Do Trace Cache, Value Prediction and Prefetching Improve SMT Throughput?

Chen-Yong Cher[1,*], Il Park[1,*], and T.N. VijayKumar

ECE, Purdue University, IN 47907, USA
{chenyong, ilpark, vijay}@ecn.purdue.edu

Abstract. While trace cache, value prediction, and prefetching have been shown to be effective in the single-threaded superscalar, there has been no analysis of these techniques in a Simultaneously Multithreaded (SMT) processor. SMT brings new factors both for and against these techniques, and it is not known how these techniques would fare in SMT. We evaluate these techniques in an SMT to provide recommendations for future SMT designs. Our key contributions are: (1) we identify a fundamental interaction between the techniques and SMT's sharing of resources among multiple threads, and (2) we quantify the impact of this interaction on SMT throughput. SMT's sharing of the instruction storage (i.e., trace cache or i-cache), physical registers, and issue queue impacts the effectiveness of trace cache, value prediction, and prefetching, respectively.

1 Introduction

Simultaneous Multithreading (SMT) has been proposed for improving processor throughput by overlapping multiple threads in a wide-issue superscalar processor. Three techniques which are used to exploit more instruction-level parallelism (ILP) and to improve single-thread performance in superscalar are: 1) trace cache to increase fetch bandwidth, 2) value prediction to break data dependences, and 3) prefetching to hide memory latency. While these techniques have been shown to be effective in the single-threaded superscalar, there has been no analysis of their effectiveness in SMT. which is becoming the microarchitecture of choice for high-performance microprocessors (e.g., Intel's Hyperthreading, Sun's Niagara, IBM's POWER5). Compared to superscalar, SMT brings new factors both for and against these techniques, and it is not known how these techniques would fare in SMT. Some of these techniques are implemented in superscalars today, and they will be included automatically in SMT when the superscalars are converted to SMT. Therefore, it is important to know how they fare in SMT.

This paper fills this important gap by evaluating these techniques in the context of an out-of-order issue SMT and provides recommendations for future SMT designs.

---

[1] Chen-Yong Cher and Il Park completed all the work in this paper during their PhD studies at Purdue University. After graduation, Chen-Yong Cher and Il Park joined IBM TJ Watson Researcher Center. This paper is not affiliated with IBM.

* Current Address: IBM T. J. Watson, P.O. Box 218, Yorktown Heights, NY 10598, USA {chenyong, ilpark}@us.ibm.com

W. Grass et al. (Eds.): ARCS 2006, LNCS 3894, pp. 232 – 251, 2006.
© Springer-Verlag Berlin Heidelberg 2006

Because SMT's goal is to improve throughput, which is also an important performance metric for server-class machines which increasingly use SMT, we evaluate the techniques in terms of processor throughput.

Our novelty is not in the techniques we study, but in their evaluation in the context of SMT. Our key contributions are: (1) we identify a fundamental interaction between the techniques and SMT's sharing of resources among multiple threads, and (2) we quantify the impact of this interaction on SMT throughput. This interaction is the key issue and common theme in our evaluation of the three techniques.

Previous studies showed that trace cache increases fetch bandwidth [22,20,19,8,3]. Trace cache creates traces from dynamic instruction sequences and allows an entire trace to be fetched in one access. A key motivation for trace cache is that increasing fetch bandwidth in superscalar is complicated and involves more than merely using many fetch ports. To utilize multiple fetch ports, superscalar needs multiple branch prediction, which is not straightforward. Implementing multiple branch prediction involves both (1) maintaining high accuracy of prediction and (2) providing multiple, contiguous fetch PCs for the same thread. Trace cache handles these issues effectively and achieves better performance than multiple branch prediction.

Unfortunately, trace cache introduces multiple copies of instructions in different traces, despite the most efficient implementation [3]. This redundancy reduces the effective size of the cache. Increasing the cache size is difficult due to latency, area, and power considerations. This trade-off of space for bandwidth seems reasonable for superscalar because a single thread may not need a large instruction cache. However, SMT needs a larger instruction storage (i.e., trace cache or i-cache) because multiple threads share the storage. In contrast to superscalar, SMT can supply multiple fetch PCs from different threads and utilize extra fetch ports effectively without needing multiple branch prediction. Therefore, it is not clear whether trace cache's trade-off of space for bandwidth will improve SMT throughput.

Value prediction predicts values instead of waiting for long-latency dependences to be resolved, speeding up computation even beyond data-flow limits [16,1,23,17,5]. Prediction accuracy can be increased and the benefit of the technique can be sustained by trading off coverage and predicting only highly-predictable long-latency operations (e.g., cache misses) [5]. In contrast to value prediction in superscalar, SMT simply tolerates L1 misses. Upon L2 misses, SMT squashes the thread [27], releasing the thread's shared resources (i.e., physical registers and issue queue slots), and overlaps the L2 miss with other threads.

Applying value prediction to SMT raises a key but subtle issue related to sharing of registers. Value prediction holds up physical registers even when the prediction is correct! Due to program-order commit, instructions that follow a correctly-value-predicted, long-latency instruction hold up registers even after completing execution. These instructions can release their registers only after the long-latency instruction completes, confirms the prediction, and commits. Building larger register file to alleviate such hold-up is not easy due to latency, area, and power considerations [2,4,18]. While this hold-up of registers may be acceptable for superscalar, it may not be profitable for SMT, in which multiple threads create a higher demand for the shared registers. It is not clear whether SMT throughput is improved more by value-predicting long-latency instructions and holding up registers; or by squashing the instructions and releasing registers so other threads can use the registers and overlap the latency.

Prefetching predicts future memory references and brings data into caches before the data is actually needed [24,14,25,6,10,13,32,9]. Recent proposals for aggressive hardware prefetching, such as Dead-block predictor[13] and its successor Time-Keeping predictor [32], are highly successful even with non-strided access patterns. SMT has two opposing effects on the opportunity available from prefetching. On one hand, because SMT can tolerate cache misses, it may present less opportunity to prefetching. On the other hand, because SMT issues memory references from multiple threads, it increases the pressure on the memory hierarchy and may present more opportunity.

Prefetching in SMT achieves coverage and accuracy comparable to those of a single thread. However, prefetching raises a subtle issue related to sharing of the issue queue. While prefetching into L2 achieves most of the benefit of prefetching into L1 without incurring L1's contention problems for a single thread [9], prefetching only into L2 causes a problem for SMT. Prefetching into L2 converts slow L2 misses into fast L2 hits; however, the L2 hits still miss in L1, resulting in the same L1 misses occurring in fewer cycles. L1 misses clog the issue queue with dependent instructions, even though L1 misses are short. While SMT without prefetching is also clogged for the L1-miss duration, it eventually incurs an L2 miss and squashes the thread [5], unclogging the issue queue to allow other threads to progress. Because prefetching causes L1 misses to occur in fewer cycles, the issue queue is clogged more often with prefetching than without. Thus, even correct prefetching may hurt SMT throughput! Unfortunately, neither removing L1 misses nor circumventing them to avoid the clogging is easy; removing L1 misses by prefetching into L1 is difficult due to L1's high contention [9], which is worse in SMT. Circumventing L1 misses is also difficult because L1 misses are known too late in the pipeline to prevent dependent instructions from entering the pipeline. Thus, in addition to the uncertainty in opportunity, the question of whether issue-queue clogging or latency hiding will impact SMT throughput more is unclear. Table 1 summarizes the trade-offs of the techniques when they are implemented in SMT.

The main results of our simulations using a subset of the SPEC2000 benchmarks are:

- Trace cache, value prediction and prefetching significantly improve single-thread performance. This result agrees with previous papers and validates our implementations.
- Given similar size for the duo of trace cache and backup i-cache as the conventional i-cache, trace cache degrades SMT throughput compared to the conventional i-cache (throughput improves for 2 threads, agreeing with the two-threaded Pentium IV's use of a trace cache). This result shows that trace cache's space-for-bandwidth trade-off hurts SMT. Giving considerable extra size to the trace cache results in the trace cache performing only marginally better than the conventional i-cache, showing that trace cache is not effective in SMT.
- Given a typical number of physical registers, value prediction degrades SMT throughput, showing that holding up registers under value prediction hurts SMT. While value prediction does improve individual threads that have long-latency misses, it does so at the cost of the other threads, defeating SMT's purpose. Using infinite physical registers and perfect confidence prediction results in value prediction performing only marginally better than conventional SMT, showing that value prediction is not effective in SMT.

- For memory-intensive workloads, there is substantial opportunity for prefetching even with many threads, showing that not all long-latency misses can be hidden by SMT. We found that prefetch coverage can be reduced to balance prefetching and issue queue clogging, improving throughput for this workload. For workloads with mixed memory demand, SMT significantly reduces opportunity. Despite reducing the coverage, prefetching slightly degrades throughput for this workload due to issue queue clogging. Like value prediction, prefetching also improves individual threads at the cost of the other threads in this workload, degrading overall throughput.
- Our findings create a new responsibility for the OS: Because the techniques improve single-thread performance, we recommend that the OS disable the techniques when running multi-programmed workload, and enable them for single-threaded workload and for high-priority threads in a multi-programmed workload.

Section 2 gives the background of the techniques. Section 3 describes our methodology. Section 4 shows our results, and Section 5 concludes the paper.

Table 1. Trade-offs of each techniques

	Trace Cache	Value Prediction	Prefetching
Pros	- satisfies SMT's high demand of fetch bandwidth	- breaks the data dependences of individual threads	- fulfills SMT's high demand on memory access
Cons	- causes redundancy in instruction storage while SMT demands high instruction storage capacity - SMT provides high fetch bandwidth without needing multiple branch prediction	- Data dependence delay can be hidden by other threads. - holds up resources even when predictions are correct	- Opportunity may drop because of thread-level parallelism. - may cause issue queue clogging even when prefetches are correct

2 A Brief Background of the Techniques

2.1 Trace Cache

Before trace cache was introduced, Tyson et al. [31] increased fetch bandwidth by predicting multiple branches every cycle with Branch Address Cache. Rotenberg et al. [22] introduced trace cache, and compared it with other high-bandwidth instruction fetch schemes. Others [20,21,8] studied important issues concerning trace cache performance such as partial matching, cache associativity, fill unit, and multiple branch prediction. Patel et al. [19] proposed branch promotion and trace packing for improving trace cache bandwidth. To achieve better utilization of trace cache space, Black et al. [3] suggested the block-based trace cache, which stores pointers to blocks constituting a trace, instead of storing instructions. Any repetition of the traces results in only the pointers being repeated instead of the entire trace, reducing space requirements.

Because we wish to study the effectiveness of trace cache's space-for-bandwidth trade-off, and because the block-based trace cache is the most space-efficient implementation that also achieves high bandwidth, we use the block-based trace cache in our evaluations. We discuss the details of this specific trace cache later in Section 4.1.1.

2.2 Value Prediction

Lipasti et al. [16] proposed last-value prediction with saturating confidence counters. Mendelson et al. [1] added a stride prediction scheme, and Farkas et al. [23] studied the implementation details for the context-based prediction scheme. Others predict load values by using recent store information [17,30]. However, without accurate prediction, value prediction may hurt performance due to misprediction penalties unless there is hardware support, such as selective recovery, to reduce the penalty. Calder et al. [5] showed the importance of confidence prediction to perform selective value prediction to avoid mispredictions and achieve good speedup even without the complicated machinery of selective recovery.

In addition to avoiding complicated selective recovery, reducing mispredictions is important for SMT so that processor resources are not wasted on incorrect execution. Therefore, we use [5]'s selective value prediction, which combines confidence prediction and value prediction, in our evaluations and discuss their details in Section 4.2.1.

2.3 Prefetching

While prefetching can be implemented in either software [24,14] or hardware, we focus on hardware prefetching in this study. Chen et al. [25] proposed the stride prefetcher, and others [11,7] used a stream buffer for prefetching. Markov prefetching uses address correlation (i.e., correlation among addresses in the cache miss stream) to improve the accuracy of prefetching arbitrary, non-strided access patterns [6,10]. These proposals focused on *what* to prefetch but do not pinpoint *when* to prefetch.

Lai et al. [13] first proposed to consider the timing of memory access patterns to determine when to prefetch and improves accuracy over [10]. They introduced Dead-Block Predictors to predict the dead blocks — i.e., the blocks that will be evicted without any use — in L1. When a block dies, the prefetcher predicts the next access to the block's set and prefetches the next access into the dead block's frame. Kaxiras et al. [12] also proposed a scheme to predict dead blocks, but they used the prediction for reducing cache leakage power and not for prefetching data. Hu et al. [32] applied [12] to prefetching and used smaller prediction tables than [13] for both dead-block predicition and next-address prediction while achieving better performance.

Lastly, Hu, et al. [9] simplified [32] by showing that when prefetching into a large highly-associative L2 cache, dead-block prediction was not necessary. [9] also showed that prefetching into L2 can achieve most of the benefit of prefetching into L1 without disrupting the highly contentious L1 with untimely or incorrect prefetches. Because SMT's multiple threads cause even higher contention on L1 than that of superscalar, we implement the latest, best-performing tag-correlating prefetching of [9] to prefetch into L2 without disrupting L1. We discuss the prefetcher implementation details in Section 4.3.1.

3 Methodology

We modified simplescalar-3.0 for our evaluation. Our simulator carefully models SMT pipeline details, including out-of-order issue, memory-bus occupancy, multiple contexts, virtual-to-physical address translation, per-thread load/store queues and active **lists**, and shared physical register file and issue queue. The simulator models a pipeline that supports thread-level squashing on branch misprediction. To improve instruction throughput in SMT, we apply squashing on L2 misses [27], except for special cases that we will mention later. Because the L1 caches are virtually indexed, we use address offsetting described in [15] to spread out accesses of different threads in the cache. We also use the Bin-Hop page allocation policy to spread out accesses in the L2 cache [15].

Our simulator runs *Alpha* binaries that are compiled with *peak* setting. We fast-forward the first two billion instructions of each thread. The fast-forwarding warms up branch predictor, the L2 and L1 caches, but do not gather statistics. We then simulate until one of the threads reaches 100 million instructions. For four or eight threads this method simulates more than 100 million instructions. Therefore, our results are unlikely to be biased by individual programs. Recently [26] proposes clustering phases to reduce simulation time while minimizing errors for simulating single program. However, clustering for a multi-programmed workload is more complicated and involves mixing phases of several programs. Because [26] does not show clustering for SMT simulations, we do not use such approach.

Table 2. Applications and workloads

Category	**Benchmarks**		
1T.ILP	**mesa**, **crafty, fma3d**, **eon,** **facerec**, **equake, sixtrack**, **galgel**		
1T.MEM	*vpr*, *apsi*, *art*, *applu*, *swim*, *lucas*, *mcf*, *ammp*		
1T.MIX	1T.ILP + 1T.MEM		
Workload	**Composition**	**Workload**	**Composition**
2T.ILP.1	**mesa, crafty**	2T.MIX.1	*vpr*, **mesa**
2T.ILP.2	**fma3d, eon**	2T.MIX.2	*apsi*, **crafty**
2T.ILP.3	**facerec, equake**	2T.MIX.3	*art*, **fma3d**
2T.ILP.4	**sixtrack, galgel**	2T.MIX.4	*applu*, **eon**
2T.MEM.1	*vpr, apsi*	2T.MIX.5	*swim*, **facerec**
2T.MEM.2	*art, applu*	2T.MIX.6	*lucas*, **equake**
2T.MEM.3	*swim, lucas*	2T.MIX.7	*mcf*, **sixtrack**
2T.MEM.4	*mcf, ammp*	2T.MIX.8	*ammp*, **galgel**
4T.ILP.1	2T.ILP.{1,2}	4T.MIX.1	2T.MIX.{1,2}
4T.ILP.2	2T.ILP.{3,4}	4T.MIX.2	2T.MIX.{3,4}
4T.MEM.1	2T.MEM.{1,2}	4T.MIX.3	2T.MIX.{5,6}
4T.MEM.2	2T.MEM.{3,4}	4T.MIX.4	2T.MIX.{7,8}
8T.ILP.1	4T.ILP.{1,2}	8T.MIX.1	4T.MIX.{1,2}
8T.MEM.2	4T.MEM.{1,2}	8T.MIX.2	4T.MIX.{3,4}

Multi-programmed workload is one of the most important workloads for SMT. To simulate real-world workloads, we choose sixteen benchmarks from SPEC2000 to compose workloads that have two, four and eight threads. Out of these sixteen benchmarks, eight achieve the highest IPCs (shown in bold) and the other eight have the most L2 cache misses per instruction (shown in italics). We mixed these benchmarks to create three representative workloads of different ILP and memory demand. Table 2 lists the SPEC2000 benchmarks and multi-programmed workloads we use in this study. The first set, called *ILP,* consists of the high-ILP programs; the second set, called *MEM,* consists of the high-miss-rate programs; and the third set, called *MIX,* combines programs from both ILP and MEM. Within a set, there are four groups (*1T, 2T, 4T, and 8T*) and each group indicates the workloads for a given number of threads. We use the ref input for all benchmarks.

Table 3. Base Configuration

Issue Width	8
L1 I-cache	64KB, 2-way, pipelined
L1 D-cache	64KB, 4-way, 3-cycle hit latency
L2 Cache	4MB, 8-way, 15-cycle hit latency
Memory	150 cycle latency, 4-cycle pipelined, split-transaction bus
Branch Predictor	16k/16k/16k spec-update, 8-cycle misprediction penalty
Physical Registers	100+*T**32 INT , 100+*T**32 FP for *T* threads
Active List	128/context
Load-Store Queue	64/context
Issue Queue	32 INT, 32 FP
MSHR	32

Table 3 lists the configuration for the basic SMT in our study. We carefully choose an aggressive SMT core such that our results are representative of many different SMT configurations in the foreseeable future; a less aggressive SMT would handicap the techniques we study because of less headroom for improvements. We use an issue width of eight as other SMT-related previous studies do [29,27,28], unless otherwise specified. For branch prediction, we use a hybrid of local and *gshare* predictors. Each context uses a 128-entry return address stack and maintains its own branch history for the gshare predictor. The SMT in this study has two fetch ports and fills up fetch bandwidth from up to two threads. We use *ICOUNT* as our SMT fetch policy as recommended in [28].

We will describe the implementation details of trace cache, value prediction and prefetching, in Section 4.1.1, Section 4.2.1, and Section 4.3.1, respectively.

4 Results

We present our results for trace cache, value prediction, and prefetching in Section 4.1, Section 4.2, and Section 4.3, respectively. As stated in Section 1, because SMT's goal is to improve throughput, which is also an important performance metric for server-class machines which increasingly use SMT, we evaluate these techniques in terms of instruction throughput.

We find that (1) given similar size for the duo of trace cache and backup i-cache as the conventional i-cache, trace cache degrades SMT throughput compared to the conventional i-cache; (2) given a typical number of physical registers, value prediction degrades throughput; (3) prefetching improves throughput for memory-intensive workloads but degrades throughput for workloads with mixed memory demand.

4.1 Trace Cache

4.1.1 Trace Cache Implementation

We implement the latest, most space-efficient block-based trace cache (TC) described in [3]. The TC is implemented using a block cache and a trace table. Each block of a block cache stores a small subtrace (e.g., a few consecutive basic blocks up to six instructions) and the trace table stores pointers to the block cache. To provide high bandwidth, the block cache is multi-ported (implemented via true ports and/or copies). The trace table provides n pointers which are used to pull out n subtraces from the block cache, and the subtraces together form the fetch unit of one trace. The subtraces are formed by observing past instances of the instruction stream. The trace table is updated with pointers to the subtraces. Because the subtraces are small, there is less repetition than trace cache using full-blown traces [22,21,20,8,19]. Furthermore, only the pointers to subtraces, but not subtraces themselves, are repeated in the full traces, achieving further compaction.

Because our results show that TC is ineffective for SMT, we make the following assumptions to ensure that our results are not due to insufficient resources or inefficient implementation: 1) Our TC uses an ideal, sequential, atomic multiple-branch predictor that accurately updates branch history even for branches predicted in the same cycle. In contrast, the base case SMT's i-cache uses a conventional, speculatively-updated predictor which predicts up to one taken branch or up to two branches per thread. The TC uses infinite branch-prediction bandwidth, therefore the branch promotion optimization in [19] is irrelevant. 2) The TC uses perfect target prediction for direct branches. 3) The TC has zero-cycle fill latency.

We implement the following key optimizations from [3]: 1) For termination, a subtrace ends upon encountering a branch, a jump, a call or a return instruction near the end of the subtrace. 2) Our TC employs partial matching which allows a substring of a trace, instead of restricting to complete traces, to be supplied. 3) We use a two-way associative "rename table" to map PCs to trace pointers. The table determines whether a trace is present in the block cache on every TC access and handles replacement in the block cache. The table's associativity effectively makes each copy of block cache two-way associative. 4) On fetching, the processor sends a request to both TC and i-cache simultaneously. If the request misses in the TC but hits in the i-cache, there is a one-cycle penalty, as in [22,21,20,19,8,3]. 5) To compensate for block-level fragmentation, the TC provides more instructions than the processor's front-end width. The front end picks the number of instructions requested to send into the pipeline and buffers any excess instructions to be combined with the next trace. Our TC has a six-instruction block size, as recommended in [3]. 6) We update the block cache speculatively on misses, as opposed to updating at commit. Other simulations (not shown) reveal that speculative update performs better.

When using the TC in SMT, we do the following to ensure that our SMT adaptation of the scheme is not disadvantaged by easily solvable problems: 1) We

employ address offsetting in the TC and its accompanying i-cache. 2) Each cycle, two threads access the TC and each thread gets half the TC bandwidth. Our simulations (not shown) reveal that this policy achieves better performance than giving the full TC's bandwidth to only one thread.

4.1.2 Trace Cache Results

Recall from Section 1 that TC trades off space for bandwidth and that the sharing of instruction storage among SMT's threads impacts this trade-off. In this section, we evaluate this trade-off in SMT. Because we found that TC benefits little with an 8-issue pipeline even for single-thread workloads (not shown), we use 16-issue width for the TC as in [22,21,20,19,8,3]. Accordingly, we also double the pipeline resources listed in Table 3, such as rename registers, issue queue, active list, load-store queue, and execution units.

Figure 1 shows the throughput improvements of TC over the base case, which has 64KB i-cache and no TC. We show three sets of workloads: ILP, MEM, and MIX, as defined in Section 3. For each set, there are four groups of bars (*1T, 2T, 4T, and 8T*) varying the number of threads as one, two, four, and eight. Each bar indicates the geometric mean of throughput improvements for the workloads in the set.

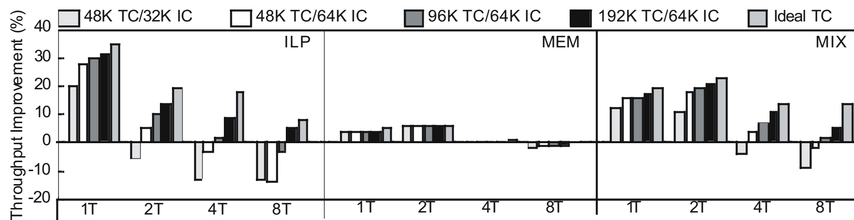

Fig. 1. Trace Cache Throughput Improvements

Because we are interested in TC's space-for-bandwidth trade-off, we vary TC size. Within each group of bars for a given number of threads, the first bar shows a 48K TC using two copies of dual-ported block cache backed up by a dual-ported 32K i-cache, for a total size of 80K, compared to the base case of a dual-ported 64K i-cache. Thus, the first bar represents our comparison using a similar total size. The next three bars from left to right show a 64K i-cache combined with TC of 48K, 96K, and 192K (1-cycle latency). These bars represent the cases where the TC configurations use extra space compared to the base case of a 64K i-cache.

To examine the upper limit of improvement through enhancing TC, by increasing size/associativity or using per-thread TC, we simulate *Ideal TC* which is an oracle configuration that has infinite size and always supplies as many instructions as the fetch bandwidth from two threads every cycle. Ideal TC does not suffer from fetch fragmentation or capacity/conflict miss, and subsumes enhancements. Ideal TC uses the same ideal multiple branch predictor as other TCs mentioned in Section 4.1.1. The last bar shows Ideal TC's throughput improvement.

Figure 1 shows two clear trends. First, TC benefits ILP and MIX but not MEM. While ILP and MIX have enough parallelism that fetch bandwidth is important for performance, MEM is dominated by data cache misses that fetch bandwidth is not important.

Second, for similar-size configurations, TC offers no benefit to SMT, and can lead to throughput degradation as the number of threads increases. This similar-size comparison is important because increasing the size of the level-1 instruction storage is difficult due to latency, area, and power considerations, as mention in Section 1. When we add an extra TC to the base 64K i-cache, TC is effective for single threads. This result agrees with previous papers [22,21,20,19,8,3] ([3] also gives extra space to TC), indicating that our TC implementation is correct. For two threads, TC improves SMT throughput; this results agrees with the two-threaded Pentium IV's use of a trace cache. However, when threads increase to more than two, TC's advantage rapidly diminishes. The *base case throughput*, shown in the first row of Table 4, continues to improve as we increase the number of threads to eight, showing that TC's diminishing returns are *not* due to pipeline saturation. Even with a large, 192K TC with single-cycle latency, TC shows only modest improvement over the base case for four or more threads. These results are no surprise when we look at the last bar, which shows the throughput improvement with an ideal TC. The last bar clearly shows that TC's potential drops rapidly as thread increases. Thus, we see that SMT's sharing of instruction storage makes TC's space-for-bandwidth trade-off unprofitable.

In SMT, applying a technique may impact low-IPC threads and high-IPC threads differently. With the goal of maximizing throughput, SMT distributes resources (fetch and front-end bandwidth, issue queue slots, etc.) among threads proportional to each thread's individual IPC (e.g., using ICOUNT). However, applying a technique may improve a low-IPC thread, fooling SMT into allocating more resources to the improved-but-still-low-IPC thread at the cost of other high-IPC threads, reducing overall throughput. That is, one thread may improve but the overall throughput may reduce. To capture such cases, [27] introduces *weighted speedup*, which is the geometric mean of IPC improvements of each thread. If the weighted speedup is more than throughput improvement, then the technique impacts (positively or negatively) low-IPC threads more than high-IPC threads; if the weighted speedup is less than throughput improvement, then the reverse is true. If the two metrics are similar, then the impact on low- and high-IPC threads are similar. Although our goal is to evaluate processor throughput, we show weighted speedup for 48K TC with 64K i-cache in the second row in Table 4. We see that weighted speedup follows the same trend as throughput, confirming that TC's advantage diminishes as the number of threads increases.

To explain TC's downward trend with an increasing number of threads, we compare base case i-cache miss rates with TC miss rates. The third row in Table 4 shows the

Table 4. Trace cache statistics

	ILP workload				MEM workload				MIX workload			
	1T	2T	3T	4T	1T	2T	3T	4T	1T	2T	3T	4T
Base case IPC	4.3	6.9	8.2	9.2	1.4	2.0	3.1	3.5	2.6	4.8	6.9	7.6
Weighted Speedup (%)	28.5	5.2	-2.3	-12.7	4.3	6.1	4.4	0.0	15.7	13.6	4.2	-1.2
64K IC miss rate (%)	0.1	0.0	0.2	0.2	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2
48K TC miss rate (%)	10.5	20.1	31.2	41.9	0.2	0.8	2.5	6.8	5.2	8.6	17.2	29.0
192K TC miss rate (%)	1.2	4.2	10.8	19.7	0.0	0.1	0.3	0.7	0.6	1.0	4.2	10.0
64K IC avg Insts	5.2	8.5	9.4	9.8	1.6	2.9	4.8	5.3	3.1	6.2	8.8	9.2
48K TC avg insts	8.5	9.2	9.0	8.4	2.0	3.5	5.1	5.4	4.4	7.9	9.3	8.9
Ideal TC avg insts	8.7	10.8	11.3	10.9	2.0	3.5	5.1	5.4	4.4	8.5	10.4	10.6

i-cache *miss rate* in the base case, and the fourth and fifth rows show the *miss rates* for the 48K and 192K TCs, respectively. The significantly-higher TC miss rates show that the efficiency of the TC rapidly decreases as the number of threads increases.

Table 4 also shows the average number of instructions supplied by a 64K i-cache (*64K IC avg insts*), a 48K TC with its accompanying 64K i-cache (*48K TC avg insts*), and an ideal TC (*Ideal TC avg insts*) to the pipeline. On average, TC can supply 8.5 instructions per cycle in a single thread, which is 63% more than an i-cache can supply. When multiple threads are available, SMT uses the second fetch port to fetch from another thread. Therefore, SMT sustains high instruction throughput without the complication of a TC. With eight threads, the base i-cache with two ports achieves 9.8 IPC, which is higher than TC's. The base case's higher bandwidth combined with the large, diverging gap between the base case's and TC's miss rates as the number of threads increases, clearly shows that TC's space-for-bandwidth trade-off is not effective in SMT.

There is an interesting observation in Figure 1: MIX gets more benefit from TC than ILP and MEM as threads increase. As expected, ILP gets the most benefit of TC in single-thread runs. As threads increase, the pressure on the TC greatly increases and the miss rate in the TC increases quickly. When we put ILP and MEM together (MIX), the ILP threads experience less pressure in the TC compared to the ILP threads in the ILP workload because ILP threads in MIX take up the slack of the TC space created by MEM threads in MIX. This argument is supported by TC's miss rate shown in Table 4. For instance, TC's miss rate for 8 threads in MIX is similar or lower than TC's miss rate for 4 threads in ILP.

Some processors use TC to hold pre-decoded instructions (e.g. Pentium IV). If such a cache holds merely decoded individual instructions but not traces spanning multiple branches, we consider such a cache to be an i-cache and not a TC, and our results are not applicable to it.

Our experiments favor TC by giving it unrealistic advantages and an aggressive, 16-issue processor which gives TC much headroom for improvement. Nevertheless, we find that TC degrades SMT throughput. Using miss latencies longer than our numbers to model future technology will shift the performance bottleneck to the back-end and reduce opportunity for the front-end, further discouraging the use of TC. We also show that an ideal TC only marginally improves throughput. Therefore, our results unequivocally prove that TC hurts SMT running more than two threads, and there is no need to vary other parameters.

4.2 Value Prediction

4.2.1 Value Prediction Implementation

We implement the latest, best-performing selective value predictor (VP) described in [5]. The value predictor uses a confidence predictor to select when to predict and a hybrid of stride and context predictors to predict values.

Because our results show that VP is ineffective for SMT, we make the following assumption to ensure that our results are not due to inefficient implementation: we assume that VP's value history is updated correctly by an oracle in the decode stage.

We implement the following key optimizations described in [5]: 1) To minimize mispredictions, we implement a history-based confidence predictor. 2) We employ

warm-up counters so that instructions with insufficient history do not update predictors. 3) To reduce mispredictions and maximize the benefit, we allow the predicted value to be used only for load instructions that incur L1 misses. According to our evaluations, this scheme has better performance than one that predicts all instructions. While we use the predictions only on misses, we predict and update on all loads regardless of a hit or a miss to accelerate the predictor's warm-up. 4) Instructions that directly or indirectly consume predicted values are assigned lower priority and can execute only on otherwise-idle execution units. These instructions resume their normal priority when the prediction outcome is known. When a misprediction is detected, the pipeline squashes the thread's instructions that are subsequent to the producer. To avoid unnecessary squashing, squash does not happen if the mispredicted value has not been consumed.

When using VP in SMT, we do the following to ensure that our SMT adaptation of the scheme is not disadvantaged by easily solvable problems: 1) Because VP benefits mostly from L2 misses, SMT's squashing on L2 misses would nullify much of the benefit of VP. Therefore, we modify the squashing policy on L2 misses in SMT. If an L2-missing load is value-predicted, we do not squash the pipeline. This mechanism allows dependent instructions to consume predicted values and later release issue queue entries. When a thread fills up its active list or load/store queue on an value-predicted L2 miss, we squash the thread's instructions only in the front end and stall fetching from the thread until the miss returns. Otherwise, fetched instructions from the thread would clog the front end preventing other threads from making progress. This squashing of the front end is not extra because the base case already squashes the pipeline on all L2 misses, regardless of whether resources fill up. 2) To reduce aliasing in prediction tables, we add tags to all prediction table entries. 3) To avoid inter-thread interference in the prediction tables, we use per-thread prediction tables.

4.2.2 Value Prediction Results

Recall from Section 1 that value-predicting a long-latency operation causes hold-up of registers until the operation completes and the prediction is confirmed. This hold-up occurs even with correct value prediction. In contrast, SMT simply squashes the thread containing the operation, releasing the resources held by the thread and allowing other threads to progress. Thus, there is a choice of value-predicting and holding up registers, versus squashing and overlapping the latency with other threads. SMT's sharing of registers among its threads impacts this choice. In this section, we evaluate this choice in SMT.

Each thread has two four-way, 8K-entry tables, one each for stride prediction and context prediction. To minimize mispredictions, each of these table also has its own 2KB confidence tables. The total size of the VP tables in an eight-thread SMT is a generous 5MB, ensuring that our results are not limited by small tables.

Figure 2 shows VP's throughput improvements compared to an SMT without VP. Similar to Section 4.1.2, Figure 2 shows three sets of workloads, ILP, MEM, and MIX, and varies the number of threads as one, two, four, and eight. Each bar indicates the geometric mean of throughput improvements for the workloads in the set.

Because we are interested in register pressure in the presence of VP, we show two configurations with different number of physical registers. One configuration, called *VP-finite*, contains ($100 + T * 32$) integer registers and ($100 + T * 32$) floating-point

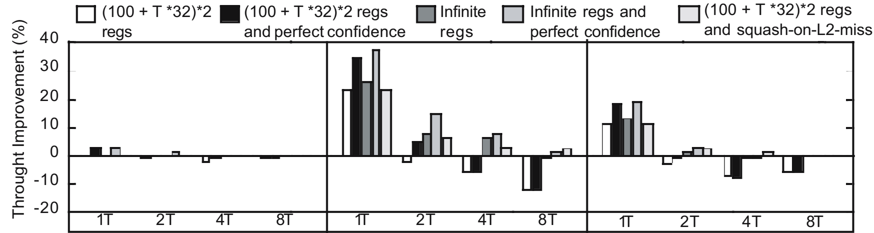

Fig. 2. Value prediction throughput improvements

registers, where T is the number of threads. The number 32 in this expression is the minimum number required for per-thread architectural registers, and 100 is the number of registers for renaming. This configuration represents a realistic number of registers, and it has also been used in [28]. The other configuration, called *VP-infinite*, has infinite physical registers. To examine if VP can benefit SMT by overlapping only L1 misses, we show a configuration called *VP-squash*. *VP-squash* uses VP if a load misses in L1 but squashes (mentioned in Section 3) if the load also misses in L2. *VP-squash* has the same number of physical registers as *VP-finite*.

Figure 2 shows a group of five bars for a given number of threads. The first bar shows *VP-finite*. The second bar shows *VP-finite* with perfect confidence prediction. The third and fourth bars show *VP-infinite* without and with perfect confidence prediction, respectively, quantifying VP's potential if register pressure were absent. The base case for the first and second bars is an SMT with as many physical registers as *VP-finite*. The base case for the third and fourth bars is an SMT with an infinite number of physical registers. The last bar shows *VP-squash* normalized to the base case for *VP-finite*.

Figure 2 shows that VP benefits MEM and MIX but not ILP. This result is hardly surprising because VP is triggered only for L1 misses, and ILP has low L1 miss rates. On the other hand, VP hides the penalties of the misses present in MEM and MIX.

Looking at MEM and MIX, this figure shows a interesting trend: VP significantly improves single-thread performance, especially for MEM. This result agrees with the results from previous VP papers [16,1,23,17,5], indicating that our value predictor is implemented correctly. However, *VP-finite*'s throughput improvements decrease significantly and become negative as the number of threads increases in both MEM and MIX. Note that the *base case throughput*, shown in the first row of Table 6, continues to improve as we increase the number of threads to eight, showing that VP's diminishing returns are *not* due to pipeline saturation. *VP-finite* with perfect confidence (second bar) shows the same trend, showing that the degradation exists even when VP is 100% accurate (albeit at non-perfect coverage). Thus, the second bars rule out mispredictions as the cause of the degradation trend.

Two reasons contribute to VP's degradation with multiple threads, even with large VP tables. First, VP's holding up of registers degrades throughput with two or more threads. Figure 2 support this observation by showing that *VP-finite*'s degradation largely disappears when infinite registers are available, as shown by *VP-infinite* (third and fourth bars). Second, SMT's latency tolerance reduces VP's opportunity. Figure 2 supports this argument by showing that even using *VP-infinite* with perfect confidence, VP's opportunity diminishes and eventually disappears, as the number of

threads increases. Near-zero opportunity combined with register pressure forces *VP-finite* to incur degradation at more than two threads. The near-zero opportunity also shows that VP's benefit would be marginal, even if the VP implements selective recovery (mentioned in Section 2.2) to reduce misprediction penalty.

The four-thread *VP-infinite* bars for MIX show a small degradation despite having infinite registers. Because MIX has high-ILP and low-ILP threads, VP's impact is not uniform (this point is also made in Section 4.1.2). VP helps low-ILP threads more than high-ILP threads to the point that SMT allocates resources to the improved-but-still-low-ILP threads at the cost of the high-ILP threads. Such allocation causes a slight overall throughput degradation. *VP-squash* improves two-thread MEM by 6%, but improves little for other workloads (0-3%). This result shows that implementing VP to overlap only L1 misses is not profitable for SMT.

Table 5 shows important statistics for VP. *Coverage* is the ratio of the number of predictions over the number of L1 load misses. WP in the table means Weighted Speedup. *Squash Rate* is the ratio of the number of squashes caused by value mispredictions over the number of predictions. We see that coverage and squash rate are fairly stable across threads, and the squash rate is low. The stability of these metrics clearly indicates that VP's degradation with two or more threads is not due to worse coverage or more value mispredictions.
The fourth, fifth and sixth rows show the *weighted speedup* (explained in Section 4.1.2) for *VP-finite*, *VP-infinite and VP-squash*. *VP-finite*'s weighted speed-ups are positive for two or more MEM and MIX threads while the overall throughput degrades. Because there is a large variance in the individual predictability and IPCs of these MEM and MIX threads, VP's impact is uneven among the threads, causing weighted speedup to deviate from overall throughput (as explained in Section 4.1.2). VP fools SMT into allocating more registers to the improved-but-still-low-ILP threads which hold up the registers from the high-ILP threads, degrading overall throughput. Because SMT's goal is to improve processor throughput, techniques which improve individual threads while degrading processor throughput defeat SMT's purpose. In fact, SMT does the reverse: SMT employs several optimizations which improve processor throughput at the cost of individual threads. For example, (1) because the SMT pipeline is typically deeper than a superscalar pipeline [29], single-thread performance slightly worsens on SMT. (2) SMT's ICOUNT, which optimizes processor throughput, may worsen a low-IPC thread by fetching more often from higher-IPC threads [28]. (3) Squashing a thread on L2 misses improves processor throughput while slightly worsening the thread's IPC [27].

Table 5. Value prediction statistics

	ILP workload				MEM workload				MIX workload			
	1T	2T	3T	4T	1T	2T	3T	4T	1T	2T	3T	4T
Base case IPC	3.5	4.6	5.1	5.3	1.1	1.8	2.9	3.9	2.1	3.8	4.7	5.2
Coverage (%)	15.0	22.6	27.9	42.2	34.4	28.4	24.0	31.0	24.3	30.6	28.7	31.3
Squash rate (%)	0.3	0.2	0.2	0.2	0.6	0.5	0.4	0.4	0.4	0.3	0.2	0.2
VP-finite WP (%)	0.5	-1.5	-1.6	-1.9	23.0	8.8	8.5	7.5	11.2	6.0	0.7	1.6
VP-infinite WP (%)	0.5	-0.4	-0.2	-0.4	26.0	18.2	19.3	14.7	12.5	10.6	7.1	6.9
VP-squash WP (%)	0.5	-0.4	-0.3	-0.4	23.0	12.2	4.4	2.9	11.3	4.5	2.3	0.9

Our experiments favor VP by giving it unrealistic advantages and an aggressive processor which gives VP much headroom for improvement. Still, VP degrades SMT throughput. Because VP holds up physical registers during an L2 miss, using longer miss latencies to model future technology will further degrade throughput. We also show that VP does not improve throughput even with infinite registers. Therefore, our results unequivocally prove that VP hurts SMT and there is no need to vary other parameters.

4.3 Prefetching

4.3.1 Prefetching Implementation

We implement the latest, best-performing tag-correlating prefetching (PF) [9]. The prefetcher decides what to prefetch by using the L1 miss stream as history to predict the next miss. The predictor is a two-level scheme, where the first level stores per-set miss stream history, and the second level stores the tag of next-misses. Upon an L1 miss, the prefetcher triggers prefetch to the predicted next miss. Instead of using dead-time prediction to trigger prefetches, this simplified prefetcher uses L1 misses as the triggers.

Because our results show that PF is effective for SMT, we do not give any undue advantage to the prefetcher to ensure that our results are not due to unjustifiable implementation assumptions. Specifically, because PF uses additional space for prediction tables, we compensate the base case by running it with a larger L2. Because the largest predictor size we use is 498KB (in an eight-thread SMT), we use a 4.5 MB L2 for all base case runs, while using only a 4MB L2 for all the PF runs. We enlarge the base case's L2 with no penalty to the L2 hit latency.

We implement the following key optimizations in the two-level predictor: 1) The predictor uses eleven bits of the L1 tag and one bit of the L1 index from the previous three misses, together with the full L1 tag from the previous miss, to form indexes into the second-level table as in [9]. 2) The first level uses per-set history as recommended in [32]. 3) The prefetcher uses 32 extra MSHRs to hold in-flight prefetch status and a 128-entry prefetch queue to hold pending prefetch requests, as recommended in [32].

When using PF in SMT, we do the following to ensure that our SMT adaptation of the scheme is not disadvantaged by easily solvable problems: 1) While [32] prefetches data into L1, [9] argues that prefetching into L1 is difficult due to L1 contention. [9] shows that prefetching data only into L2 achieves most of the benefit of prefetching into L1 while entirely eliminating dead-block prediction. This effect is seen because L1 miss latency can be overlapped easily with ILP. While prefetching into L2 in SMT introduces the issue queue clogging described in Section 1, we could reduce prefetch coverage to balance prefetching and issue queue clogging and improve overall throughput. Therefore, we evaluate prefetching into L2. 2) The second-level table is accessed with the previous L1 miss and history, that is also made of L1 misses. Because the L1 is physically tagged, the L1 miss stream has physical addresses which are already randomized by bin-hopping (Section 3). Consequently, the second-level table does not need any offsetting to reduce conflicts. 3) Each level of prediction tables may be configured to be shared across threads or to be private to each thread. Because there is not much difference between shared or private for the second level, we use a shared second-level table. We show both private and shared configurations for the first-level table. 4) We increase the second-level shared table

size with the number of threads (the table size is T * 120KB for T threads, except for eight-thread SMT we use T=4 to keep the size under 512KB). We increase the size with no change in the associativity, keeping the implementation reasonable. Although the table is shared among threads, no major conflicts among the threads occur because the table is accessed using physical addresses, which are unique across the threads.

4.3.2 Prefetching Results

Recall from Section 1 that because SMT tolerates latency but at the same time increases pressure on the memory hierarchy by overlapping multiple threads, the opportunity for PF is unknown. While PF reduces memory latency, prefetching into L2 encourages L1 misses in fewer cycles, which causes clogging of the issue queue and slows down the other threads. Ironically, only correct prefetches cause this clogging. Thus, SMT's sharing of the issue queue across multiple threads impacts PF's effectiveness. In this section, we evaluate these opposing effects of PF in SMT.

While we showed that TC and VP do not improve SMT throughput; in this section, we will show that PF improves throughput for MEM, but has limited opportunity for MIX.

Figure 3 shows PF's throughput improvements compared to the SMT without PF. As before, Figure 3 shows the three sets of workloads, ILP, MEM, and MIX, and varies the number of threads as one, two, four, and eight. Each bar indicates the geometric mean of throughput improvements for the workloads in the set. Note that the Y-axis scale has changed from the previous graphs.

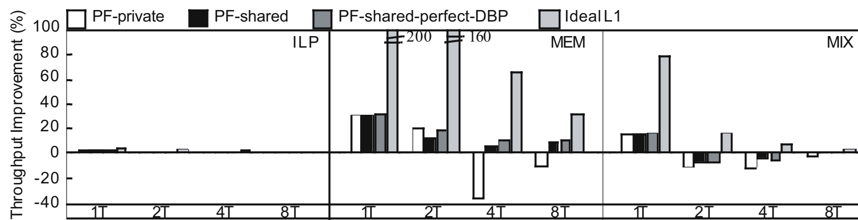

Fig. 3. Prefetching throughput improvements

Because prefetching into L2 causes the issue queue clogging problem, we studied ways to reduce this clogging. First, we experimented with prefetching into L1. Unlike prefetching into L2, prefetching into L1 cannot use L1 misses as triggers (prefetched block will displace useful data) and needs dead-block prediction [32]. We found that because of high pressure on L1 in SMT, the dead time is shorter in SMT than that in a single thread. The shorter dead time makes dead block prediction harder. Second, we resumed prefetching into L2 and tried to avoid clogging by preventing instructions which are past an L1 miss that hits in a prefetched L2 block from entering the pipeline. To this end, we used an L1 miss predictor which stops fetching past a predicted L1 miss. The predictor essentially needs to balance accuracy (avoid incorrectly stopping fetch due to mispredictions) and coverage (identify all the misses). Unfortunately, achieving this balance proved to be difficult. Therefore, we looked into other ways to reduce the clogging. Taking a hint from VP, which reduces coverage to reduce

mispredictions, we tried to reduce prefetch coverage to prevent too many L2 misses from being converted to L1 misses, which cause clogging. Reasoning that a shared first-level history would have less coverage than a private first-level history due to inter-thread interference, we experimented with these two configurations. We found that the shared first-level history works better and achieves throughput improvements. Because PF already shows throughput improvement for MEM and little opportunity for MIX, further enhancements through either prefetching into L1 or using a better L1 miss predictor will improve MEM's throughput more and will not improve MIX. Any such further enhancement will only reinforce our conclusions.

Figure 3 shows a group of four bars for a given number of threads. The first bar shows PF-private, which uses a private first-level history. The second bar shows PF-shared, which uses a shared first-level history. To confirm that the absence of dead-block prediction does not affect PF (as previously shown in [9]), the third bar, PF-shared-prefect-DBP, shows PF with perfect L2-dead-block prediction. To show the potential of PF, the last bar shows Ideal L1, which lets every access from the processor hit in L1.

From Figure 3, we see that PF does not benefit ILP much with multiple threads, because ILP has low L1 miss rates. Looking at MEM, PF significantly improves single-thread performance. This result agrees with the results from previous PF papers [32,9], indicating that our prefetcher is implemented correctly. With multiple threads, we see that while PF-private degrades throughput, PF-shared improves throughput. PF-private's poor performance is due to issue queue clogging, as can be seen in the second and third rows in Table 6 by the larger fraction of time the Issue Queue (*IQ*) stays clogged (*IQ clog frac.*) with PF-private (*PF-p*) than with PF-shared (*PF-s*). Note that the base case throughput, shown in the first row of Table 6, continues to improve as we increase the number of threads to eight, showing that PF-private's diminishing returns are not due to pipeline saturation. Thus we see that SMT's sharing of the issue queue among its threads accounts for the difference between PF-private's failure and PF-shared's success.

We also see that PF-shared-perfect-DBP is marginally better than PF-shared, showing that using L1 misses as triggers is a good dead-block predictor, as also claimed by [9]. Ideal L1 shows that though the opportunity reduces with more threads, there is still substantial opportunity even with eight threads. PF-shared captures some

Table 6. Prefetching statistics

	ILP workload				MEM workload				MIX workload			
	1T	2T	3T	4T	1T	2T	3T	4T	1T	2T	3T	4T
Base case IPC	3.5	4.6	5.1	5.3	1.2	1.8	3.1	4.0	2.1	3.8	4.7	5.2
PF-p IQ clog frac. (%)	2.5	1.0	0.3	0.3	37.7	25.2	36.8	18.2	18.8	10.3	7.2	1.7
PF-s IQ clog frac. (%)	2.5	1.0	0.5	0.4	37.7	24.8	11.5	6.1	18.8	9.6	3.9	1.1
Base case L2 miss (%)	9.8	7.5	9.1	4.8	26.8	25.3	26.7	32.6	18.0	24.8	21.9	21.9
PF-p L2 miss (%)	7.9	5.0	4.9	3.4	12.2	9.1	12.0	15.8	10.0	11.1	9.5	10.8
PF-s L2 miss (%)	7.9	5.0	7.5	4.6	12.2	16.9	19.9	26.5	10.0	12.4	17.1	18.6
PF-p accuracy (%)	22.7	24.6	46.6	68.2	66.1	85.1	83.7	86.9	42.7	63.2	81.3	80.5
PF-s accuracy (%)	22.7	24.0	34.7	30.5	66.1	73.6	67.6	62.2	42.7	59.3	66.5	55.9
PF-p WP (%)	2.0	1.2	0.5	0.3	30.7	53.0	22.3	20.9	15.5	19.7	26.0	12.4
PF-s WP (%)	2.0	1.0	0.2	0.0	30.7	21.7	14.7	10.7	15.5	16.7	7.2	2.4

of this opportunity, achieving 7% improvement with four threads and 9% with eight threads. These results show that when memory latency is a major bottleneck, even multiple threads cannot tolerate all L2 misses, and PF is effective.

For MIX, PF-shared suffers 6% and 4% degradation with two and four threads, respectively. Despite using a shared configuration, this workload causes issue queue clogging, resulting in slight throughput degradation. This result is not surprising when we look at Ideal L1, which shows little opportunity for PF with increasing threads. This limited opportunity combined with issue queue clogging forces PF-shared to degrade with two or more threads.

Table 6 presents important statistics for PF. The fourth, fifth, and sixth rows show the L2 miss rates in the base case and PF-private and PF-shared, respectively. These miss rates confirm that PF is effective in reducing L2 misses. PF-private's miss rates are lower than those of PF-shared, indicating that PF-private has higher coverage than PF-shared. This higher coverage causes clogging problems that result in throughput degradation. The next two rows show that the accuracy of PF-private and PF-shared behave similarly to coverage and have the same effect. Finally, we show weighted speedup for PF-private and PF-shared. PF-private has positive weighted speedups for MEM and MIX while it degrades throughput, showing that PF-private improves low-IPC threads with high miss rates at the cost of overall throughput. PF-shared has positive weighted speedups but lower than those of PF-private due to lower coverage. For MEM, PF-shared has both positive weighted speedups and improved throughput, indicating that PF-shared improves low-IPC threads without hurting the other threads.

Our experiments do not give PF any undue advantage and yet show that PF improves SMT throughput for MEM. Because PF hides L2-miss latencies, using longer latencies will further improve throughput. For MIX workload, we showed that PF does not improve even with an ideal L1. Therefore, our results unequivocally prove that PF improves MEM and does not improve MIX, and there is no need to vary other parameters.

5 Conclusions

In this paper, we evaluated trace cache, value prediction and prefetching in SMT. We found that SMT's sharing of the instruction storage (i.e., trace cache or i-cache), physical registers, and issue queue impacts the effectiveness of trace cache, value prediction, and prefetching, respectively.

We found that: (1) Trace cache introduces multiple copies of the same instructions in different traces, trading off space for bandwidth. However, SMT needs a large instruction storage because multiple threads share the storage. Furthermore, trace cache's benefit of supplying many instructions in one fetch diminishes in SMT because SMT can do so by fetching from multiple threads. Our simulations showed that when compared to a similar-sized i-cache, trace cache's space-for-bandwidth trade-off degrades SMT throughput (for 2 threads, throughput improves, supporting Intel's decision to use a trace cache in the two-threaded P4). (2) Value prediction causes hold-up of physical registers and cannot release them until after the predicted instruction completes and commits. Because SMT's multiple threads share physical registers, this hold-up stalls progress in other threads. Thus, unlike superscalar, SMT incurs throughput degradation even with correct value predictions. Our simulations

showed that with a typical number of physical registers, value prediction degrades SMT throughput; and with unlimited registers, value prediction's benefit disappears with an increasing number of threads. (3) Prefetching into L2 converts slow L2 misses into fast L2 hits. However, the L2 hits still miss in L1, resulting in the same L1 misses occurring in fewer cycles. Because instructions dependent on the L1 misses clog the issue queue and because SMT's multiple threads share the issue queue, this clogging stalls progress in other threads. With prefetching, L1 misses occur in fewer cycles, clogging the issue queue more often. Thus, unlike superscalar, SMT incurs throughput degradation even with correct prefetches. Therefore, SMT needs to balance prefetching and issue queue clogging. Our simulations showed that prefetch coverage can be reduced to achieve such balance, improving throughput for memory-intensive workloads. However, for workloads with mixed memory demand (high-ILP and memory-intensive threads), prefetching has little opportunity and slightly degrades throughput.

On one hand, the techniques are ineffective for multi-programmed workloads and in many cases hurt throughput; on the other hand, the techniques significantly improve single-thread performance, and disabling them to improve multi-programmed throughput would hurt single-thread performance. In an SMT with thread priority, these techniques may also hurt high-priority threads in a multi-programmed workload. Thus, our findings create a new responsibility for the OS: We recommend that the OS disable the techniques when running multi-programmed workload, and enable them for single-threaded workload and for high-priority threads in a multi-programmed workload.

References

1. A.Mendelson and F.Gabbay. Speculative execution based on value prediction. Technical report, Technion, 1997.
2. R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In *Proc. of the 34th MICRO*, Nov. 2001.
3. B. Black, B. Rychlik, and J. P. Shen. The block-based trace cache. In *Proc. of the 26th ISCA*, Oct. 1999.
4. E. Borch, E. Tune, S. Manne, and J. Emer. Loose loops sink chips. In *Proc. of 8th HPCA*, Feb. 2002.
5. B. Calder, G. Reinman, and D. M. Tullsen. Selective value prediction. In *Proc. of the 26th ISCA*, May 1999.
6. M. J. Charney and A. P. Reeves. Generalized correlation-based hardware prefetching. Technical Report EE-CEG-95-1, Cornell University, Feb. 1995.
7. K. I. Farkas and N. P. Jouppi. Complexity/performance tradeoffs with non-blocking loads. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 211–222, Apr. 1994.
8. D. H. Friendly, S. J. Patel, and Y. N. Patt. Alternative fetch and issue policies for the trace cache fetch mechanism. In *Proc. of the 30th MICRO*, Nov. 1997.
9. Z. Hu, M. Martonosi, and S. Kaxiras. Tcp: Tag correlating prefetchers. In *Proc. of 9th HPCA*, Feb. 2003.
10. D. Joseph and D. Grunwald. Prefetching using markov predictors. In *Proc. of the 24th ISCA*, June 1997.
11. N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proc. of the 17th ISCA*, May 1990.

12. S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: Exploiting generational behaviour to reduce cache leakage power. In *Proc. of the 28th ISCA*, June 2001.
13. A.-C. Lai, C. Fide, and B. Falsafi. Dead-block prediction and dead-block correlating prefetchers. In *Proc. of the 28th ISCA*, June 2001.
14. M. H. Lipasti, W. J. Schmidt, S. R. Kunkel, and R. R. Roediger. Spaid:software prefetching in pointer and call intensive environments. In *Proc. of the 28th MICRO*, Nov. 1995.
15. J. Lo, L. Barroso, S. Eggers, K. Gharachorloo, H. Levy, and S. Parekh. An analysis of database workload performance on simultaneous multithreaded processors. In *Proc. of the 25th ISCA*, June 1998.
16. M.H.Lipasti, C.B.Wilkerson, and J.P.Shen. Value locality and data speculation. In *Proc. of the 7th ASPLOS*, Oct. 1996.
17. A. Moshovos and G. S. Sohi. Streamlining inter-operation memory communication via data dependence prediction. In *Proc. of the 30th MICRO*, Dec. 1997.
18. I. Park, M. D. Powell, and T. N. Vijaykumar. Reducing register ports for higher speed and lower energy. In *Proc. of the 35th MICRO*, Nov. 2002.
19. S. J. Patel, M. Evers, and Y. N. Patt. Improving trace cache effectiveness with branch promotion and trace packing. In *Proc. of the 25th ISCA*, June 1998.
20. S. J. Patel, D. H. Friendly, and Y. N. Patt. Evaluation of design options for the trace cache fetch mechanism. In *IEEE Transactions on Computers, Special Issue on Cache Memory and Related Problems*.
21. S. J. Patel, D. H. Friendly, and Y. N. Patt. Critical issues regarding the trace cache fetch mechanism. Technical Report CSE-TR-335-97, University of Michigan, May 1997.
22. E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *Proc. of the 29th MICRO*, Dec. 1996.
23. Y. Sazeides and J. E. Smith. Implementations of context based value predictors. Technical Report ECE-97-8, University of Wisconsin-Madison, Dec. 1997.
24. T.C.Mowry, M.S.Lam, and A.Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proc. of the 5th ASPLOS*, Oct. 1992.
25. T.F.Chen and J.L.Baer. Reducing memory latency via non-blocking and prefetching caches. In *Proc. of the 5th ASPLOS*, Oct. 1992.
26. G. H. Timothy Sherwood, Erez Perelman and B. Calder. Automatically characterizing large scale program behavior. In *Proc. of the 10th ASPLOS*, Oct. 2002.
27. D. M. Tullsen and J. A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *Proc. of the 34th MICRO*, Dec. 2001.
28. D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proc. of the 23rd ISCA*, May 1996.
29. D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *Proc. of the 22nd ISCA*, June 1995.
30. G. S. Tyson and T. M. Austin. Improving the accuracy and performance of memory communication through renaming. In *Proc. of the 30th MICRO*, Dec. 1997.
31. T.-Y. Yeh, D. Marr, and Y. Patt. Increasing instruction fetch rate via multiple branch prediction and a branch address cache,. In *In Proc. of the 7th ACM Int. Conf. on Supercomputing*, July 1993.
32. S. K. Zhigang Hu and M. Martonosi. Timekeeping in the memory system: Predicting and optimizing memory behavior. In *Proc. of the 29th ISCA*, May 2002.