A Low-Complexity, High-Performance Fetch Unit for Simultaneous Multithreading Processors

Ayose Falcón Alex Ramirez Mateo Valero

Computer Architecture Department
Universitat Politècnica de Catalunya — Barcelona, Spain
{*afalcon,aramirez,mateo*}*@ac.upc.es*

Abstract

Simultaneous Multithreading (SMT) is an architectural technique that allows for the parallel execution of several threads simultaneously. Fetch performance has been identified as the most important bottleneck for SMT processors. The commonly adopted solution has been fetching from more than one thread each cycle. Recent studies have proposed a plethora of fetch policies to deal with fetch priority among threads, trying to increase fetch performance.

In this paper we demonstrate that the simultaneous sharing of the fetch unit, apart from increasing the complexity of the fetch unit, can be counterproductive in terms of performance. We evaluate the use of high-performance fetch units in the context of SMT. Our new fetch architecture proposal allows us to feed an 8-way processor fetching from a single thread each cycle, reducing complexity, and increasing the usefulness of proposed fetch policies.

Our results show that using new high-performance fetch units, like the FTB or the stream fetch, provides higher performance than fetching from two threads using common SMT fetch architectures. Furthermore, our results show that our design obtains better average performance for any kind of workloads (both ILP and memory bounded benchmarks), in contrast to previously proposed solutions.

1. Introduction

Simultaneous Multithreading (SMT) [23, 24] is a technique that allows instructions from many applications or threads to coexist in each stage of the processor pipeline. Processor resources are shared among several tasks each cycle, taking advantage not only from the existing ILP of each thread but also from TLP among them. The higher throughput of instructions obtained increases the overall SMT performance. In case that a thread is stalled because of a branch misprediction or a cache miss, processor resources can be used efficiently by instructions from other threads.

As is the case with superscalar processors, the SMT fetch unit is one of the most important parts of the processor. Instruction supply to the following stages of the processor depends on it. The presence of branches and I-cache misses in the instruction flow cause fetch stalls that affect the whole processor. In spite of the important role of the SMT fetch unit, current implementations are far from optimal performance. Hence, the fetch unit is currently the most important bottleneck of an SMT processor [1, 22].

Fetching from a single thread seems to be insufficient to feed an 8-way execution core. Solutions that try to widen the fetch throughput are targeted to fetch from multiple threads in a single cycle [6, 12, 22]. A priority policy is used to assess which thread should be fetched first. When no more instructions can be fetched from this highest priority thread, then the second highest priority thread is fetched. This process continues until the instruction fetch limit is reached or no more threads are available.

However, fetching from multiple threads is not a trivial issue. Replicating the fetch unit to have an individual engine for each thread is too complex and too expensive. Thus, the most significant changes required to convert a superscalar processor into an SMT take place in the fetch stage. First, a program counter must be maintained for each thread. Second, multiple branch predictions (one per thread) are needed, with a branch predictor port devoted for each thread. Also, in order to improve the branch prediction accuracy, a return address stack and a branch history register are needed for each thread. Third, multiple I-cache accesses must be done in a single cycle and a memory port for each thread is necessary. The cache should also be multi-bank to reduce bank conflicts. After these cache accesses are finalized, instructions fetched from each thread must be aligned and joined in a single packet. All these modifications, especially the inclusion of more ports, seriously affect the implementation of the processor, both in terms of layout and cycle time [2, 3].

In this paper we evaluate current SMT fetch implementations and propose a new implementation targeted to solve the two main problems of the SMT fetch unit: high complexity and low performance. We demonstrate that it is possible to maintain a high fetch throughput without needing to simultaneous share the fetch unit among threads. Our proposal consists of a fetch architecture which is capable of fetching enough instructions from a single thread without reducing performance and allows for the implementation of simpler fetch policies.

This paper is organized as follows: Section 2 discusses related work. Section 3 describes SMT fetch architectures needed to fetch from one thread and from many threads, as well as the high-performance, low complexity fetch units we propose for SMT. Section 4 describes our experimental methodology and Section 5 present simulation results. Finally, Section 6 concludes the paper.

2. Related Work

Simultaneous Multithreading [23, 24] is a technique able to exploit task-level parallelism by issuing instructions from multiple threads in the same cycle. Sharing resources dynamically among all threads reduces the probability that a functional unit is idle for several cycles.

In a later paper, Tullsen et al. [22] evaluate a realistic SMT implementation and highlight two main bottlenecks in an SMT processor: fetch bandwidth and branch predictor accuracy. To fill all fetch slots, the SMT architecture is extended to fetch from several threads each cycle. Also, some policies to give a priority to threads, such as *Round-Robin* or *ICOUNT*, are introduced. However, on the one hand, the presence of taken branches in the instruction flow limits the amount of instructions that can be fetched from a single thread. On the other hand, branch mispredictions caused by an inefficient branch predictor imply that many instructions fetched must be later discarded. Burns et al. [1] also detect fetch fragmentation and branch prediction accuracy as the key factors in increasing the general SMT performance.

We propose a new SMT fetch architecture that attacks the two main sources of performance degradation identified in previous papers and that is able to maintain a continuous flow of instructions to feed the rest of the pipeline.

Fetch policies have been proposed to reduce the number of wrong-path instructions in the pipeline. Luo et al. [11] use a confidence estimator to find threads that are likely to be in the correct path. A thread with many low-confidence branches is given a lower priority and, once the low confidence branches have been resolved, the priority of the thread is increased. If the number of unresolved low confidence branches overcomes a given threshold, the thread is stalled. Knijnenburg et al. [9] employ a branch classifier to filter out difficult to predict branches. When a branch classified as difficult is encountered, the thread is stalled until the branch is resolved.

Both the stream predictor [16] and the gskew predictor [14] we use provide high prediction accuracy, which implies that a high percentage of fetched instructions are from the correct path. Moreover, these fetch architectures could be used in conjunction with fetch policies based on branch predictability to increase SMT performance.

High performance fetch is also necessary to achieve a high overall performance in traditional superscalar processors. Fetch bandwidth is limited by the I-cache hit rate, branch prediction accuracy, and the presence of taken branches in the instruction flow. Several techniques have been proposed to overcome these limits, by fetching non-continuous instruction sequences in a single cycle. These techniques need multiple branch predictions to increase branch predictor bandwidth, a complex or special purpose cache to increase instruction bandwidth, and some kind of merging mechanism to join all instructions in a single block.

The *Branch Address Cache* [25] employs a multi-bank BTB (called BTAC) for supplying fetch addresses of predicted blocks. An extended two-level branch predictor is used to predict multiple branches per cycle. Although this fetch unit was initially designed as a two-stage unit, it requires a complex alignment network after the interleaved I-cache access that will probably affect the cycle time or require a new pipeline stage.

The *Collapsing Buffer* [5] employs a multiple branch predictor and an interleaved BTB to predict multiple blocks per cycle. A devoted buffer is used to merge groups of instructions separated by a taken branch. A significant amount of logic is needed after accessing the BTB and after accessing the multi-bank I-cache, which enlarges the fetch pipeline with one or two stages.

The *Trace Cache* [18] solves cycle time problems of previous techniques by moving part of the logic out of the critical path. A special purpose cache stores sequences of dynamic instructions collected by the trace fill unit at the back end of the pipeline. A multiple branch predictor (Trace Predictor) is also used to generate indexes to access the Trace Cache. If there is a miss, a core fetch unit with an interleaved cache is used to provide instructions. This scheme effectively avoids cycle time problems, but at the cost of extensive additional hardware.

Oberoi and Sohi [15] propose a mechanism for out-of-order fetching in superscalar processors. It uses multiple sequencers that prefetch instructions into a pool of buffers, from which they are taken to rename.

Although the techniques discussed above can be alternative solutions to the fetch bandwidth problem of an SMT processor, their complexity makes them inefficient to implement. The fetch target buffer (FTB) [17] and fetching instruction streams [16] are good techniques to increase the fetch bandwidth of superscalar processors with low complexity. In this paper, we show that they also help to reduce the complexity of the SMT fetch as well as maintaining high performance. Furthermore, we show that they also allow for the implementation of fine grain fetch policies.

3. Fetch Architectures for SMT

In this section we examine current fetch architectures for SMT. We study a fetch unit design which is necessary to implement policies that fetch only from one thread each cycle, as well as a fetch unit design for policies that try to fetch from more than one thread (we will study those that fetch from a maximum of two threads). We present results for a typical fetch engine for SMT which uses a *gshare* [13] branch predictor plus a BTB [10].

Next, we present two high-performance fetch engines: a *gskew* [14] branch predictor plus an FTB [17] design, and a fetch configuration based on instruction streams [16].

3.1. Fetching from a Single Thread

The most basic implementation of an SMT fetch unit is the one that fetches from only one thread each cycle, denoted by *1.X fetch policy*: up to *X* instructions from *1* thread. The implementation of this fetch configuration is depicted in Figure 1. There is a fine-grained, non-simultaneous sharing of the fetch unit which implies a simple fetch design, similar to that of a superscalar processor. The instruction cache does not need any modification. As there are no multiple simultaneous accesses, there is no need to have a multibanked cache to avoid bank conflicts. Also, a simple single-port instruction cache can be used because no more than one thread will access the cache each cycle.

There is only one addition to a superscalar fetch design: a fetch policy is necessary to decide which thread should be fetched each cycle, among those threads that are not blocked due to an instruction cache miss. The easiest way is to do this in a *Round Robin* [22] fashion, although better policies can be applied, like ICOUNT [22].

The main problem of this fetch architecture is that a single program/thread is not enough to fully use the available fetch bandwidth. On the one hand, branch predictors that predict only one branch per cycle limit the fetch bandwidth to 6-8 instructions per cycle, the typical size of a basic block in integer codes. On the other hand, instructions are sometimes in non-contiguous cache locations, hence fetching from a cache line or from continuous cache lines does not provide sufficient instructions to fill the fetch bandwidth required by an 8-way SMT processor. Therefore, it is hard to achieve overall high performance by using a fetch unit that is limited to one branch prediction and one instruction cache line per cycle.

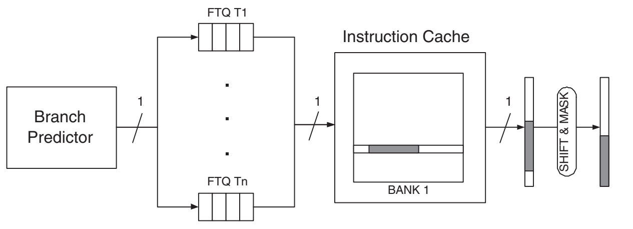


Figure 1. Fetch architecture for a *1.X* fetch policy

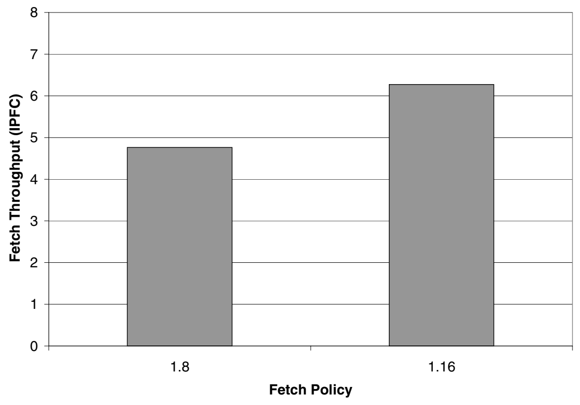


Figure 2. Fetch throughput (in instructions per fetch cycle) using ICOUNT fetching only from one thread each cycle (*gzip-twolf*)

As an example, Figure 2 shows simulation results for a two-thread workload of one thread with high ILP (*gzip*) and one memory bounded thread (*twolf*) using the ICOUNT fetch policy and the gshare+BTB fetch unit (simulation parameters are described in Section 4). Fetch throughput is measured in instructions per fetch cycle (IPFC), i.e., the average number of instructions provided by the fetch unit on every fetch request. Experiments with different workloads and number of threads show similar trends.

As can be seen, this fetch implementation is far from optimal. With a fetch bandwidth of 8 instructions per cycle, it has a fetch performance less than 5 instructions per cycle. Our results show that gshare+BTB provides more than 4 instructions only 60% of the fetch cycles. It is able to provide 8 instructions (and, therefore, fully use the fetch bandwidth) only 31% of the fetch cycles. Overall, the available fetch width is heavily underused.

Furthermore, when increasing the fetch bandwidth to 16 instructions, this fetch architecture is still unable to use half the fetch bandwidth. Our results show that gshare+BTB provides more than 8 instructions only 32% and 16 instructions only 6% of the fetch cycles. Hence, it is clear that increasing the fetch bandwidth beyond the size of a basic block is not an effective solution to achieving a high fetch performance when using fetch architectures that predict individual branches.

Clearly, this architecture can not provide a high fetch performance (8 or 16 instructions per cycle) if it is limited to a single thread.

3.2. Fetching from Multiple Threads

The common solution adopted in SMT to overcome the fetch bandwidth problem is to fetch from multiple threads each cycle [22]. Figure 3 shows the fetch architecture necessary to implement a *2.X fetch policy* (up to *X* instructions from *2* threads). Fetching simultaneously from two cache lines and then merging instructions into a single long line reduces the fetch bottleneck and increases performance.

However, a fetch design using a policy that fetches from two threads simultaneously is hard to implement [2, 8, 22] and requires additional hardware and complexity (emphasized with grey boxes in Figure 3).

With regard to the *branch predictor*, it must provide as many predictions as the number of threads to be fetched in a cycle. A branch predictor port is needed for each thread.

With regard to the *instruction cache*, it requires a port for each thread to fetch a cache line. It requires multiple banks to avoid, or at least to diminish, bank conflicts due to concurrent access. A common way is to implement it as a multiport cache with interleaved banks [20]. Bank-conflict logic must be added before the cache access to avoid different threads accessing the same bank in the same cycle. Moreover, the cache must be non-blocking, so there must be a MSHR (*Miss Status Holding Register*) for each thread. All these changes in the I-cache imply an increase in the number of decoders, read amplifiers, multiplexers, and registers.

With regard to *the logic after the cache access*, it is necessary to perform a mask operation to isolate the correct instructions and a shift operation to align instructions. These operations must be replicated, one per fetched thread. Afterward, individual cache lines are merged in a single cache line that will be passed to the decode stage. Implementing this realignment network is not trivial and it will probably impact the cycle time or the pipeline depth.

Figure 4 shows fetch throughput results when fetching from two threads simultaneously. There is a noticeable improvement in the fetch throughput compared to a 1.X fetch policy (light bars in Figure 4, taken from Figure 2). As more threads are fetched, there are more possibilities to fill the fetch bandwidth totally.

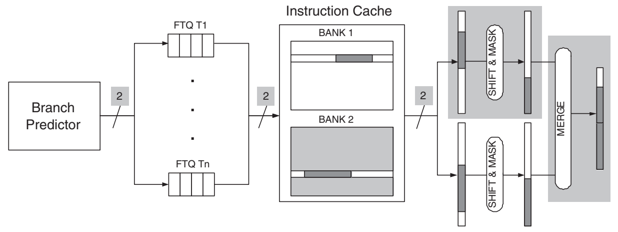


Figure 3. Fetch architecture for a *2.X* fetch policy

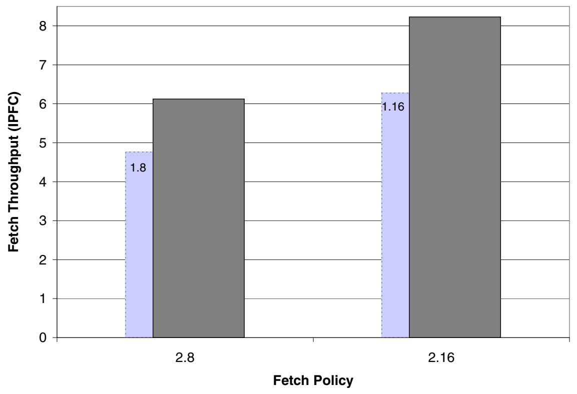


Figure 4. Fetch throughput (in instructions per fetch cycle) using ICOUNT fetching from up to two threads each cycle (*gzip-twolf*)

On the one hand, fetching from two threads up to 8 instructions (2.8) provides a 28% improvement of fetch throughput over fetching from one thread (1.8). For 2.8, gshare+BTB provides 8 instructions 54% of the fetch cycles and more than 4 instructions 80% of the fetch cycles.

On the other hand, extending the fetch width to 16 instructions, fetching from two threads (2.16) gives a 33% improvement of fetch throughput over fetching from only one thread (1.16). With the 1.16 policy it is very difficult to provide 16 instructions in a cycle (less than 9% of the fetch cycles). For 2.16, gshare+BTB provides 16 instructions 16% and more than 8 instructions 46% of the fetch cycles.

These results show that fetching from two threads each cycle does reach a relatively good fetch performance, but at the expense of an increased implementation cost.

3.3. High Performance Fetch Engines

In order to increase SMT fetch performance, techniques attempt to increase fetch throughput by fetching from several threads simultaneously. However, the complexity of fetching from more than one thread can be avoided if a single thread were able to provide sufficient instructions to fill the fetch bandwidth.

In the Related Work Section we have discussed solutions to deal with the problem of low fetch bandwidth in the context of superscalar processors. All of them are complex implementations based on having a multiple branch predictor and a special purpose cache to overcome the limits of a single cache line. In this paper, we propose a new SMT fetch unit implementation based on two different branch predictors that provide high performance with low complexity: gskew+FTB and the stream front-end.

First, Reinman et al. [17] propose the fetch target buffer

(FTB), which increases the effective fetch width provided by a BTB by ignoring some non-taken branches. Potentially, a fetch block provided by an FTB can contain several non-taken branches. Hence, fetch blocks provided by an FTB are larger than fetch blocks provided by a classical BTB. In conjunction with the FTB, we employ a *gskew* [14] branch predictor to predict conditional branches. This branch predictor improves two-level branch predictors by decreasing conflict aliasing in the prediction tables. Branches are stored in three prediction tables which are accessed using three different indices. Taking the prediction of each table, a majority vote selects the branch prediction in order to diminish the effect of a wrong prediction due to conflict aliasing in any of the tables.

The gskew+FTB fetch unit provides higher fetch performance than that provided by a gshare+BTB fetch unit. Our numbers show that gskew+FTB obtains a 5% performance improvement on average over a gshare+BTB on a superscalar processor.

Second, Ramirez et al. [16] propose a fetch architecture based on fetching instruction streams. A stream is a dynamic sequence of instructions, from the target of a taken branch to the next taken branch. It can contain several non-taken branches and thus many basic blocks. A stream predictor is used to predict which stream should be executed next: given a starting address and some path information, it provides the number of sequential instructions until the next taken branch, and the target of this taken branch.

The stream fetch gives high performance for superscalar processors, providing on average an 11% performance improvement over a gshare+BTB and a 5.5% improvement over gskew+FTB. It is only 1.5% lower than using a trace cache mechanism, but with much lower complexity.

Our goal is to reduce the SMT fetch complexity by employing new fetch unit designs which provide better fetch performance. If we could obtain a single-thread throughput that would almost fill the fetch bandwidth, there would be no need to implement complex fetch designs to fetch from two or more threads per cycle. In our case, the fetch implementation would correspond to the 1.X policy shown in Figure 1, instead of to the more complex 2.X policy shown in Figure 3.

4. Simulation Setup

We use SPECint2000 for our simulations. Benchmarks are compiled on a DEC Alpha AXP-21264 using Compaq's C/C++ compiler with '*-O2*' flag. Additionally, code layout is optimized using *spike* [4] with profile data obtained from executing the *train* input set. Due to the large simulation time of SPEC2000, traces of the most representative 300 million instruction slices have been collected, following the idea presented in [19]. Table 1 shows the *ref* input sets and the amount of fast-forward applied during trace collection.

	Ref input	Fast-fwd (billions inst.)	Avg. BB size (insts.)
164.gzip	graphic	68.1	11.02
175.vpr	place	2.1	9.68
176.gcc	166.i	15	5.76
181.mcf	inp.in	43.5	3.92
186.crafty	crafty.in	74.7	9.24
197.parser	ref.in	83.1	6.37
252.eon	cook	57.6	8.73
253.perlbmk	splitmail.535	45.3	10.06
254.gap	ref.in	79.8	9.16
255.vortex	lendian1.raw	58.2	6.50
256.bzip2	inp.program	51.3	10.02
300.twolf	ref	324.3	8.00

Table 1. SPECint2000 characteristics

Table 2 shows the workloads used in our simulations. We have used workloads including 2, 4, 6, and 8 threads. Workloads are classified according to the characteristics of the included benchmarks: with high instruction-level parallelism (ILP), with bad memory behaviour (MEM), or a mix of benchmarks with high ILP and bad memory behaviour (MIX). Due to the characteristics of SPECint2000, with few benchmarks that are really memory bounded, a MEM workload is only feasible for 2 and 4 threads.

The simulator used is a modified trace-driven version of SMTSIM [23]. Our simulator permits execution along wrong paths by having a separate basic block dictionary in which information of all static instructions is contained.

We have modified the fetch stage of the simulator, by decoupling it in two stages, a prediction and a fetch stage. Therefore, the pipeline depth of the simulator is incremented from 8 to 9 stages. Previous work has shown that a decoupled fetch implementation is beneficial not only for superscalar processors [17], but also for SMT [7]. Fetch target queues (FTQs, one per thread) provide latency tolerance between the branch predictor and the fetch unit. The branch predictor generates fetch requests for each thread which are stored in the FTQs, and the fetch unit takes requests from FTQs to drive I-cache accesses. Note that all the configurations presented in this paper implement a decoupled fetch, and no configuration obtains a special advantage from its use.

Workload	Benchmarks
2_ILP	eon, gcc
2_MEM	mcf, twolf
2_MIX	gzip, twolf
4_ILP	eon, gcc, gzip, bzip2
4_MEM	mcf, twolf, vpr, perlbmk
4_MIX	gzip, twolf, bzip2, mcf
6_ILP	eon, gcc, gzip, bzip2, crafty, vortex
6_MIX	gzip, twolf, bzip2, mcf, vpr, eon
8_ILP	eon, gcc, gzip, bzip2, crafty, vortex, gap, parser
8_MIX	gzip, twolf, bzip2, mcf, vpr, eon, gap, parser

Table 2. Mutithreaded Workloads

Fetch Width	8/16 instr.
Fetch Policy	ICOUNT
Fetch Buffer	32 instr.
Dec. & Ren. Width	8 instr.
Gshare Predictor	64K-entry, 16 bits history
Gskew Predictor	3 x 32K-entry, 15 bits history
BTB/FTB	2K-entry, 4 way associative
Stream Predictor	1K-entry, 4 way + 4K-entry, 4 way DOLC index: 16-2-4-10
RAS ⋆	64-entry
FTQ size ⋆	4-entry
Functional Units	6 int, 4 ld/st, 3 fp
Instruction queues	32-entry int, 32-entry ld/st, 32-entry fp
Reorder Buffer ⋆	256-entry
Physical Registers	384 int and 384 fp
L1 I-Cache	32KB, 2-way, 8 banks
L1 D-Cache	32KB, 2-way, 8 banks
L2 Cache	1MB, 2-way, 8 banks, 10 cyc.
Line size	64 bytes
TLB	48-entry I + 128-entry D
Main Memory latency	100 cycles

Table 3. Simulation parameters (resources marked with ⋆ are replicated per thread)

In our experiments, we adopt the ICOUNT fetch policy [22]. ICOUNT gives priority to threads according to the number of instructions in the decode, rename and dispatch stages of the processor. Its aim is to balance resources among the threads by prioritizing threads with the fewest number of active instructions in the pipeline. It is important to observe that in our fetch configuration, the fetch policy selects the FTQ from which requests should be taken for fetching (recall Figures 1 and 3). Therefore, the branch predictor has to generate a fetch request for the thread selected by ICOUNT in a previous cycle, which is stored in its corresponding FTQ.

We provide simulation results obtained by three different fetch architectures: a standard SMT fetch unit formed by a *gshare* [13] branch predictor plus a *BTB* [10], and two enhanced SMT fetch units, namely, a *gskew* [14] branch predictor plus a *FTB* [17], and the stream fetch unit [16].

The baseline configuration is shown in Table 3. We have selected branch predictor sizes that require a hardware budget of approximately 45 KB. In case that the branch predictor requires multiple tables or mechanisms (direction and target separately), the total hardware budget is this size.

It is important to highlight that, as the decode width is limited to 8 instructions, overall performance values can not exceed 8 instructions per cycle. Instructions provided by the fetch unit are stored in an intermediate fetch buffer where they remain until they proceed to the decode stage. If the decode stage is stalled and the fetch buffer fills up, fetch is also stalled until room is available in the fetch buffer to allocate new instructions. That implies that there is no intrinsic gain from widening the fetch bandwidth beyond the decode width limit.

5. Simulation Results

In this section we present results obtained with the stream and the gskew+FTB fetch engines for SMT, comparing them to the extensively used gshare+BTB SMT fetch unit. We have found that the influence of the fetch unit performance on the overall SMT throughput differs depending upon benchmark characteristics. For this reason, we first analyze the results obtained when simulating only benchmarks with high ILP. Next, we analyze the results obtained when memory bounded benchmarks are included.

For each experiment, we provide numbers of fetch performance, denoted by *Fetch Throughput* (measured in Instructions Per Fetch Cycle, *IPFC*), and overall SMT performance, *Commit Throughput* (measured in Instructions Per Cycle, *IPC*).

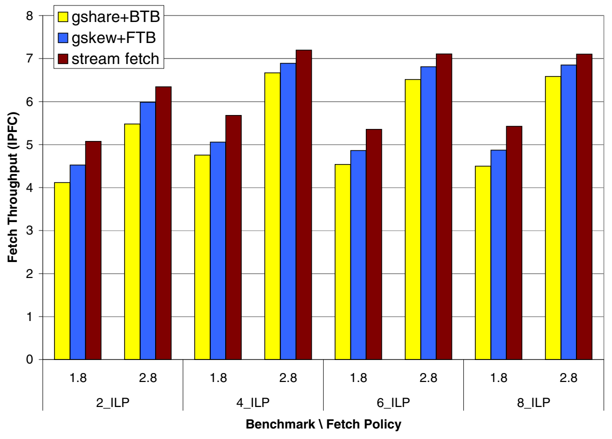
5.1. SMT executing *ILP* benchmarks

Figure 5 shows the IPC results obtained from workloads containing only high ILP benchmarks. These benchmarks have few memory problems due to data accesses, as well as large amounts of independent sequential instructions. In this environment, the fetch throughput is the real limiting factor for overall SMT throughput. The higher the number of instructions provided by the fetch unit, the higher the overall performance.

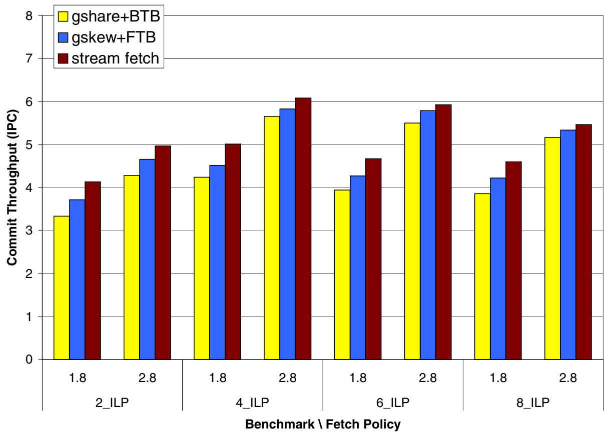
As shown in Figure 5(a), with equivalent hardware resources and fetching from one or two threads every cycle, both gskew+FTB and the stream fetch unit outperform the typical gshare+BTB. Using ICOUNT.1.8, gskew+FTB obtains a 9% average commit improvement over gshare+BTB, while the stream fetch obtains a 10% improvement over gskew+FTB, and a 20% over gshare+BTB. For ICOUNT.2.8, gskew+FTB obtains a 5% improvement over gshare+BTB, while the stream fetch achieves 9% over gshare+BTB and 4% over gskew+FTB. Obviously, as the stream fetch has a better fetch performance, it also provides a better commit performance for this kind of workload, outperforming both gshare+BTB and gskew+FTB in all cases, as shown in Figure 5(b).

Taking the results from Figure 5, it is clear that if several threads with high ILP are simulated, fetching from only one thread is a very limiting factor. The fetch policy itself is not a limiting factor, as it selects from *high-quality* threads, and it is not that important which one is selected. However, a high fetch bandwidth is really important, so the gains obtained using a better fetch unit are more noticeable. Consequently, having a fetch bandwidth of 8 instructions, the best is to fetch from two threads instead of from only one, in an attempt to fill all fetch slots.

However, as we have shown in Section 3, fetching from more than one thread requires much more hardware com-



(a) Fetch throughput



(b) Commit throughput

Figure 5. Throughput using ICOUNT.1.8 vs ICOUNT.2.8 and simulating only ILP benchmarks

plexity that can impact the cycle time. Both stream fetch and gskew+FTB are efficient when providing instructions from only one thread each cycle. However, limiting them to fetch up to 8 instructions per cycle is very restrictive if ILP is high and the fetch unit is able to fetch more than 8 instructions per cycle.

The alternative we propose is to widen the fetch bandwidth to increase fetch throughput instead of fetching from several threads. Widening the fetch bandwidth from 8 to 16 is simple and does not require many changes. Cache lines already contain 16 instructions in our processor configuration and buses until the decode stage are already 16-instruction wide. We only have to modify the hardware to select 16 instructions instead of 8.

Figure 6 shows simulation results for this new configuration. Throughput of the stream fetch unit is higher when we use ICOUNT.1.16 instead of ICOUNT.2.8, achieving a 9% of commit improvement on average. The stream fetch unit provides large fetch blocks that can potentially include several non-taken branches, and limiting the fetch bandwidth to 8 instructions seriously limits its performance.

In contrast, gshare+BTB and gskew+FTB have a lower performance with ICOUNT.1.16 than with ICOUNT.2.8 (9.7% of commit slowdown for gshare+BTB and 4% for gskew+FTB). The reason is that these fetch architectures predict branches individually and therefore limit the fetch bandwidth to one basic block per cycle. FTB diminishes this effect by ignoring some non-taken branches, although it is still unable to provide fetch blocks with a length close to 16 instructions.

The all-in-one solution that is fetching many instructions from many threads (ICOUNT.2.16) provides higher fetch performance for all fetch units, but, of course, at a much higher cost.

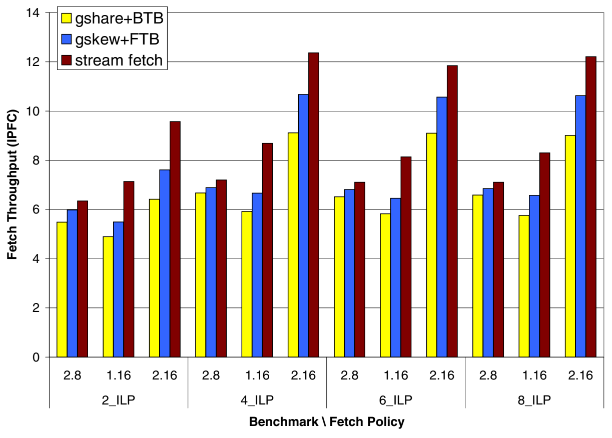
As shown in Figure 6(b) the stream fetch we implement with ICOUNT.1.16 outperforms the other fetch implementations with ICOUNT.2.8 (19% commit improvement over gshare+BTB and 13% over gskew+FTB). Moreover, our performance results with ICOUNT.1.16 are similar to those obtained with the other fetch units and ICOUNT.2.16. Only gskew+FTB outperforms the stream fetch for 4_ILP and 6_ILP by 2.7% and 6.8%, respectively. We consider that this loss in performance is compensated by the smaller hardware budget needed to implement an ICOUNT.1.16 fetch policy versus an ICOUNT.2.16 fetch policy.

We conclude that for ILP workloads the stream fetch using ICOUNT.1.16 is the design offering the best cost/performance trade-off, with higher performance than ICOUNT.2.8 and less complexity than either ICOUNT.2.8 or ICOUNT.2.16.

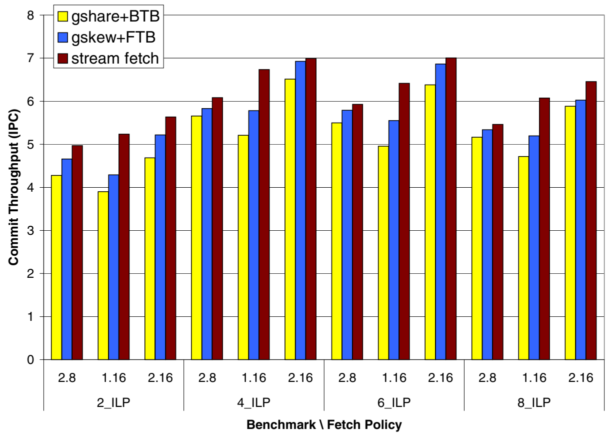
5.2. SMT executing memory bounded benchmarks (*MIX* & *MEM*)

Figure 7 shows simulation results when memory bounded benchmarks are included in the workload. As shown in Figure 7(a), fetch throughput results have the same trend noted in the previous section for ILP benchmarks. The gskew+FTB fetch unit provides better fetch performance than gshare+BTB, while the stream fetch outperforms both gshare+BTB and gskew+FTB. Besides this, fetching from two threads simultaneously gives the opportunity to provide more instructions every cycle, so the fetch throughput increases from ICOUNT.1.8 to ICOUNT.2.8, as expected.

However, the behaviour of workloads including memory bounded benchmarks differs from the behaviour observed previously when only ILP benchmarks were executed. Figure 7(b) shows the overall performance of these workloads.

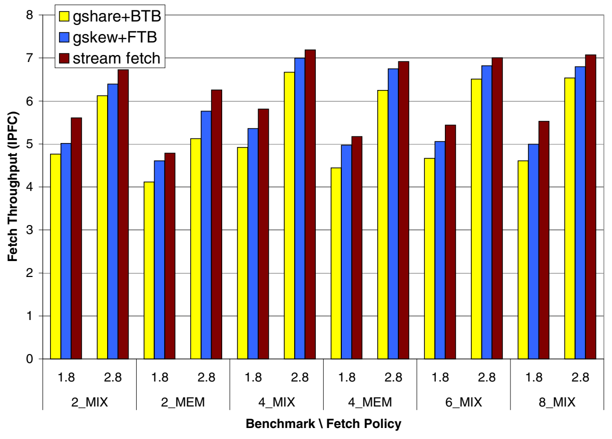


(a) Fetch throughput

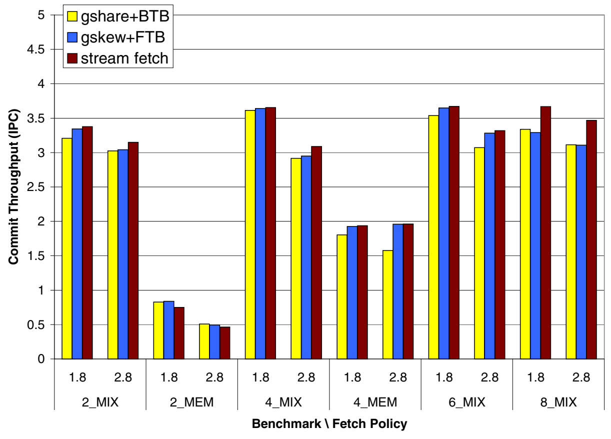


(b) Commit throughput

Figure 6. Throughput using ICOUNT.1.16 vs ICOUNT.2.X and simulating only ILP benchmarks



(a) Fetch throughput



(b) Commit throughput

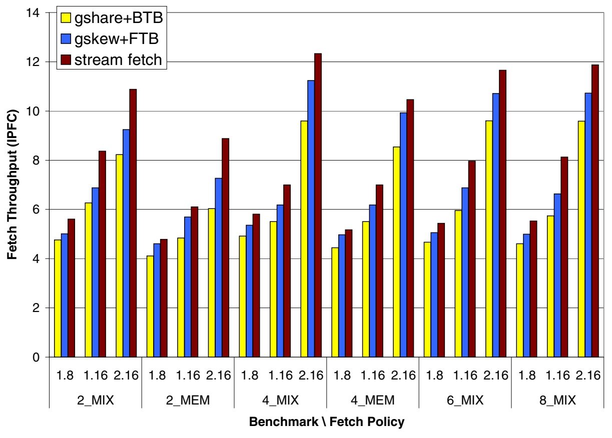
Figure 7. Throughput using ICOUNT.1.8 vs ICOUNT.2.8 and simulating memory bounded benchmarks

Unexpectedly, fetching from two threads actually decreases performance.

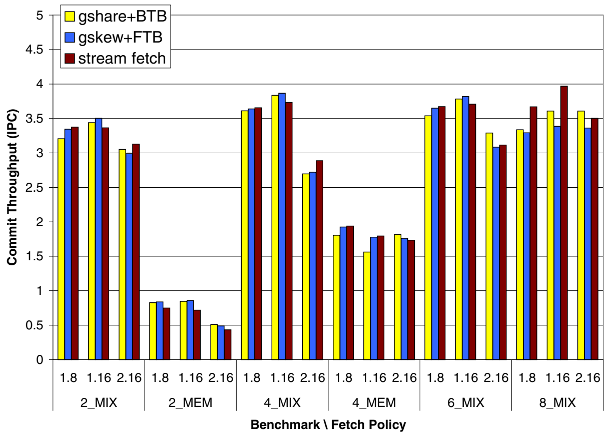
The problem of fetching from several threads using memory bounded benchmarks was also identified in [21]. A single thread with poor cache performance can strangle overall SMT performance, by monopolizing resources that could be exploited by other threads. The solution proposed in [21] is to free resources occupied by a stalled thread by flushing its instructions from the pipeline. Other techniques also propose flushing or stalling threads according to other criteria [6, 9, 11].

As shown in Figure 7(b), the stream fetch unit fetching from only one thread outperforms fetching from two threads for all workloads, even when the fetch width is limited to 8 instructions per cycle. Fetching from only one thread allows to fill the pipeline only with instructions from the thread with less in-flight instructions, which is probably a thread that is not stalled due to a data cache miss. Fetching from a second *low-quality* thread allows this second thread to take over more resources, which will not be available for the other threads until the cache miss is resolved. Our way of solving this problem is to fetch instructions only from the first, highest priority thread to fill all available fetch slots.

The conclusion is that improving fetch unit bandwidth is good if there are no stall situations in the pipeline, as happens with high ILP workloads. Moreover, using a powerful fetch unit inappropriately by fetching from a second, low-quality thread, can be harmful to overall SMT performance,



(a) Fetch throughput



(b) Commit throughput

Figure 8. Throughput using ICOUNT.1.16 vs ICOUNT.1.8 and ICOUNT.2.16, and simulating memory bounded benchmarks

as happens with memory bounded benchmarks.

Figure 8 shows results when the fetch bandwidth is extended to 16 instructions per cycle. We have maintained the bar of ICOUNT.1.8 from Figure 7, as it provides better performance than ICOUNT.2.8. Obviously, fetch performance (Figure 8(a)) benefits from increasing the number of instructions that can be fetched. However, the stream fetch is able to provide more instructions by widening fetch bandwidth to 16 instructions than the other techniques.

As shown previously, fetching from many threads when having memory bounded workloads is not a good solution to improve overall SMT performance. Even ICOUNT.2.16 obtains worse commit performance than ICOUNT.1.16 and ICOUNT.1.8 in almost all cases, as shown in Figure 8(b). The best performance is obtained with ICOUNT.1.16, which combines a wide fetch with a fine grain thread selection. Both gskew+FTB and stream fetch unit using ICOUNT.1.16 obtain a 3 to 4% improvement on average over gshare+BTB using ICOUNT.1.8.

These results show that, although a high-performance fetch unit is necessary to increment the overall SMT performance, a selection scheme is equally important to speed up this performance. The ICOUNT fetch policy tries to share resources equally among threads. In spite of this being a good policy if there are many threads with a high ILP, it can be bad if a thread is stalled for several cycles and is given more and more resources. Therefore, the best solution is to fetch only from a *high-quality* thread, and as much as possible. Our proposal does just that: we implement a fetch unit which can fetch only from one thread (always the most promising, according to the fetch policy), and which is able to provide sufficient instructions to maintain an instruction flow from the fetch onward.

We conclude that both gskew+FTB and the stream fetch increase fetch throughput even when memory bounded benchmarks are simulated. In spite of performance speedups obtained, the potential provided by more effective fetch engines is not totally exploited by the fetch policy (in our case, ICOUNT), which is unable to translate all the potential provided by these fetch units into a higher overall performance. Current fetch policy proposals are based on fetch units fetching from two threads using a gshare predictor with a BTB, which has low fetch performance as we have shown in this paper. Future fetch policy proposals should be targeted to exploiting the fetch potential provided by a high bandwidth fetch unit fetching from a single thread, in order to increment the overall SMT performance.

6. Conclusions

The fetch unit is the most significant obstacle to achieving higher performance in SMT. To alleviate this, the common solution adopted (from the publication of [22] on), has been fetching from several threads in a single cycle.

In this paper, we have shown that implementing a fetch architecture fetching from more than one thread is too expensive, both in terms of cost and complexity. We have demonstrated that a better solution to increment the SMT fetch performance is not to fetch few instructions from several threads, but to fetch many instructions from a single thread. This can be done by implementing more accurate branch predictors, such as a gskew+FTB or a stream predictor employed in this paper. Implementing a fetch unit fetching only from one thread solves the problem of the high-complexity of the SMT fetch unit.

In addition, we have shown that fetching from more than one thread does not always provide better performance. Fetching from several threads in a single cycle can even be counterproductive, because sharing the fetch unit among threads also implies competition among them. In certain situations, this can even lead to a performance loss.

We conclude that the use of a fine-grain, non-simultaneous shared fetch unit for SMT implies a new point of view of the SMT fetch problem. This also implies that some fetch policies proposed so far in the literature have to be revisited to adapt to a new kind of fetch organization.

Acknowledgments

This work was supported by the Ministry of Education of Spain under contract TIC–2001–0995–C02–01 and grant AP2000–3923. We thank Francisco J. Cazorla, Peter M. Knijnenburg, and Oliverio J. Santana for their help and comments on this work.

References

[1] J. Burns and J.-L. Gaudiot. Exploring the SMT fetch bottleneck. In *Procs. of the Workshop on Multi-Threaded Execution, Architecture and Compilation (MTEAC'99)*, June 1999.

[2] J. Burns and J.-L. Gaudiot. Quantifying the SMT layout overhead – does SMT pull its weight? In *Procs. of the 6th Intl. Conference on High Performance Computer Architecture*, pages 109–120, Jan. 2000.

[3] J. Burns and J.-L. Gaudiot. Area and system clock effects on SMT/CMP processors. In *Procs. of the Intl. Conference on Parallel Architectures and Compilation Techniques*, pages 211–218, Sept. 2001.

[4] R. Cohn, D. Goodwin, P. Lowney, and N. Rubin. Spike: An optimizer for Alpha/NT executables. *USENIX Windows NT Workshop*, pages 17–23, Aug. 1997.

[5] T. Conte, K. Menezes, P. Mills, and B. Patel. Optimization of instruction fetch mechanism for high issue rates. In *Procs. of the 22nd Intl. Symposium on Computer Architecture*, pages 333–344, June 1995.

[6] A. El-Moursy and D. Albonesi. Front-end policies for improved issue efficiency in SMT processors. In *Procs. of the 9th Intl. Conference on High Performance Computer Architecture*, pages 31–40, Feb. 2003.

[7] A. Falcón, O. J. Santana, A. Ramírez, and M. Valero. Tolerating branch predictor latency on SMT. In *Procs. of the 5th International Symposium on High Performance Computing (ISHPC-V)*, pages 86–98, Oct. 2003.

[8] A. Klauser and D. Grunwald. Instruction fetch mechanisms for multipath execution processors. In *Procs. of the 32nd Intl. Symposium on Microarchitecture*, pages 38–47, Nov. 1999.

[9] P. M. Knijnenburg, A. Ramirez, F. Latorre, J.-L. Larriba-Pey, and M. Valero. Branch classification to control instruction fetch in simultaneous multithreaded architectures. In *International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA'02)*, pages 67–76, Jan. 2002.

[10] J. Lee and A. Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, 17(1):6–22, Jan. 1984.

[11] K. Luo, M. Franklin, S. Mukherjee, and A. Seznec. Boosting SMT performance by speculation control. In *Procs. of the 15th International Parallel and Distributed Processing Symposium*, Apr. 2001.

[12] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. In *International Symposium on Performance Analysis of Systems and Software*, pages 164–171, Jan. 2001.

[13] S. McFarling. Combining branch predictors. Technical Report TN-36, Compaq Western Research Lab., June 1993.

[14] P. Michaud, A. Seznec, and R. Uhlig. Trading conflict and capacity aliasing in conditional branch predictors. In *Procs. of the 24th Intl. Symposium on Computer Architecture*, pages 292–303, June 1997.

[15] P. Oberoi and G. Sohi. Out-of-order instruction fetch using multiple sequencers. In *Procs. of the Intl. Conference on Parallel Processing*, pages 14–26, Aug. 2002.

[16] A. Ramirez, O. J. Santana, J. L. Larriba-Pey, and M. Valero. Fetching instruction streams. In *Procs. of the 36th Intl. Symposium on Microarchitecture*, pages 371–382, Nov. 2002.

[17] G. Reinman, B. Calder, and T. Austin. Optimizations enabled by a decoupled front-end architecture. *IEEE Transactions on Computers*, 50(4):338–355, Apr. 2001.

[18] E. Rotenberg, S. Benett, and J. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *Procs. of the 29th Intl. Symposium on Microarchitecture*, pages 24–35, Dec. 1996.

[19] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Procs. of the Intl. Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, Sept. 2001.

[20] G. Sohi and M. Franklin. High-bandwith data memory systems for superscalar processors. In *Procs. of the 4th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 53–62, Apr. 1991.

[21] D. Tullsen and J. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *Procs. of the 34th Intl. Symp. on Microarchitecture*, pages 318–327, Dec. 2001.

[22] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Procs. of the 23rd Intl. Symposium on Computer Architecture*, pages 191–202, May 1996.

[23] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Procs. of the 22nd Intl. Symposium on Computer Architecture*, pages 392–403, June 1995.

[24] W. Yamamoto and M. Nemirovsky. Increasing superscalar performance through multistreaming. In *Procs. of the Intl. Conference on Parallel Architectures and Compilation Techniques*, pages 49–58, June 1995.

[25] T.-Y. Yeh, D. Marr, and Y. Patt. Increasing the instruction fetch rate via multiple branch prediction and a branch address cache. In *Procs. of the 7th Intl. Conference on Supercomputing*, pages 67–76, July 1993.