Out-of-Order Instruction Fetch using Multiple Sequencers

Paramjit Oberoi and Gurindar Sohi

Computer Sciences Department, University of Wisconsin - Madison
1210 West Dayton Street, Madison, WI 53706-1685, USA
{param,sohi}@cs.wisc.edu

Abstract

Conventional instruction fetch mechanisms fetch contiguous blocks of instructions in each cycle. They are difficult to scale since taken branches make it hard to increase the size of these blocks beyond eight instructions. Trace caches have been proposed as a solution to this problem, but they use cache space inefficiently.

We show that fetching large blocks of contiguous instructions, or wide fetch, is inefficient for modern out-of-order processors. Instead of the usual approach of fetching large blocks of instructions from a single point in the program, we propose a high-bandwidth fetch mechanism that fetches small blocks of instructions from multiple points in a program.

In this paper, we demonstrate that it is possible to achieve high-bandwidth fetch by using multiple narrow fetch units operating in parallel. Our mechanism performs as well as a trace cache, does not waste cache space, is more resilient to instruction cache misses, and is a natural fit for techniques that require fetching multiple threads, like multithreading, dual-path execution, and speculative threads.

1 Introduction

Modern processors need a large instruction window to ensure that many independent instructions are available for execution at any time. As processors with more functional units are built, it is also necessary to increase the instruction fetch bandwidth so that the instruction window size can be correspondingly increased.

Increasing the instruction fetch bandwidth beyond eight instructions per cycle poses special problems. Typical programs contain taken branches every eight instructions on average [6]. At every taken branch in a program, the fetch unit must be redirected to fetch instructions from a new address. Since most instruction caches can only supply data from contiguous memory locations in one cycle, instructions from the branch target address cannot be fetched in the same cycle as the instructions up to the branch. This restricts instruction fetch bandwidth to the average number of instructions between taken branches. The wider the fetch unit, the more likely it is that fetch slots will be wasted because of discontinuities in the instruction stream.

Proposed solutions to this problem can be divided into two categories: (a) augmenting the branch predictor to predict multiple branches per cycle [21] and the instruction cache to supply multiple discontinuous lines per cycle [4], and (b) storing instructions in dynamic execution order in the cache (i.e. using a *trace* cache) [11,12,16]. The first solution makes the branch predictor and the cache more complex, potentially increasing the cycle time; the second solution leads to inefficient use of cache space, potentially increasing cache miss rates.

Both classes of solutions work by fetching a large number of contiguous instructions from a single point in the program every cycle. In this paper, we propose a high-bandwidth fetch mechanism that fetches a small number of contiguous instructions from multiple points in the program, as opposed to fetching a large number of contiguous instructions from a single point.

Fetching a large block of contiguous instructions, or *wide fetch*, is inefficient for out-of-order processors. Figure 1 illustrates how fetch and execution of consecutive instructions overlaps in time. This data is from a 16-wide processor augmented with a trace cache. Traces are variable length, up to a maximum of 16 instructions. The trace selection algorithm is described in Section 2.4.1, and the simulated machine configuration is discussed in Section 3. On average, the processor fetches one trace every two cycles. The first instruction of a typical trace starts execution three cycles after the trace has been completely fetched, and it takes about 20 cycles for all instructions in the trace to begin executing.

As clearly illustrated by the figure, traces are fetched consecutively but their execution is almost entirely concurrent. This is not an entirely unexpected result: previous work has shown that significant parallelism exists between instructions of different traces [20]. Therefore, the actual order in which instructions are executed is very different from their order in the program. First, a small number of data-independent instructions in both traces get

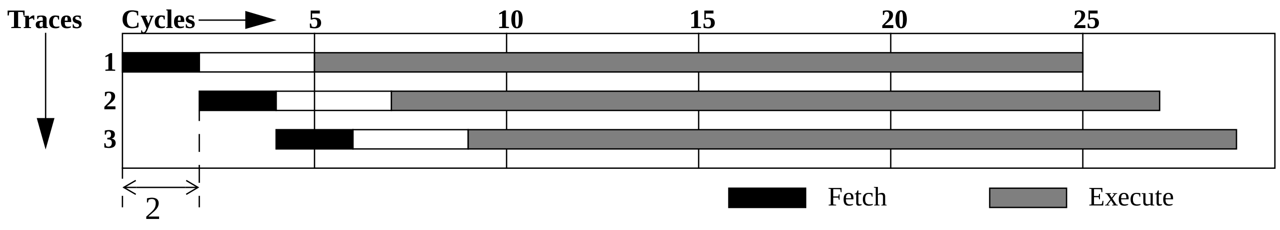

Figure 1. Temporal relationship between instructions in consecutive traces

executed, followed by the rest of the instructions in the traces in dataflow order.

Even though a trace cache can fetch an entire trace in one cycle, all instructions in the trace are *not needed* in that cycle since only a small fraction will be executed immediately. Wide fetch is inefficient for out-of-order processors since instructions are not needed in sequential order. A few instructions from each trace that are first in dataflow order are the ones needed earliest, followed later by the rest of the instructions in the trace. Wide fetch mechanisms must fetch not only the critical instructions that are first in dataflow order, but also all the intervening instructions.

We propose a fetch mechanism that fetches multiple traces concurrently using multiple narrow instruction sequencers instead of one wide sequencer. Since multiple traces are fetched concurrently, the individual instructions are fetched out-of-order. Narrow sequencers use the available bandwidth more effectively since fewer fetch slots are wasted due to branches and cache-line boundaries. Moreover, just like out-of-order execution is able to overlap long latency operations with other useful instructions, out-of-order fetch can tolerate instruction supply delays like I-cache misses by fetching other useful instructions while the miss is resolved. This mechanism can be thought of as just-in-time trace constructor [8] that can build multiple traces concurrently.

We also observed that programs display a remarkably large amount of trace locality. In some benchmarks as many as 70% of the dynamic traces repeat within the next sixteen traces. It may be better to reuse the constructed traces rather than discarding them immediately after use. This turns our mechanism into a small trace cache with a very fast trace construction mechanism. Whereas a trace cache can exploit almost all the locality that is available, it is slow at constructing traces; this mechanism can exploit a smaller amount of locality, but can construct traces quickly. Reusing trace buffers also has the advantage of reducing accesses to the instruction cache by more than 80% in some cases and 50% on average.

Multiple sequencers also make it possible to use the fetch unit in far more flexible ways than a monolithic fetch unit would allow. It is much easier to implement techniques like dual-path execution [5,10], speculative threads [22], etc., that require fetching multiple threads. In a multithreaded processor, multiple sequencers enable a much finer grained control over allocation of fetch resources to threads than is possible otherwise.

The next section describes our mechanism in detail. Section 3 contains an evaluation of the proposed mechanism. We discuss some related work in Section 4. Section 5 concludes the paper with a summary and some future research directions.

2 Instruction Fetch with Multiple Sequencers

The conventional approach of fetching instructions from the location pointed to by the program counter is insufficient for fetching instructions from multiple points in the program. To be able to fetch from multiple locations, we must know the addresses of multiple instructions in the near future rather than just a single current address.

Instead of keeping track of control flow at the granularity of individual instructions we divide the instruction stream into coarser units called traces. A *trace* is a dynamic sequence of instructions in program order, potentially spanning control instructions. Control flow can be predicted on the granularity of traces using a trace predictor [11,16]. It has been demonstrated that trace predictors can achieve equivalent or higher prediction accuracies than conventional branch predictors [7].

Once future control flow can be predicted at trace granularity, multiple traces can be fetched concurrently by using multiple instruction sequencers. This parallelizes instruction fetch: the latency and the width of a single instruction sequencer are no longer the primary determinants of the fetch bandwidth since it is straightforward to add more instruction sequencers to increase the total raw fetch bandwidth.

2.1 Design Details

The architecture we are proposing is illustrated in Figure 2. The instruction fetch queue (IFQ) between the fetch and decode stages is preceded by a set of trace buffers. The fetch unit is replicated, and fetched instructions are placed in trace buffers rather than directly

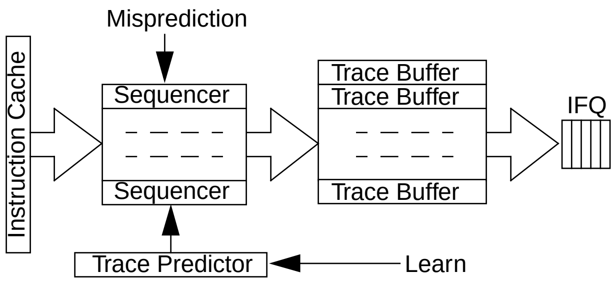

Figure 2. Multiple Sequencers

in the IFQ. A trace predictor is used for control prediction instead of a branch predictor. The L1 instruction cache is banked so that multiple sequencers can fetch instructions concurrently. Bank accesses are controlled by a bank access controller that receives requests for cache accesses, converts these requests into a sequence of bank accesses, and schedules the accesses in order to avoid conflicts. All pipeline stages after the IFQ are unchanged.

Each trace buffer is a small FIFO queue of instructions. Associated with it is a set of registers that describe its *fetch context*: a starting address, a program counter (PC), branch prediction bits, and bits indicating whether the buffer is valid and active. A buffer is *valid* when it contains a trace not completely consumed by the IFQ. A valid buffer is *active* if instructions are still being fetched into it, i.e., if the trace being fetched into it has not been constructed completely. All valid trace buffers are linked to each other by a sequence of next-trace pointers.

Instructions are fetched into active trace buffers starting at the address pointed to by each buffer's PC. Each trace buffer's PC is updated as usual when instructions are fetched into it. The IFQ reads instructions out of the oldest trace buffer and follows the next trace pointer when it encounters the end of the current trace. Once all instructions in the trace are inserted into the IFQ, the trace buffer is marked invalid.

2.2 Fetch Unit

When a program starts executing, the fetch unit fetches instructions sequentially, as usual, and places them in an available trace buffer instead of the instruction fetch queue. It also checks each instruction fetched for trace termination conditions. At the end of the trace, it obtains a new trace buffer, links it to the old one, and marks the old buffer inactive. Then, it continues fetching instructions sequentially into the new trace buffer.

On a trace prediction, the fetch unit obtains a new trace buffer and adds it to the end of the chain of valid buffers. When multiple trace buffers are active, instructions are fetched into all of them simultaneously if enough sequencers are available.

A new trace buffer can be created in the following ways: first, when the trace predictor makes a prediction, and second, by fall through from the previous trace if no prediction is available. A third way is when the fetch unit is redirected after a misprediction—this will be discussed in Section 2.6. Branches are assumed not-taken if predictions are not available.

Instruction fetch into different buffers is completely decoupled. Stalls in one buffer do not affect the other active buffers. Trace end points may be reached in an order completely different from program order. The IFQ still receives the instructions in program order, so no changes are needed to the machine beyond the IFQ, except for some mechanisms for recovering from mispredictions and training the trace predictor.

2.3 Banked Instruction Cache

To enable multiple sequencers to fetch instructions concurrently, the instruction cache must be able to supply multiple cache lines in the same cycle. Although this can be achieved by multiporting the cache so that multiple lines can be read out of a single bank in the same cycle, that would substantially increase the size of the cache, slow it down, and increase its power consumption. We instead achieve the same effect by banking the cache and adding a bank access controller that schedules access to the banks.

Lines are mapped to banks using standard low-order interleaving. The bank access controller services requests in oldest-trace-first order, servicing at most one request from each sequencer in a cycle.

2.4 Trace Selection and Prediction

Good trace selection involves balancing several contradictory requirements. First, the traces must be reasonably long. At the same time, traces should be terminated at the end of control structures like loops and functions to increase the prediction accuracy and decrease the number of unique traces. However, we don't want to stop traces at each control instruction since being able to fetch past control instructions is one of the primary motivations for building traces.

We use function boundaries as the primary division, along with some other constraints to ensure reasonable size, high prediction accuracy, and a small working set. The reader is referred to other papers [11,17] for a more detailed exploration of trace selection techniques.

2.4.1 Trace Selection. We limit the size of traces to 16 instructions. Traces are terminated if (1) they are too long, (2) a call, return, or indirect branch is encountered, or (3) the trace is longer than eight instructions and an unconditional branch is encountered.

Table 1: Trace Characteristics of SPEC 2000 Benchmarks

Benchmark	Dynamic Instructions	Traces	Average Trace Size	Dynamic Traces	Traces Contributing 95% instructions
Integer					
bzip2	8822 M	1819	12.79	690 M	109 (6%)
crafty	4265 M	7541	12.02	355 M	909 (12%)
gap	1246 M	9074	10.70	117 M	972 (11%)
gcc	2016 M	38180	11.26	179 M	7165 (19%)
gzip	3367 M	1942	12.06	279 M	58 (3%)
mcf	260 M	1424	9.84	26 M	132 (9%)
parser	4203 M	6496	10.35	406 M	692 (11%)
Floating Point					
ammp	5491 M	2932	13.11	419 M	332 (11%)
equake	1443 M	2182	11.10	130 M	356 (16%)
lucas	3689 M	1090	15.68	235 M	130 (7%)
mesa	2845 M	2543	11.30	252 M	110 (4%)

Terminating traces at calls/returns and indirect branches enables using a return address stack (RAS) and an indirect branch predictor to supplement the trace predictor, significantly increasing prediction accuracy (see Section 2.4.3 for details). Terminating traces at all unconditional branches when the trace is longer than eight instructions reduces the number of unique traces in the program since traces tend to start at fewer points. This leads to higher prediction accuracy and smaller trace cache miss rate.

Table 1 lists the trace characteristics of the SPEC CPU 2000 benchmarks used in this study. Most benchmarks have a fairly small number of static traces. Integer benchmarks have more traces, as well as smaller traces, than floating point benchmarks. As the last column shows, the number of traces that execute frequently is a small percentage of the number of traces in all benchmarks, and a relatively small absolute number for all benchmarks except gcc.

2.4.2 Trace Prediction. We use the trace predictor proposed by Jacobson et al. [7]. Each trace is assigned a trace identifier obtained by combining bits from the starting address of the trace and its branch history. The predictor consists of a correlated table indexed by trace identifiers of the previous traces, and a smaller filtering table to eliminate the easy to predict traces from the primary table. The trace predictor is based on the Multiscalar task predictor named "DOLC" [1].

Each entry in the predictor contains the starting address of the trace, a bitmap encoding the directions of conditional branches, and bits indicating whether the trace ends with a function call, return, or indirect branch. In case of a trace ending with a call or indirect branch, the predictor also contains the address of the last instruction. The predictor is indexed by a hash function applied to the trace history buffer. The predictor is updated in the commit stage as instructions retire.

The letters D, O, L, and C in the name of the predictor stand for the four parameters that define the predictor: the size of the history buffer (or **D**epth), the number of bits extracted from the identifier of **C**urrent trace, the number of bits extracted from the identifier of **L**ast trace, and the number of bits extracted from the identifiers of **O**lder traces. We use the values D=9, C=9, L=7, and O=4. The primary table contains 64k lines and the secondary table 16k lines. Figure 3 shows both the trace prediction accuracy and the branch prediction accuracy of the predictor.

2.4.3 Returns and Indirect Branches. Since instructions are being fetched out of order, it is possible for a later return instruction to be fetched before an earlier return instruction. However, return instructions must access the RAS in program order for its predictions to be correct. A similar argument holds for the indirect branch predictor.

We access the RAS or the indirect predictor when traces are predicted. Traces can contain at most one call, return, or indirect branch since these instructions end traces. The trace predictor predicts whether a trace will end with these instructions, and the address of the instruction if required. Since traces are predicted in program order, the RAS and the indirect branch predictor are accessed in program order as well.

Restoring predictor state on mispredictions is handled by making a copy of any data that is modified, just like in conventional out-of-order processor. When the last instruction of a trace is fetched, the prediction is verified. On mispredictions, the predictor state is repaired by restoring it to the backed up value and then redoing the modifications made by future traces.

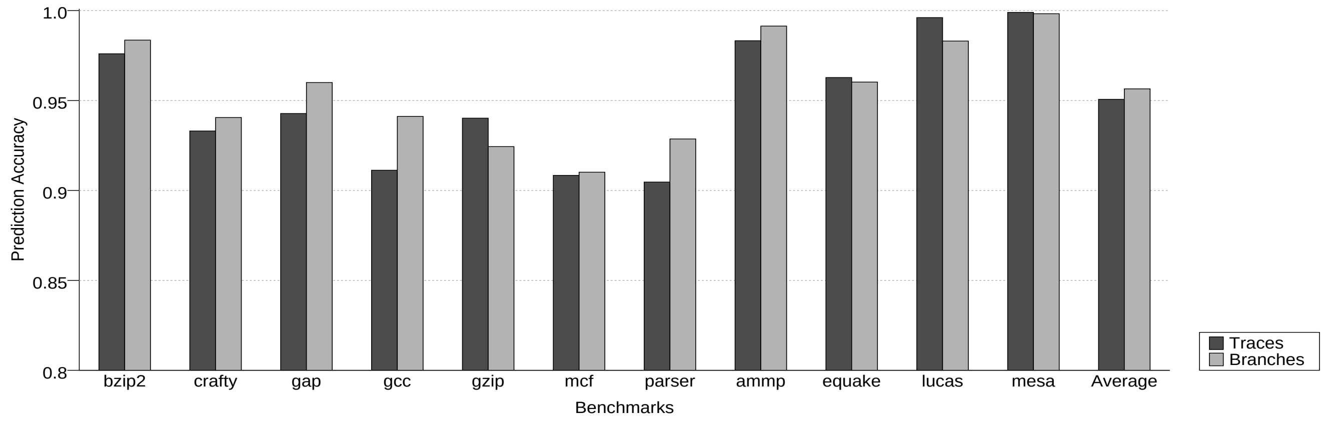

Figure 3. Trace and Branch Prediction Accuracy

2.5 Trace Reuse

The technique as described above discards instructions in trace buffers once they have been decoded. An alternative is to keep instructions in the buffer and to reuse buffers if control flow reaches the same trace again. If there is sufficient locality in the instruction stream, trace reuse could lead to an improvement in performance as well as a reduction in cache/memory traffic.

Reduced cache traffic makes the performance much more robust in the presence of bank conflicts. It also reduces the power consumed by the L1 instruction cache. However, it may not lead to an net reduction in power usage as compared to a processor without multiple sequencers, since the reduction is offset by the power consumed by trace buffers.

2.6 Recovering from Branch Mispredictions

On a branch misprediction, in addition to simply redirecting the fetch unit, the current fetch context must be restored so that (1) the trace selection algorithm does not get misaligned, and (2) the trace identifier history remains accurate. This can be done with mechanisms that already exist in all processors for restoring the global branch history after mispredictions.

Some mispredictions can be detected in the fetch stage itself by comparing the PC following the last instruction in a trace to the predicted next trace. In case of a mismatch, all future traces are marked invalid. Early detection of such mispredictions allows earlier recovery, and reduces the number of spurious instructions fetched and executed.

2.7 Out-of-Order Renaming

Once the instructions are being fetched out of order, it is desirable to be able to execute independent instructions from a later trace before previous traces have been fetched completely. This requires renaming instructions out of order. We believe that a solution similar to those proposed by Stark et al. [19] and Cher et al. [3] can be used to solve this problem.

The trace predictor can be augmented to predict a *rename mask* that identifies independent instructions. This mask can be used to selectively execute only these instructions until all prior traces have been fetched. Delayed instructions can be renamed when all their sources are available. Alternatively, they can be renamed speculatively, and on a misspeculation the source register values can copied into the predicted physical registers after execution of the source instructions.

This paper concentrates only on the fetch component of the instruction supply problem. We do not evaluate out-of-order renaming in this paper.

3 Experimental Evaluation

We used a simulator based on the SimpleScalar toolset [2] to model a multiple sequencer based fetch mechanism. Parameters of the base-case processor are shown in Table 2. We simulate a 16-wide processor with large caches to ensure that it can achieve high IPC, and can therefore benefit from high bandwidth instruction fetch. Large caches also ensure that the conventional instruction fetch mechanism works as well as possible, which shows that the performance improvement due to multiple sequencers cannot simply be achieved by enlarging the instruction cache.

All benchmarks were taken from the SPEC CPU 2000 suite and compiled with optimization using the Compaq/Alpha vendor compiler (version 6.4-214). Test inputs were used, and the programs were simulated for at most one billion instructions. Table 1 lists some characteristics of the benchmarks. We used only a subset of the benchmarks because of limitations in the simulator: (1) many floating point programs produced output that did not match the

Table 2: Simulation Parameters

Width	Fetch, decode and commit at most 16 instructions per cycle
Functional Units	16 integer ALUs, 4 integer multipliers, 4 floating point ALUs, 1 floating point multiplier, 4 load/store units
In-flight Instructions	256 entry instruction window 128 entry load/store queue
L1 Caches (Insn & Data)	64K, 2-way set-associative, 1 cycle access time, 64b blocks
L2 Cache (Unified)	256K, 4-way set-associative, 10 cycle access time, 128 byte blocks
Memory	100 cycle access time

reference output because of differences in the behavior of floating point instructions, (2) no support for the `exec` system call, and (3) problems related to Fortran runtime libraries.

We compare three different instruction fetch mechanisms: conventional 16-wide fetch (**W16**), trace cache (**TC**), and multiple sequencers (**MS**). **W16** fetches instructions sequentially, stopping at the first taken branch or cache line boundary. The number of branch predictions per cycle is unlimited. **TC** models a 2-way set associative trace cache with 16 instructions per trace. On a cache hit, the entire trace can be fetched in one cycle if there are sufficient slots in the instruction fetch queue. On a cache miss, instructions are fetched from the L1 instruction cache using the **W16** mechanism. The processor contains an L1 instruction cache of the same size as the trace cache. We found that this division gave better results than a large trace cache without an L1 instruction cache. In one cycle, instructions can be fetched from either the trace cache, or the L1 instruction cache, but not both. The trace predictor is described in Section 2.4. The total size of the level one instruction storage is kept the same whenever two schemes are compared (i.e. when **TC** is compared to **W16**, the sum of the size of the trace cache and L1 instruction cache in **TC** is equal to the size of the L1 instruction cache in **W16)**.

The various **MS** configurations are labeled using the convention **MS-NxMw** where N is the number of sequencers and M the width of each sequencer. For example, **MS-2x8w** denotes two 8-wide sequencers. There are 16 trace buffers of 16 instructions each. The instruction cache is divided into eight banks as described in Section 2.3. The trace predictor and the trace selection algorithm are identical to those used by **TC**. The trace predictor can make one prediction every cycle. New trace buffers are activated on predictions made by the predictor regardless of the number of buffers already active. Sequencers are assigned to trace buffers in oldest first order. The L1 instruction cache is the same as in **W16**.

The results section is structured as follows. First, in Section 3.1, we study the effect of multiple sequencers on instruction cache traffic. In Section 3.2 we compare the performance of multiple sequencers with conventional instruction sequencing and trace caches. Finally, in Section 3.3 we study the behavior of these mechanisms under high cache miss rates.

3.1 Instruction Cache Traffic

When building a high bandwidth fetch unit, it is inevitable that the number of instructions fetched will be much greater than the number of committed instructions, due to fetching down mispredicted paths. An over-eager fetch mechanism may increase memory traffic and worsen instruction cache performance, doing more harm than good.

Figure 4 shows the number of instructions fetched by **W16**, **TC**, and **MS** without trace reuse, normalized by the number of committed instructions. The number of instructions fetched is equal to the total number of instructions read from the L1 cache for **W16** and **MS**, and the sum of the number of instructions fetched from the L1 cache and the trace cache for **TC**. Floating point benchmarks show only a small increase in the number of instructions fetched. Integer benchmarks show a relatively higher increase of 40% on average. This is comparable to the number of extra instructions fetched by a trace cache.

Figure 5 presents the same data as Figure 4, except that trace reuse is now enabled. Trace reuse directly translates into reduced cache traffic, since the instructions corresponding to reused traces do not have to be fetched from the instruction cache. Cache traffic is reduced dramatically—by 50% over all benchmarks on average, and by more than 80% for four benchmarks (`bzip2`, `gzip`, `mcf`, and `ammp`). The trace buffers act as a filter cache [9,15] making the number of instructions fetched from the cache smaller than the number of instructions executed in most cases.

Interestingly, two benchmarks that benefit the most from trace reuse, `gzip` and `mcf`, are also the ones that had the most wasted instructions without trace reuse. This suggests that most of the mispredictions in them are due to a small number of traces that occur frequently, but are hard to predict—for example, an unpredictable switch statement in a long running loop.

3.2 Performance

As discussed earlier, one way of thinking about this technique is that it constructs traces *just in time* so that by the time control reaches a trace the entire trace has already been constructed. If this happens often, this scheme will

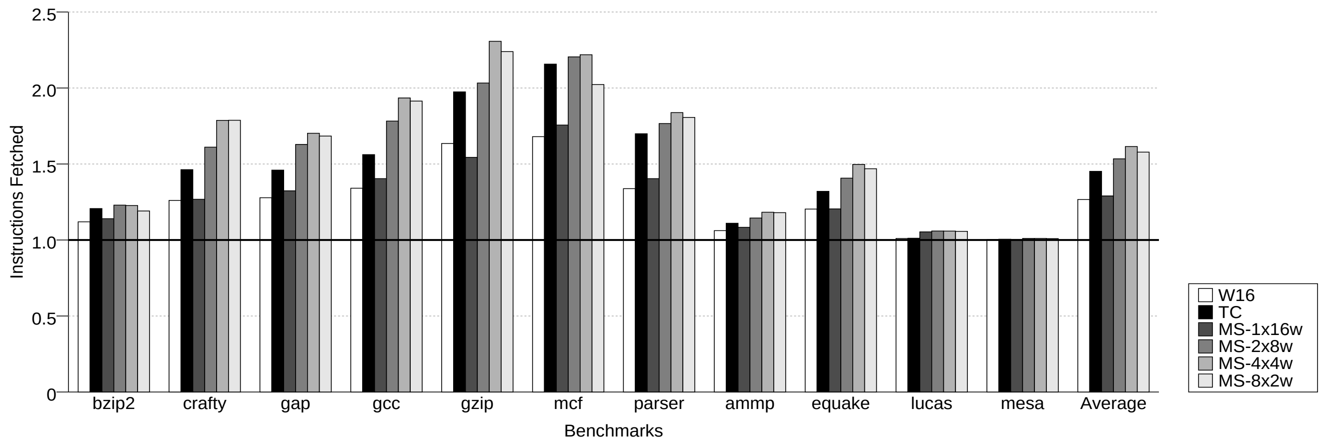

Figure 4. Extra Instructions fetched (without Trace Reuse)

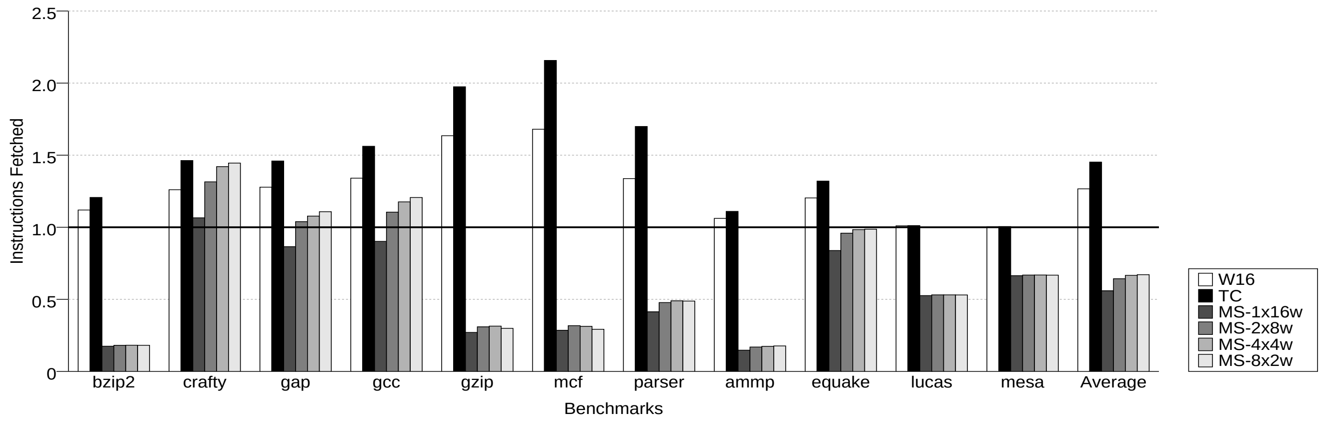

Figure 5. Extra Instructions fetched (with Trace Reuse)

provide the illusion of a trace cache, and therefore will perform as well as a trace cache. In fact, it is likely that in that case **MS** will perform better than a trace cache, since a trace cache requires both predictability as well as locality in the sequence of traces, whereas **MS** requires only predictability since traces are constructed on the fly.

Figure 6 shows the fraction of traces pre-constructed completely before they are needed. On average, 85% of the traces are successfully pre-constructed. The graph also demonstrates the effectiveness of multiple narrow sequencers: **MS-1x16w** is able to construct only 60% of the traces in time, and as the number of sequencers increases the number of successfully constructed traces increases.

Figure 7 directly compares the performance of different fetch mechanisms, normalized by the performance of **W16**. **MS-2x8w** performs better than **TC** on the average, and **MS-4x4w** performs as well as **TC**. **TC** performs poorly on the benchmarks `gcc` and `crafty` since both these benchmarks have a large number of frequently executed traces, whereas **MS**, which uses cache space more efficiently, performs well. **TC** performs comparably to **MS** on both these benchmarks if the trace cache size is increased.

Performance decreases as the width of the fetch unit decreases, especially when the fetch unit is narrower than four. Multiple narrow fetch units rely on being able to predict future traces ahead of time, and the probability of misprediction increases as traces are predicted further into the future. For example, eight two-wide fetch units would be able to maintain instruction supply only if it were possible to accurately predict the next eight traces at all points in the program. If the trace prediction accuracy is 95%, the eighth trace has a one in three ($1 - 0.95^8 = 0.34$) chance of being mispredicted.

The integer benchmarks show more benefit from high bandwidth fetch than floating point programs. Improving instruction fetch bandwidth does not help floating point programs since they are usually limited by large instruction latencies (cache misses, floating point operations).

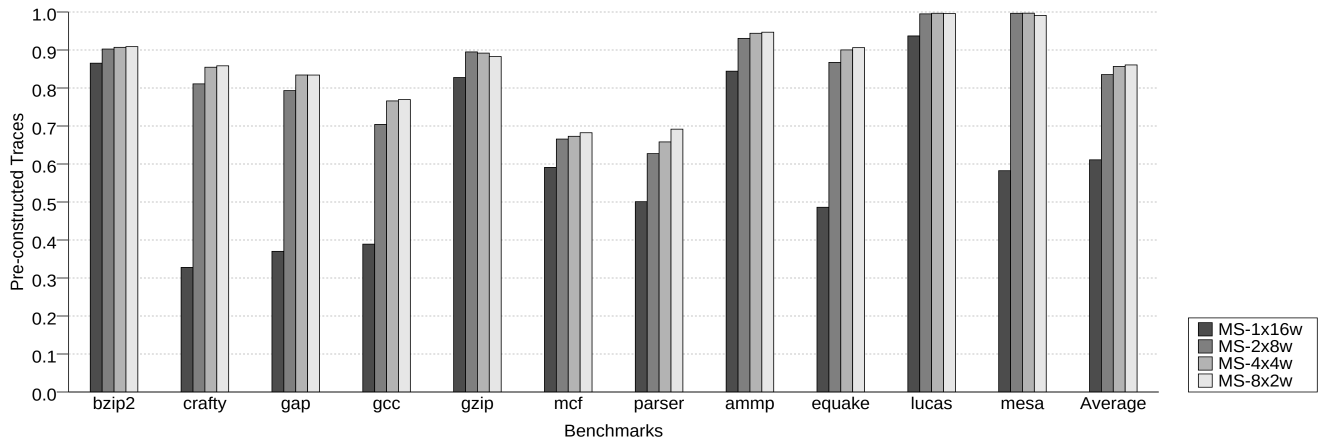

Figure 6. Pre-constructed traces

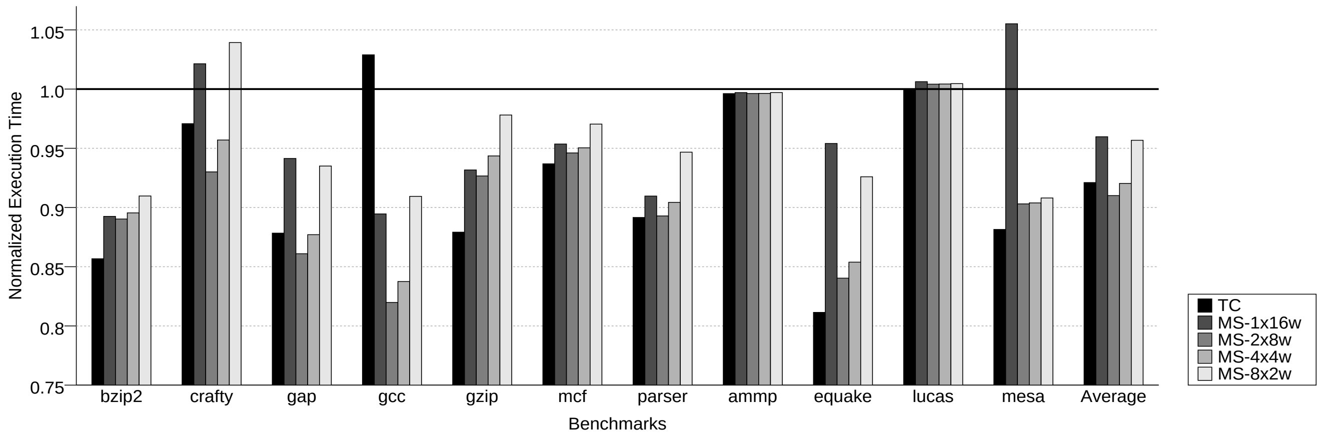

Figure 7. Performance

3.3 Instruction Cache Size

Figure 8 shows the performance of **W16**, **TC** and **MS** over a range of instruction cache sizes. The figure plots execution time normalized to **W16** with a 64K instruction cache. **TC** suffers the most as the cache becomes smaller, since efficient use of cache space is more critical with small caches. **W16** outperforms **TC** at cache sizes of less than 16K.

The four **MS** schemes provide the most robust performance, slowing down less than 10% even when the cache is one-eighth in size. Multiple sequencers are able to utilize the available cache space more efficiently since they do not have the storage overheads associated with trace caches. In case an L1 cache miss does occur, they are better at tolerating the miss latency since other instructions can be fetched while the miss is handled, and multiple misses can be overlapped with each other.

MS-1x16w behaves as robustly as the other **MS** schemes, even though it cannot fetch multiple traces in parallel. This suggests that it is the ability to initiate multiple cache misses in parallel that is the important factor in tolerating small cache sizes.

4 Related Work

Stark et al. [19] proposed a limited form of out-of-order instruction fetch for tolerating instruction cache misses, and proposed several ways of out-of-order renaming. Unlike the technique described in this paper, instruction fetch proceeded normally during most of the execution. Instructions were fetched out-of-order only on cache misses.

Trace preconstruction was proposed by Jacobson et al. [8] to decrease the number of trace cache misses on programs with large working sets. Their focus was constructing a set of traces well ahead of the current trace so that when control reached that point in the future it would suffer no trace cache misses. Their technique maintains a stack containing entries corresponding to the

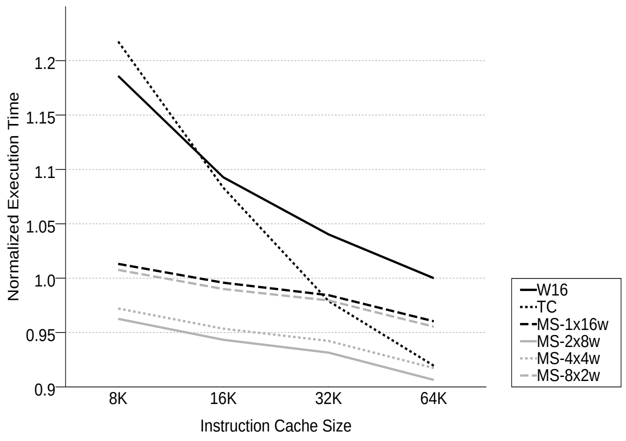

Figure 8. Performance variation with I-cache size

hierarchical structure of functions and loops in the program. The stack is used to identify potential preconstruction points many cycles ahead of when they are needed. In contrast, our scheme uses a standard trace predictor to make predictions and tries to stay just a little ahead of the processor. These two factors are related: the reason we are able to use a standard trace predictor is that we only predict control flow in the immediate future. Unlike the aim of trace preconstruction—prefetching potential trace cache misses—our scheme takes the idea to its limit by making the cache very small and constructing all the traces just before control flow reaches them.

Fetch Target Queue [14] was proposed by Reinman et al. to decouple instruction fetch from the rest of the execution pipeline. Their scheme predicts targets of future branches in advance of when the branches are fetched and inserts the target addresses in the fetch target queue. This queue can then be used for prefetching cache blocks that are not present in the level one instruction cache. However, the fetch bandwidth of the processor is still limited by the density of taken branches in the instruction stream.

Speculative multithreading architectures like Multiscalar [1,18] come closest to this technique as far as the nature of instruction fetch is concerned. They typically consist of multiple execution cores, each of which has a fetch unit and a trace/task predictor that assigns traces to cores. Since each execution core fetches instructions it needs by itself, instructions are fetched as and when they are needed, in an order different from program order. The technique proposed in this paper decouples the decision to build clustered fetch units from the decision to build clustered execution cores.

Another approach to high bandwidth fetch is changing the code layout to correspond more closely to the desired fetch order. Ramirez et al. proposed a profile based compiler optimization called a *Software Trace Cache* [13] that rearranges basic blocks in the program so that the instruction cache stores continuous traces of instructions, just like a trace cache. Their results show that the best performance is achieved by a combination of both the hardware and software trace cache.

5 Conclusions and Future Directions

High bandwidth instruction fetch is essential for building high performance processors. Conventional instruction fetch techniques are difficult to scale up to provide this extra bandwidth since the fetch unit needs to be redirected on each taken branch. Trace caches are a brute force solution to this problem. They are capable of supplying instructions at a very high rate but are expensive in terms of their area requirements since they utilize cache space inefficiently.

Sequencing through the program at the granularity of traces and fetching multiple traces simultaneously by using a replicated fetch unit can be used to get the best of both worlds: fetch bandwidth of a trace cache, and the storage efficiency of an instruction cache. Trace-granularity sequencing decouples the fetch of different parts of the program from each other, and this decoupling enables parallelizing instruction fetch by using multiple sequential instruction sequencers. We described the design of such a fetch unit in detail and demonstrated that it is capable of achieving similar fetch bandwidth to a trace cache, and, at the same time, decreasing the number of instructions fetched from the instruction cache.

Our results also suggest that multiple sequencers are more resilient to larger I-cache miss rates than a trace cache. An important area of future work is evaluating this mechanism in the context of future technology trends like variable latency caches, longer access times and power consumption restrictions. We expect that multiple sequencers will turn out to be a good fit for the requirements of future processors.

Fetching instructions out of order is only half the battle, since even when instructions are fetched out of order they simply wait in a buffer for instructions before them to be fetched before they can be executed. In the future, we plan to relax this restriction as well by renaming instructions out-of-order and issuing them to execution units without requiring all prior instructions to be fetched.

6 Acknowledgements

We would like to thank Adam Butts, Brian Fields, Manoj Plakal, and Ravi Rajwar for commenting on drafts 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.

7 References

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

[2] 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.

[3] 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.

[4] 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.

[5] T. Heil and J. E. Smith. Selective Dual Path Execution. Technical report, University of Wisconsin-Madison, Nov. 1996.

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

[7] 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.

[8] Q. Jacobson and J. E. Smith. Trace Preconstruction. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 37–46, Vancouver, British Columbia, June 12–14, 2000.

[9] J. Kin, M. Gupta, and W. H. Mangione-Smith. The Filter Cache: An Energy Efficient Memory Structure. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 184–193, Research Triangle Park, North Carolina, Dec. 1–3, 1997.

[10] M. J. Knieser and C. A. Papachristou. Y-Pipe: A Conditional Branching Scheme without Pipeline Delays. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 125–128, Portland, Oregon, Dec. 1–4, 1992.

[11] 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.

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

[13] 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.

[14] G. Reinman, B. Calder, and T. M. Austin. Optimizations Enabled by a Decoupled Front-End Architecture. *IEEE Transactions on Computers*, 50(4):338–355, Apr. 2001.

[15] R. Rosner, A. Mendelson, and R. Ronen. Filtering Techniques to Improve Trace-Cache Efficiency. In *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques*, Barcelona, Spain, Sep. 8–12, 2001.

[16] 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.

[17] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. E. Smith. Trace Processors. In *Proc. 30th International Symposium on Microarchitecture*, pages 138–148, Dec. 1997.

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

[19] 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.

[20] S. Vajapeyam and T. Mitra. Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 1–12, Denver, Colorado, June 2–4, 1997.

[21] 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.

[22] C. B. Zilles and G. S. Sohi. Execution-Based Prediction Using Speculative Slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 2–13, Göteborg, Sweden, Jun. 30–July 4, 2001.