with an array of cache block buffers. Each of these buffers stores a single cache block, and can thus be filled using a single access to the instruction cache. The

fragment predictor is replaced with a *next block predictor*, which predicts a sequence of cache blocks and a mask specifying which instructions in the cache block are a part of the dynamic instruction stream. There are no individual sequencers: all addresses required are generated directly by the next block predictor. The scheduler selects the blocks to be fetched, forwards the addresses to the appropriate cache banks, and ensures the data from the cache is routed correctly.

The renaming unit can either be in-order, in which case it reads blocks from the cache block buffers in program order; alternatively, it too can be out-of-order. As we saw in Section 3, an out-of-order renaming unit is likely to be required to derive maximum benefit from an out-of-order fetch unit. Techniques similar to those described in Section 3 can be used to build an out-of-order renaming unit.

4.3.1 Next Block Predictor

The path based prediction technique used earlier for the fragment predictor is likely to be effective in this situation as well. There is previous work on predicting *instruction streams* by Ramirez et al. [31] that can be adapted to predict cache blocks. They have demonstrated prediction accuracies similar to those seen for trace prediction.

Predictor bandwidth is an important requirement for the next block predictor, since all program sequencing depends on the next block predictor. If the predictor can predict n blocks every cycle, and each block contains i instructions on average, the throughput of the fetch unit will be limited to a maximum of $n \bullet i$ instructions. Typical values of i are likely to be small (~8 instructions), and therefore it is important for n to be more than one.

Various techniques have been proposed in the past for predicting multiple basic blocks every cycle [35, 43, 44, 45]. They typically work by storing predictions/addresses corresponding to multiple basic blocks in a single predictor entry, or by multiporting the predictor so that multiple entries can be read simultaneously. This increases the space required for the predictor. Another possibility is to aggregate information about several contiguous blocks in a single predictor entry and to translate that to individual block addresses on the fly, as has been proposed by FTQ [32] and Stream Predictor [31]. I plan on investigating these and possibly other techniques to increase prediction bandwidth of the block predictor.

4.3.2 Scheduler

The primary function of the scheduler is to pick non-conflicting cache blocks to be fetched in a cycle. It may simply always prefer earlier blocks to later ones, or it may use additional heuristics to decide which block to choose for fetch. Section 5 will discuss possible heuristics in greater detail.

4.4 Instruction Cache Block Size

Structuring the front-end in this fashion changes the trade-offs regarding the optimal cache block size. Conventional sequential front ends can only fetch one cache block per cycle (or some multiple thereof), so it is important that blocks be large. However, larger blocks decrease the flexibility of the cache to choose what to keep, and so increase the chance of conflict misses. Larger blocks also limit the granularity at which fetch can be reordered. Thus the optimal cache block size for this mechanism may be smaller than for conventional fetch mechanisms.

On the other hand, the smaller the cache block size, the more cache block predictions would be needed per cycle, the more cache ports would be required to sustain the same net throughput, and the more read/write ports would be required in the cache block buffers. I plan to investigate these trade-offs in detail in my research.

5 Heuristics for Reordering Fetch

The technique described in the Section 4 allows complete flexibility in the order of fetching blocks from the instruction cache, but the flexibility of reordering is used only when fetching sequentially is not possible. It may be possible to actively modify the order in which instructions are fetched in order to improve certain facets of processor behavior. This becomes especially important as the sizes of instruction windows increase: a small instruction window can be filled completely with only a few instruction cache accesses so the exact order of cache accesses is not very important, but the order in which a large window is filled may make a significant difference.

In addition, as we saw in Section 3.4, out-of-order renaming units operate less effectively than in-order renaming units if the order of renaming instructions is arbitrary. It is therefore desirable to be able to control the order in which instructions are fetched and renamed so that performance degradation of out-of-order renaming can be minimized.

The following sections discuss some potential heuristics for optimizing fetch order.

5.1 Fetching Critical Instructions First

Previous research has shown that certain instructions in the program are more "critical" than others since they lie on the critical path of execution. Many techniques have been proposed to identify these instructions and give them priority for execution [11, 12, 38, 41, 42]. These techniques usually try to execute these critical instructions earlier and/or faster. Out-of-order fetch enables the processor to fetch and execute these instructions much earlier than otherwise possible. Criticality predictions can be produced by an independent predictor, or directly integrated with the next block predictor so that each entry in the cache block buffer also stores a field representing urgency of the instructions.

The applicability of out-of-order fetch to this problem would also depend on the distribution of critical instructions in the program. Since fetch can only be reordered on a cache-block granularity, no optimization would be possible if each cache block had a critical instruction. Using appropriately-sized cache blocks would be important to ensure that sufficient reordering flexibility is possible.

Most studies of criticality have so far focussed on individual instructions. It would be interesting to study whether the same phenomena is observed at a higher granularity.

5.2 Data-dependence–based Fetch

Alternatively, it is also possible to fetch instruction blocks in the order of their data dependencies. As we have already seen, live-outs can be predicted fairly accurately; live-ins can be predicted in a similar fashion, and the front-end can then select blocks to fetch depending on direct information of how independent the computation in a particular block is. This can help mitigate the performance degradation due to out-of-order renaming discussed in Section 3.4.

Data dependencies are yet another property that has only been studied at the instruction granularity, and it is not clear whether data-dependencies at the cache-block granularity would give sufficient reordering flexibility to the

Table 3: Schedule

Topic	Component	Completion
Parallel Fetch and Rename	Parallel Fetch – Basic Technique	Complete (ICPP 2002)
	Parallel Rename – Basic Technique	Complete (ISCA 2003)
	Fragment selection heuristics, Fragment reuse	Summer 2003
	Live-out predictor, Scalability	Summer 2004
Block-Based Out-of-Order Fetch	Basic Technique	Fall 2003
	Reordering Heuristics	Spring 2004
	Design space exploration	Summer 2004
Dissertation	Write, Interview	Fall 2004

front-end. Unlike individual instructions, instruction blocks would almost always be dependent on the last few preceeding blocks. However, fetch order optimization may still be possible by taking into account the number and length of the dependency chains between instructions.

5.3 Lower Resource Utilization

Just like dependency information can be used to select independent instruction blocks for fetch, it can also be used to delay fetch of dependent blocks. This can be used to decrease the lifetime of instructions in the processor, possibly reducing the pressure on various processor resources. This can be important in resource constrained environments, or in SMT processors where the freed resources can be used to execute instructions from another thread. This may also enable some power optimizations like turning off certain sections of the processor, etc. A scheme for delaying instruction fetch, called *Just-in-Time Instruction Delivery* [17], has been shown to reduce energy consumption significantly in the context of a sequential fetch mechanism with minimal impact on performance. The benefits are likely to be even greater with the increased flexibility available to an out-of-order front-end.

6 Summary and Schedule

My research focuses on out-of-order front-ends. Current front-ends operate sequentially, even though most programs contain complex control flow that cannot be mapped onto a sequential storage order. This mismatch makes it difficult and expensive to achieve high front-end performance. A front-end explicitly designed around the idea of out-of-order processing is able to achieve high front-end performance, and has additional benefits like higher latency tolerance and greater flexibility.

My dissertation will contain detailed descriptions and characterizations of two different out-of-order front ends: (1) Parallel fetch and rename, and (2) Block-based out-of-order fetch. Both these designs have different strengths and weaknesses, which I will explore in my research. The status of each of these is detailed below:

1. **Parallel Fetch and Rename:** This mechanism has been implemented in an execution-driven simulator and a preliminary study has been completed. Since fragment selection is critical to this mechanism, I plan on exploring fragment selection heuristics in detail. If required by the fragment selection heuristics I examine, I will study frag-

ment prediction strategies as well. Parallel renaming is another component that needs further study, in particular its interactions with fragment selection and its implications for physical register organization. In addition, I will study fragment reuse and live-out prediction.

2. **Block-Based Out-of-Order Fetch:** This mechanism is still a the concept stage: I will implement it in the aforementioned simulator, and study the design issues that come up. Research challenges include next-block prediction, prediction bandwidth, and granularity selection. Finally, I will study the various scheduling policies that become possible once a general out-of-order fetch mechanism is available.

A schedule of the tasks appears in Table 3. I plan on graduating before the end of 2004.

7 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] B. Black, B. Rychlik, and J. P. Shen. The Block-Based Trace Cache. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 196–207, Atlanta, Georgia, May 2–4, 1999.

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

[5] 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.

[6] 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.

[7] I. K. Chen, C. C. Lee, and T. N. Mudge. Instruction Prefetching using Branch Prediction Information. In *International Conference on Computer Design*, pages 593–601, October 1997.

[8] 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.

[9] 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.

[10] K. Ebciouglu and E. R. Altman. DAISY: Dynamic Compilation for 100% Architectural Compatibility. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 26–37, Denver, Colorado, June 2–4, 1997.

[11] B. Fields, S. Rubin, and R. Bod'ik. Focusing Processor Policies via Critical-Path Prediction. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 74–85, Göteborg, Sweden, June 30–July 4, 2001.

[12] B. R. Fisk and R. I. Bahar. The Non-Critical Buffer: Using Load Latency to Improve Data Cache Efficiency. In *IEEE International Conference on Computer Design*, Texas, 1999.

[13] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, second edition, 1996.

[14] W-M. W. Hwu and P. P. Chang. Achieving High Instruction Cache Performance with an Optimizing Compiler. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 242–251, Jerusalem, Israel, May 28–Jun 1, 1989.

[15] E. Jacobsen, E. Rotenberg, and J. E. Smith. Assigning confidence to conditional branch predictions. In *Proc. 29th International Symposium on Microarchitecture*, Dec. 1996.

[16] 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, Research Triangle Park, North Carolina, Dec. 1–3, 1997.

[17] T. Karkhanis, J. E. Smith, and P. Bose. Saving Energy with Just in Time Instruction Delivery. In *Proceedings of the 2002 International Symposium on Low Power Electronics and Design*, pages 178–183, 2002.

[18] R.E. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2), Mar./Apr. 1999.

[19] S. McFarling and J. Hennessy. Reducing the Cost of Branches. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 396–403, Tokyo, June 2–5, 1986.

[20] 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.

[21] 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.

[22] P. S. Oberoi and G. S. Sohi. Parallelism in the Front-End. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, San Diego, California, June 9–11, 2003.

[23] 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.

[24] 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.

[25] 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.

[26] K. Pettis and R. C. Hansen. Profile Guided Code Positioning. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 16–27, White Plains, New York, June 20–22, 1990.

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

[28] A. Ramirez, L. A. Barroso, K. Gharachorloo, R. Cohn, J. L. Larriba-Pey, P. G. Lowney, and M. Valero. Code Layout Optimizations for Transaction Processing Workloads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 155–164, Göteborg, Sweden, June 30–July 4, 2001.

[29] 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.

[30] A. Ram'irez, J. L. Larriba-Pey, and M. Valero. Trace Cache Redundancy: Red & Blue Traces. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, pages 325–333, Toulouse, France, January 8–12, 2000.

[31] 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.

[32] 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*, pages 234–245, Atlanta, Georgia, May 2–4, 1999.

[33] G. Reinman, B. Calder, and T. Austin. Fetch Directed Instruction Prefetching. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, pages 16–27, Haifa, Israel, November 16–18, 1999.

[34] 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.

[35] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud. Multiple-Block Ahead Branch Predictors. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 116–127, Cambridge, Massachusetts, October 1–5, 1996.

[36] J. E. Smith and W-C. Hsu. Prefetching in Supercomputer Instructin Caches. In *Proceedings of Supercomputing*, November 1997.

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

[38] S. T. Srinivasan, R. D-C. Ju, A. R. Lebeck, and C. Wilkerson. Locality vs. Criticality. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 132–143, Göteborg, Sweden, June 30–July 4, 2001.

[39] 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, Research Triangle Park, North Carolina, Dec. 1–3, 1997.

[40] J. W. Stark. *Out-of-Order Fetch, Decode, and Issue*. PhD thesis, University of Michigan, 2000.

[41] E. Tune, D. Liang, D. M. Tullsen, and B. Calder. Dynamic Prediction of Critical Path Instructions. In *Proc. 7th International Symposium on High Performance Computer Architecture*, Jan. 2001.

[42] C-L. Yang, B. Sano, and A. R. Lebeck. Exploiting Instruction Level Parallelism in Geometry Processing for Three Dimensional Graphics Applications. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 14–24, Dallas, Texas, November 30–December 2, 1998.

[43] T-Y. Yeh, D. T. Marr, and Y. N. Patt. Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache. In *Conference Proceedings, 1993 International Conference on Supercomputing*, pages 67–76, Tokyo, July 20–22, 1993.

[44] T-Y. Yeh and Y. N. Patt. A Comprehensive Instruction Fetch Mechanism for a Processor Supporting Speculative Execution. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 129–139, Portland, Oregon, December 1–4, 1992.

[45] T-Y. Yeh and Y.N. Patt. Branch History Table Indexing to Prevent Pipeline Bubbles in Wide-Issue Superscalar Processors. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 164–175, Austin, Texas, Dec. 1–3, 1993.

[46] C. Zilles. *Master/Slave Speculative Parallellization and Approximate Code*. PhD thesis, University of Wisconsin-Madison, 2002.