Memory-Level Parallelism Aware Fetch Policies for Simultaneous Multithreading Processors

STIJN EYERMAN and LIEVEN EECKHOUT
Ghent University

3

A thread executing on a simultaneous multithreading (SMT) processor that experiences a long-latency load will eventually stall while holding execution resources. Existing long-latency load aware SMT fetch policies limit the amount of resources allocated by a stalled thread by identifying long-latency loads and preventing the thread from fetching more instructions—and in some implementations, instructions beyond the long-latency load are flushed to release allocated resources.

This article proposes an SMT fetch policy that takes into account the available memory-level parallelism (MLP) in a thread. The key idea proposed in this article is that in case of an isolated long-latency load (i.e., there is no MLP), the thread should be prevented from allocating additional resources. However, in case multiple independent long-latency loads overlap (i.e., there is MLP), the thread should allocate as many resources as needed in order to fully expose the available MLP. MLP-aware fetch policies achieve better performance for MLP-intensive threads on SMT processors, leading to higher overall system throughput and shorter average turnaround time than previously proposed fetch policies.

This article extends "A Memory-Level Parallelism Aware Fetch Policy for SMT Processors" published in the *Proceedings of the 13th International Symposium on High-Performance Computer Architecture (HPCA)* in February 2007. It extends our prior work: by providing an expanded presentation and discussion; by providing performance numbers for a more realistic baseline processor configuration, which includes a hardware prefetcher; in addition, the experimental setup is more rigorous by considering SimPoint simulation points and more aggressively optimized binaries; by studying the impact of the SMT processor microarchitecture configuration on the performance of the MLP-aware fetch policy, thus providing more insight into the performance trends to be expected across microarchitectures; by exploring and evaluating a number of alternative MLP-aware fetch policies; by evaluating the proposed fetch policies using system-level metrics, namely system throughput (STP) and average normalized turnaround time (ANTT); by comparing the proposed MLP-aware fetch policy against static and dynamic resource partitioning.
S. Eyerman and L. Eeckhout are Postdoctoral Fellows with the Fund for Scientific Research–Flanders (Belgium) (FWO–Vlaanderen).
Authors' address: Stijn Eyerman and Lieven Eeckhout, ELIS Department, Ghent University, Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium; email: {seyerman,leeckhou}@elis.UGent.be.

DOI 10.1145/1509864.1509867 http://doi.acm.org/10.1145/1509864.1509867

Categories and Subject Descriptors: C.4 [**Performance of systems**]: Design Studies

General Terms: Design, Performance, Experimentation

Additional Key Words and Phrases: Simultaneous Multithreading (SMT), Fetch Policy, Memory-Level Parallelism (MLP)

ACM Reference Format:

Eyerman, S. and Eeckhout, L. 2009. Memory-level parallelism aware fetch policies for simultaneous multithreading processors. ACM Trans. Architec. Code Optim. 6, 1, Article 3 (March 2009), 33 pages. DOI = 10.1145/1509864.1509867 http://doi.acm.org/ 10.1145/1509864.1509867

1. INTRODUCTION

A thread experiencing a long-latency load (a last cache level miss or D-TLB miss) in a simultaneous multithreading processor [Tullsen et al. 1995; Tullsen et al. 1996; Tuck and Tullsen 2003] will stall while holding execution resources without making progress. This affects the performance of the coscheduled thread(s) because the coscheduled thread(s) cannot make use of the resources allocated by the stalled thread.

Tullsen and Brown [2001] and Cazorla et al. [2004a] recognized this problem and proposed to limit the resources allocated by threads that are stalled due to long-latency loads. In fact, they detect or predict long-latency loads, and as soon as a long-latency load is detected or predicted, the fetching of the given thread is stalled. In some of the implementations studied by Tullsen and Brown [2001] and Cazorla et al. [2004a], instructions may even be flushed in order to free execution resources allocated by the stalled thread, such as reorder buffer (ROB) space, instruction queue entries, and so on, in favor of the nonstalled thread(s). A limitation of these long-latency aware fetch policies is that they do not preserve the memory-level parallelism (MLP) (long-latency loads overlapping in time) being exposed by the stalled long-latency thread. As a result, independent long-latency loads no longer overlap but are serialized by the fetch policy.

This article proposes a fetch policy for SMT processors that takes into account MLP for determining when to fetch stall or flush a thread executing a long-latency load. More in particular, we predict the MLP distance per load miss, or the number of instructions in the dynamic instruction stream over which we expect to observe MLP, and based on the predicted MLP distance, we decide to (i) fetch stall or flush the thread in case there is no MLP, or (ii) continue allocating resources for the long-latency thread for as many instructions as predicted by the MLP predictor. The key idea is to fetch stall or flush a thread only in case there is no MLP; in case there is MLP, we only allocate as many resources as required to expose the available MLP. The end result is that in the no-MLP case, the other thread(s) can allocate all the available resources improving its (their) performance. In the MLP case, our MLP-driven fetch policy does not penalize the MLP-sensitive thread, as done by the previously proposed long-latency aware fetch policies [Tullsen and Brown 2001; Cazorla et al. 2004a]. Our experimental results using SPEC CPU2000 show that the MLP-aware fetch policy achieves a 5.1% higher system throughput (STP) and a 18.8% shorter average-normalized turnaround time (ANTT) for

MLP-intensive workloads compared to previously proposed load-latency aware fetch policies [Tullsen and Brown 2001; Cazorla et al. 2004a]; and a 20.2% and 21% better STP and ANTT, respectively, compared to ICOUNT [Tullsen et al. 1996]. For mixed ILP/MLP-intensive workloads, our MLP-aware fetch policy achieves a 22.4% and 4% better STP compared to ICOUNT and load-latency aware fetch policies, respectively; and a 19.2% and 13.9% better ANTT, respectively.

Dynamic resource partitioning mechanisms, such as DCRA proposed by Cazorla et al. [2004b] and learning-based resource partitioning proposed by Choi and Yeung [2006], also aim at exploiting MLP by giving more resources to memory-intensive threads. DCRA gives a fixed amount of additional resources to memory-intensive threads regardless of the amount of MLP; the MLP-aware fetch policies proposed in this article, on the other hand, drive resource allocation using precise MLP information, and our evaluation shows that the proposed MLP-aware fetch policy outperforms DCRA for memory-intensive workloads. Learning-based resource partitioning learns the amount of resources to give to each thread through performance feedback; MLP-aware fetch policies are more responsive to dynamic workload behavior than learning-based resource partitioning.

This article is organized as follows. We first revisit MLP and quantify the amount of MLP available in our benchmarks (Section 2). We then discuss the impact of MLP on SMT performance (Section 3). These two sections motivate for the MLP-aware fetch policy that we propose in detail in Section 4. After having detailed our experimental setup in Section 5, we then evaluate our MLP-aware fetch policy in Section 6. Before concluding in Section 8, we also discuss related work in Section 7.

2. MEMORY-LEVEL PARALLELISM

We refer to a memory access as being long-latency in case the out-of-order processors cannot hide (most of) its penalty. In contemporary processors, this is typically the case for accessing off-chip memory hierarchy structures, such as large off-chip caches or main memory. The penalty for a long-latency load is typically quite large—on the order of 100 or more processor cycles. (Note that we use the term long-latency load to collectively refer to long-latency data cache misses and data TLB misses.) Because of the long latency, in an out-of-order superscalar processor, the ROB typically fills up on a long-latency load because the load blocks the ROB head, then dispatch stops, and eventually issue and commit cease [Karkhanis and Smith 2002]. When the miss data returns from memory, instruction issuing resumes.

Multiple long-latency loads can be outstanding simultaneously in contemporary superscalar out-of-order processors. This is made possible through various microarchitecture techniques such as nonblocking caches, miss status handling registers, and so on. In fact, in an out-of-order processor, long-latency loads that are relatively close to each other in the dynamic instruction stream, overlap with each other at execution time [Karkhanis and Smith 2002, 2004]. The reason is that as the first long-latency load blocks the ROB head, the ROB will

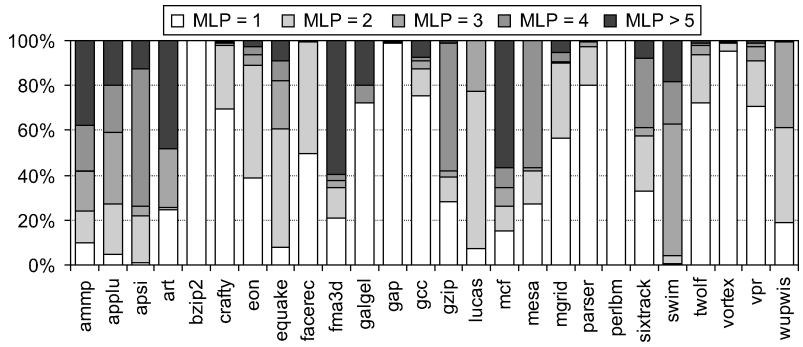

Fig. 1. Amount of MLP for all of the benchmarks assuming a 256-entry ROB superscalar processor.

eventually fill up. As such, a long-latency load that makes it into the ROB will overlap with the independent long-latency loads residing in the ROB, as long as there are enough miss status handling registers and associated structures available. In other words, in case multiple independent long-latency loads occur within W instructions from each other in the dynamic instruction stream, with W being the size of the ROB, their penalties will overlap [Karkhanis and Smith 2002; Chou et al. 2004]. This is called *memory-level parallelism (MLP)* [Glew 1998]; the latency of a long-latency load is hidden by the latency of another long-latency load.

We use the MLP definition by Chou et al. [2004], which is the average number of long-latency loads outstanding when there is at least one long-latency load outstanding. Figure 1 shows the amount of MLP in all of the SPEC CPU2000 benchmarks assuming a 256-entry ROB superscalar processor. Table I (fourth column) shows the average MLP per benchmark. (We refer to Section 5 for a detailed description on the experimental setup.) In these MLP characterization experiments, we consider a long-latency load to be an L3 data cache load miss or a D-TLB load miss. We observe that the amount of MLP varies across the benchmarks. Some benchmarks exhibit almost no MLP—a benchmark having an MLP close to 1 means there is limited MLP. Example benchmarks are bzip2, gap, and perlbmk. Other benchmarks exhibit a fair amount of MLP (e.g., applu, apsi, art, and fma3d).

The fifth column in Table I shows the impact MLP has on overall performance. This number was obtained from an experiment in which we compare the performance difference between a serialized execution of all independent long-latency loads versus a parallel execution of all independent long-latency loads on a 256-entry ROB processor. And thus, it quantifies the performance impact due to MLP (i.e., an MLP impact of 50% means that MLP speeds up the execution by a factor of 2). For several benchmarks, MLP has a substantial impact on overall performance, up to 77.9% for fma3d. Based on this observation, we can classify the various benchmarks according to their MLP-intensiveness (see the rightmost column in Table I). We classify a benchmark as an MLP-intensive benchmark in case the impact of the MLP on overall performance

Table I. The SPEC CPU2000 Benchmarks, Their Reference Inputs, the Number Long-Latency Loads Per 1K Instructions (LLL), the Amount of MLP, the Impact of MLP on Overall Performance, and the Type of the Benchmarks; These Numbers Assume a 256-entry ROB Superscalar Processor with a 4MB L3 Cache

Benchmark	Input	LLL	MLP	MLP Impact	Type
bzip2	program	0.14	1.00	0.03%	ILP
crafty	ref	0.08	1.34	1.29%	ILP
eon	rushmeier	0.00	1.83	0.08%	ILP
gap	ref	0.36	1.02	0.28%	ILP
gcc	166	0.01	1.70	0.22%	ILP
gzip	graphic	0.08	1.81	3.22%	ILP
mcf	ref	17.36	5.17	60.39%	MLP
parser	ref	0.14	1.24	1.20%	ILP
perlbmk	makerand	0.30	1.00	0.01%	ILP
twolf	ref	0.10	1.37	1.05%	ILP
vortex	ref2	0.39	1.06	1.49%	ILP
vpr	route	0.09	1.43	1.35%	ILP
ammp	ref	1.71	3.94	40.25%	MLP
applu	ref	14.24	4.26	69.63%	MLP
apsi	ref	0.78	6.15	35.41%	MLP
art	ref-110	0.19	8.58	7.34%	ILP
equake	ref	24.60	2.69	58.19%	MLP
facerec	ref	0.41	1.51	7.56%	ILP
fma3d	ref	17.67	6.27	77.87%	MLP
galgel	ref	0.24	3.84	14.24%	MLP
lucas	ref	10.63	2.15	46.40%	MLP
mesa	ref	0.45	2.88	19.64%	MLP
mgrid	ref	6.04	1.76	35.84%	MLP
sixtrack	ref	0.10	2.61	4.92%	ILP
swim	ref	15.08	3.66	67.47%	MLP
wupwise	ref	2.00	2.20	36.81%	MLP

is larger than 10%. The other benchmarks are classified as ILP-intensive benchmarks. We will use this benchmark classification later in this article when evaluating the impact of our MLP-aware fetch policies on various mixes of workloads.

3. IMPACT OF MLP ON SMT PERFORMANCE

When running multiple threads on an SMT processor, there are two ways cache behavior affects overall performance. First, coscheduled threads affect each other's cache behavior as they compete for the available resources in the cache. In fact, one thread with poor cache behavior may evict data from the cache detoriating the performance of the other coscheduled thread(s). Second, memory-bound threads can hold critical execution resources while not making any progress because of the long-latency memory accesses. In particular, a long-latency load cannot be committed as long as the miss is not resolved. In the meantime, the fetch policy keeps on fetching instructions from the blocking thread. As a result, the blocking thread allocates execution resources without making any further progress. This article deals with the latter problem of long-latency threads holding execution resources.

The ICOUNT fetch policy [Tullsen et al. 1996], which fetches instructions from the thread(s) least represented in the front-end pipeline and the instruction queues, partially addresses this issue. ICOUNT tries to balance the number of instructions in the pipeline among the various threads so that all threads have an approximately equal number of instructions in the front-end pipeline and instruction queues. As such, the ICOUNT mechanism already limits the impact long-latency loads have on overall performance—the stalled thread is most likely to consume only a part of the resources. Without ICOUNT, the stalled thread is likely to allocate even more resources.

Tullsen and Brown [2001] recognize the problem of long-latency loads and, therefore, propose two mechanisms to free the allocated resources by the stalled thread. In their first approach, called *stall*, they prevent the thread executing a long-latency load to fetch any new instructions until the miss is resolved. The second mechanism, called *flush*, goes one step further and also flushes instructions from the pipeline. These mechanisms allow the other thread(s) to allocate execution resources while the long-latency load is being resolved; this improves the performance of the non-stalled thread(s). Cazorla et al. [2004a] improve the mechanism proposed by Tullsen and Brown by predicting long-latency loads. When a load is predicted to be long-latency, the thread is prevented from fetching additional instructions.

The ICOUNT and long-latency aware fetch policies do not completely solve the problem though, the fundamental reason being that they do not take into account MLP. Upon a long-latency load, the thread executing the long-latency load is prevented from fetching new instructions and (in particular implementations) may even be (partially) flushed. As a result, independent long-latency loads that are close to each other in the dynamic instruction stream cannot execute in parallel. In fact, they are serialized by the fetch policy. This excludes MLP from being exposed and thus penalizes threads that show a large amount of MLP. The MLP-aware fetch policy, which we discuss in great detail in Section 4, alleviates this issue and results in improved performance for MLP-intensive threads.

4. MLP-AWARE FETCH POLICY FOR SMT PROCESSORS

The MLP-aware fetch policy that we propose in this article consists of three mechanisms. First, we identify long-latency loads, or alternatively, we predict whether a given load is likely to be long latency. Second, once the long-latency load is identified or predicted, we predict the load's MLP distance. Third, we drive the fetch policy, using the predicted MLP distance. These three mechanisms will now be discussed in more detail.

4.1 Identifying Long-Latency Loads

We use two mechanisms for identifying long-latency loads—these two mechanisms will be used in conjunction with two different mechanisms for driving the fetch policy, as will be discussed in Section 4.3. The first mechanism simply labels a load as a long-latency load as soon as the load is found out to be an L3 miss or a D-TLB miss.

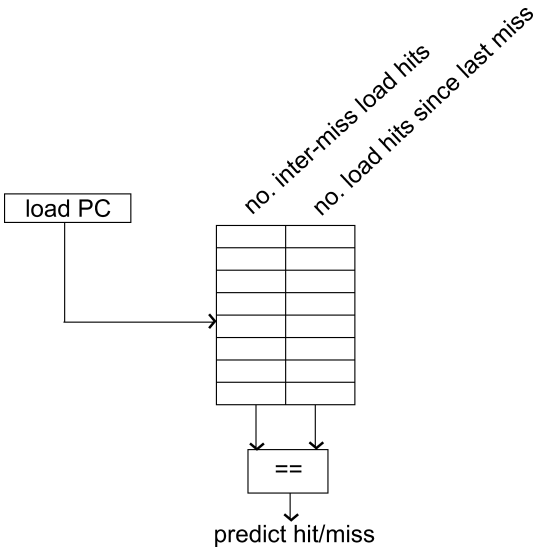

Fig. 2. Long-latency load miss pattern predictor.

The second mechanism is to predict whether a load is going to be a long-latency load. The predictor is placed at the pipeline front-end, and long-latency loads are predicted as they traverse the front-end pipeline. We use the miss pattern predictor proposed by Limousin et al. [2001] shown in Figure 2. The miss pattern predictor consists of a table indexed by the load PC; each table entry records (i) the number of load hits (by the same static load) between the two most recent long-latency loads and (ii) the number of load hits (by the same static load), since the last long-latency load. In case the latter matches the former, that is, in case the number of load hits since the last long-latency load equals the most recently observed number of load hits between two long-latency loads, the load is predicted to be a long-latency load. The predictor table is updated when a load executes. This predictor, thus, basically is a last value predictor for the number of hits between two long-latency misses per static load. The predictor used in our experiments is a 2K-entry table with 6 bits per entry (the total hardware cost is 12Kbits); and we assume one table per thread. During our experimental evaluation, we explored a wide range of long-latency load predictors, such as a last value predictor and the 2-bit saturating counter load miss predictor proposed by El-Moursy and Albonesi [2003]. We concluded, though, that the miss pattern predictor outperforms the other predictors—this conclusion was also reached by Cazorla et al. [2004a]. Note that a load hit/miss predictor has been implemented in commercial processors, as is the case in the Alpha 21264 microprocessor [Kessler et al. 1998] for predicting whether to speculatively issue load-consumers.

4.2 Predicting MLP

Once a long-latency load is identified, either through the observation of a long-latency cache miss or through prediction, we need to predict whether the load is

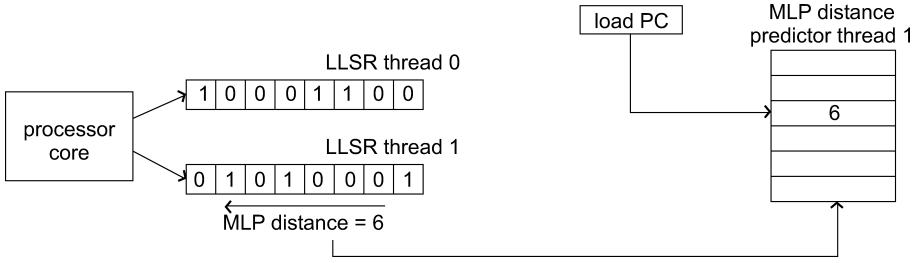

Fig. 3. Updating the MLP distance predictor.

exhibiting MLP. The MLP distance predictor that we propose consists of a table indexed by the load PC. Each entry in the table contains the MLP distance, or the number of instructions one needs to go down the dynamic instruction stream in order to observe the maximum MLP for the given ROB size. We assume one MLP distance predictor per thread; the MLP distance predictor is assumed to have 2K entries and a total hardware cost of 14Kbits in this article.

Updating the MLP distance predictor, as illustrated in Figure 3, is done using a structure called the long-latency shift register (LLSR). The LLSR has as many entries as there are ROB entries divided by the number of threads (assuming a shared ROB), and there are as many LLSRs as there are threads. Upon committing an instruction from the ROB, we shift the LLSR over 1 bit position from tail to head, and then insert 1 bit at the tail of the LLSR. The bit being inserted is a "1" in case the committed instruction is a long-latency load and a "0" if not. Along with inserting a "0" or a "1", we also keep track of the load PCs in the LLSR. In case a "1" reaches the head of the LLSR, we update the MLP predictor table. This is done by computing the MLP distance, which is the bit position of the last appearing "1" in the LLSR when reading the LLSR from head to tail. The MLP distance then is the number of instructions one needs to go down the dynamic instruction stream in order to achieve the maximum MLP for the given ROB size. In the example given in Figure 3, the MLP distance equals 6. The MLP distance predictor is updated by inserting the computed MLP distance in the predictor table entry pointed to by the long-latency load PC. In other words, the MLP distance predictor proposed here is a fairly simple last value predictor: The most recently observed MLP distance is stored in the predictor table. The total hardware cost for the LLSR structures equals 1.6Kbits in our setup: 128 bits per thread (for the shift register) plus 128 times 11 bits (for keeping track of the load PC indexes in the 2K-entry MLP distance predictor), assuming a 256-entry ROB, 128-entry load/store queue processor configuration. According to our experimental results, this predictor performs well for our purpose.

Figure 4 shows the cumulative distribution of the predicted MLP distance for the six most MLP-intensive programs assuming a 256-entry ROB processor with a 128-entry LLSR. We observe a wide range of MLP distance characteristics. For example, mcf and fma3d have a large predicted MLP distance (more than 100 instructions), which implies that MLP is to be exploited at

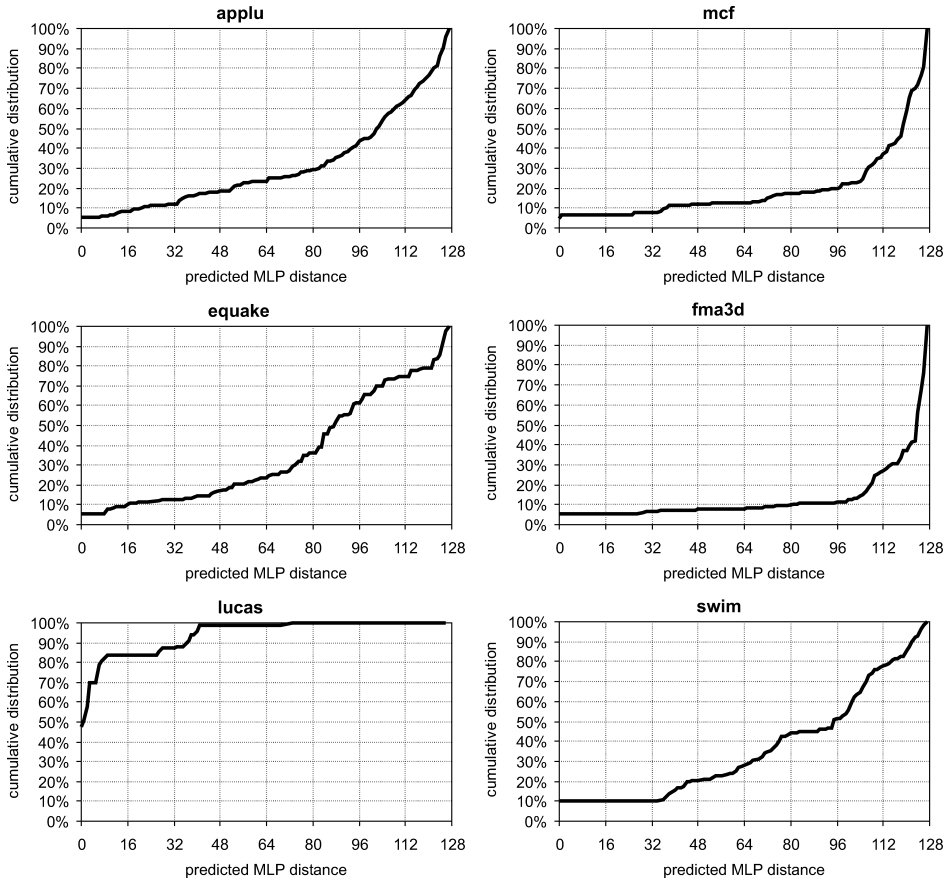

Fig. 4. Cumulative distribution of the predicted MLP distance for six MLP-intensive programs assuming a 256-entry ROB processor with a 128-entry LLSR.

large distances; lucas on the other hand, has all of its MLP over short distances, that is, nearly 100% of the exploitable MLP is to be observed at an MLP distance of less than 40 instructions; equake is an average example, which finds its exploitable MLP over a distance that is smaller than 90 instructions 50% of the time. These results suggest that an MLP predictor may improve resource utilization in an SMT processor: Only a fraction of the resources need to be allocated for exposing the exploitable MLP while providing the remaining resources to the other thread(s).

Note that the LLSR in the implementation evaluated in this article does not make a distinction between dependent and independent long-latency loads. The "1s" inserted in the LLSR represent a long-latency load irrespective of the fact whether these long-latency loads are dependent or independent from each other. By consequence, in case these long-latency loads are independent, the resulting MLP distance corresponds to the actual MLP available. If, on the other hand, these long-latency loads are dependent upon each other, the MLP

distance will overestimate the available MLP. The overestimation may be small though in case the last dependent and independent long-latency loads appear close to each other in the dynamic instruction stream. An interesting avenue for future work may be to exclude dependent loads when computing the MLP distance.

4.3 MLP-Aware Fetch Policy

We consider two mechanisms for driving the MLP-aware fetch policy, namely stall fetch and flush. These two mechanisms are similar to the ones proposed by Tullsen and Brown [2001] and Cazorla et al. [2004a]; however, these previous approaches did not consider MLP.

In the stall-fetch approach, we first predict in the front-end pipeline whether a load is going to be a long-latency load. In case of a predicted long-latency load, we then predict the MLP distance, say m instructions. We then fetch stall after having fetched m additional instructions.

The flush approach is slightly different. We first identify whether a load is a long-latency load. This is done by observing whether the load is an L3 miss or a D-TLB miss; there is no long-latency load prediction involved. For a long-latency load, we then predict the MLP distance m. If more than m instructions have been fetched since the long-latency load, say n instructions, we flush the last n - m instructions fetched. If less than m instructions have been fetched since the long-latency load, we continue fetching instructions until m instructions have been fetched. Note that the flush mechanism requires that the microarchitecture supports checkpointing. Commercial processors, such as the Alpha 21264 [Kessler et al. 1998], effectively support checkpointing at all instructions. If the microprocessor would only support checkpointing at branches for example, our flush mechanism could flush instructions starting from the next branch after the m instructions.

Our MLP-aware fetch policies also implement the continue the oldest thread (COT) mechanism proposed by Cazorla et al. [2004a]. COT means that in case all threads stall because of a long-latency load, the thread that stalled first gets priority for allocating resources. The idea is that the thread that stalled first will be the first thread to get the data back from memory and continue execution.

Note also that the proposed MLP-aware fetch policies resort to the ICOUNT fetch policy in the absence of long-latency loads.

5. EXPERIMENTAL SETUP

We use the SPEC CPU2000 benchmarks in this article with reference inputs (see Table I). These benchmarks are compiled for the Alpha ISA, using the Compaq C compiler (`cc`) version V6.3-025 with the `-O4` optimization option. For all of these benchmarks, we select 200M instruction (early) simulation points, using the SimPoint tool [Sherwood et al. 2002; Perelman et al. 2003]. We use a wide variety of randomly selected two-thread and four-thread workloads. The two-thread workloads are given in Table II. These two-thread workloads are classified as ILP-intensive, MLP-intensive, or mixed ILP/MLP-intensive workloads.

Table II. The Two-Thread Workloads Used in the Evaluation, Divided into Three Categories: ILP-Intensive Workloads, MLP-Intensive Workloads, and Mixed ILP/MLP-Intensive Workloads

ILP-intensive
vortex-parser
crafty-twolf
facerec-crafty
vpr-sixtrack
vortex-gcc
gcc-gap

MLP-intensive
apsi-mesa
mcf-swim
mcf-galgel
wupwise-ammp
swim-galgel
lucas-fma3d
mesa-galgel
galgel-fma3d
applu-swim
mcf-equake
applu-galgel
swim-mesa

Mixed ILP-MLP
swim-perlbmk
galgel-twolf
fma3d-twolf
apsi-art
gzip-wupwise
apsi-twolf
mgrid-vortex
swim-twolf
swim-eon
swim-facerec
parser-wupwise
vpr-mcf
equake-perlbmk
applu-vortex
art-mgrid
equake-art
parser-ammp
facerec-mcf

Table III. The Four-Thread Workloads Used in the Evaluation, Sorted by the Number of MLP-Intensive Benchmarks in the Workload

#MLP	Workload
0	vortex-parser-crafty-twolf facerec-crafty-vpr-sixtrack
1	swim-perlbmk-vortex-gcc galgel-twolf-gcc-gap fma3d-twolf-vortex-parser apsi-art-crafty-twolf gzip-wupwise-facerec-crafty apsi-twolf-vpr-sixtrack
2	mgrid-vortex-swim-twolf swim-eon-perlbmk-mesa parser-wupwise-vpr-mcf equake-perlbmk-applu-vortex art-mgrid-applu-galgel parser-ammp-facerec-mcf swim-perlbmk-galgel-twolf
2	fma3d-twolf-apsi-art gzip-wupwise-apsi-twolf equake-art-parser-ammp
3	apsi-mesa-swim-eon mcf-swim-perlbmk-mesa mcf-galgel-vortex-gcc wupwise-ammp-vpr-mcf swim-galgel-parser-wupwise lucas-fma3d-equake-perlbmk mesa-galgel-applu-vortex
4	galgel-fma3d-art-mgrid applu-swim-mcf-equake applu-galgel-swim-mesa apsi-mesa-mcf-swim mcf-galgel-wupwise-ammp

The four-thread workloads are shown in Table III. These workloads vary from ILP-intensive, over-mixed ILP/MLP-intensive workloads, to MLP-intensive workloads.

We use the SMTSIM simulator v1.0 [Tullsen 1996] in all of our experiments. The processor model being simulated is the 4-wide superscalar out-of-order SMT processor shown in Table IV. The default fetch policy is ICOUNT 2.4 [Tullsen et al. 1996], which allows up to four instructions from up to two threads to be fetched per cycle. We added a write buffer to the simulator's processor model: Store operations leave the ROB upon commit and wait in

Table IV. The Baseline SMT Processor Configuration

Parameter	Value
fetch policy	ICOUNT 2.4
pipeline depth	14 stages
(shared) reorder buffer size	256 entries
(shared) load/store queue	128 entries
instruction queues	64 entries in both IQ and FQ
rename registers	100 integer and 100 floating-point
processor width	4 instructions per cycle
functional units	4 int ALUs, 2 ld/st units and 2 FP units
branch misprediction penalty	11 cycles
branch predictor	2K-entry gshare
branch target buffer	256 entries, 4-way set associative
write buffer	8 entries
L1 instruction cache	64KB, 2-way, 64-byte lines
L1 data cache	64KB, 2-way, 64-byte lines
unified L2 cache	512KB, 8-way, 64-byte lines
unified L3 cache	4MB, 16-way, 64-byte lines
instruction/data TLB	128/512 entries, fully-assoc, 8KB pages
cache hierarchy latencies	L2 (11), L3 (35), MEM (350)
hardware prefetcher	8 stream buffers, 8 entries each, w/ stride predictor

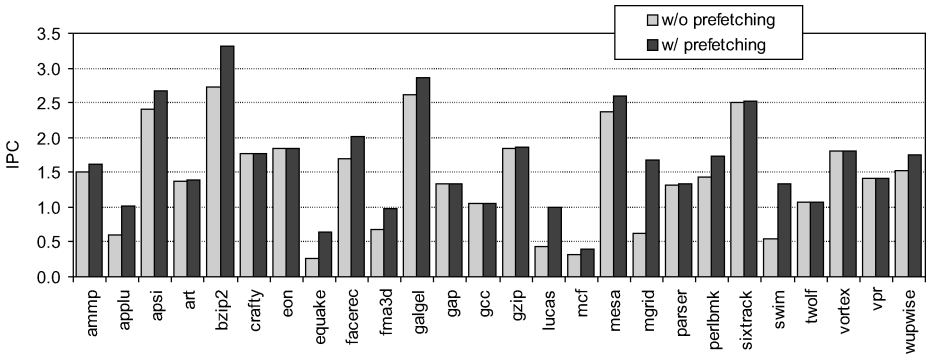

Fig. 5. Graph showing performance (IPC) for the baseline processor configuration running single-threaded workloads both with and without hardware prefetching.

the write buffer for writing to the memory subsystem; commit blocks in case the write buffer is full and we want to commit a store. The baseline SMT processor configuration contains an aggressive hardware prefetcher consisting of 8 stream buffers, 8 entries each. The stream buffers are guided by a 2K-entry stride predictor indexed by the load PC, and stream buffers are allocated using the confidence scheme described by Sherwood et al. [2000]. Figure 5 shows single-threaded performance for all the benchmarks with and without hardware prefetching. The (harmonic) average IPC speed-up achieved through this hardware prefetcher equals 20.2%. We observe large performance improvements for some of the benchmarks (see e.g., bzip2, applu, equake, lucas, and mgrid).

We use two system-level performance metrics in our evaluation: STP and ANTT [Eyerman and Eeckhout 2008]. STP is a system-oriented metric, which measures the number of jobs completed per unit of time and is defined as:

$$STP = \sum_{i=1}^{n} \frac{CPI_i^{ST}}{CPI_i^{MT}},$$

with CPI_i^{ST} and CPI_i^{MT}, the cycles per instruction achieved for program i during single-threaded and multithreaded execution, respectively; there are n threads running simultaneously. STP is a higher-is-better metric and corresponds to the weighted speed-up metric proposed by Snavely and Tullsen [2000].

ANTT is a user-oriented metric, which quantifies the average user-perceived slowdown due to multithreading. ANTT is computed as

$$ANTT = \frac{1}{n} \sum_{i=1}^{n} \frac{CPI_i^{MT}}{CPI_i^{ST}}.$$

ANTT equals the reciprocal of the hmean metric proposed by Luo et al. [2001] and is a lower-is-better metric. In our earlier work [Eyerman and Eeckhout 2008], we argued that both STP and ANTT should be reported in order to gain insight into how a given multithreaded architecture affects system-perceived and user-perceived performance, respectively.

When simulating a multiprogram workload, simulation stops when one of the coexecuting programs, say program j, has executed 200 million instructions. The other thread(s) $i \neq j$ will then have reached x_i million instructions (less than 200 million instructions); the single-threaded CPI_i^{ST} used in the above formulas then equals single-threaded CPI after x_i million instructions. When we report average STP and ANTT numbers across a number of multiprogram workloads, we use the harmonic and arithmetic mean for computing the average STP and ANTT, respectively, following the recommendations on the use of averages by John [2006].

6. EVALUATION

The evaluation of the MLP-aware SMT fetch policy is done in a number of steps. We first evaluate the prediction accuracy of the long-latency load predictor. We subsequently evaluate the prediction accuracy of the MLP predictor. We then evaluate the effectiveness of the MLP-aware fetch policy and compare it against prior work. And finally, we study the impact of the microarchitecture on the performance of an MLP-aware SMT fetch policy, explore variations of the MLP-aware policy proposed in this article, and compare against static and dynamic resource partitioning.

6.1 Long-Latency Load Predictor

An MLP-aware stall fetch policy requires that we can predict long-latency loads in the front-end stages of the processor pipeline. Figure 6 shows the prediction accuracy for the 2K-entry 12Kbits long-latency load predictor that is, the number of correct hit/miss predictions divided by the number of load instructions.

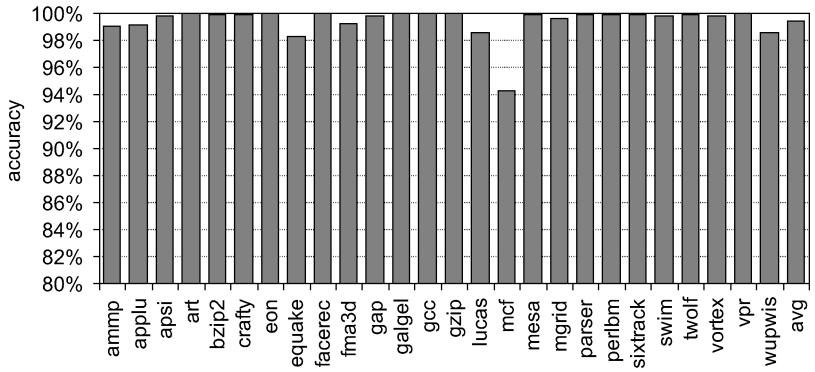

Fig. 6. The accuracy of the long-latency load predictor: the number of correct hit/miss predictions per load.

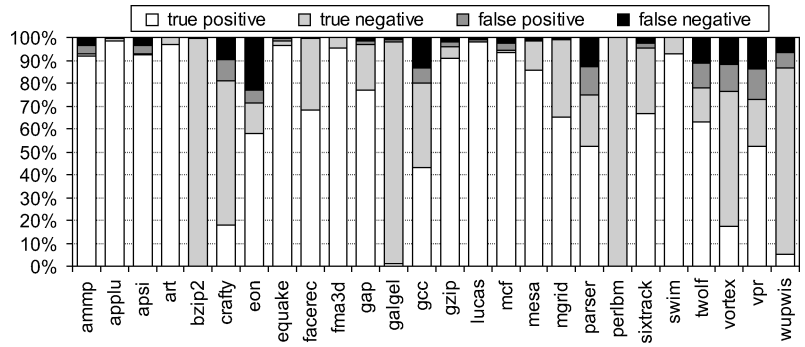

Fig. 7. Evaluating the accuracy of the MLP predictor for predicting MLP.

We observe that the accuracy achieved is very high, no less than 94%, with an average prediction accuracy of 99.4%.

The number of correct miss predictions divided by the number of misses is also high. For the memory-intensive benchmarks (with at least one miss every 200 instructions), we achieve a prediction accuracy of at least 85% and up to 99% (for applu, equake, fma3d, lucas, mgrid, and swim); the only exception is mcf for which the predictor achieves a prediction accuracy of 59%.

6.2 MLP Predictor

We now evaluate the effectiveness of the MLP predictor. This is done in two steps. We first evaluate whether the MLP predictor can make an accurate (binary) MLP prediction. Subsequently, we evaluate whether the MLP predictor can accurately predict the MLP distance.

Figure 7 evaluates the ability of the MLP predictor for predicting whether a long-latency load is going to expose MLP (i.e., is the predicted MLP distance zero in case the actual MLP distance is zero, and is the predicted MLP distance nonzero in case the actual MLP distance is nonzero?). A true positive means the MLP predictor predicts MLP in case there is MLP; a true negative means

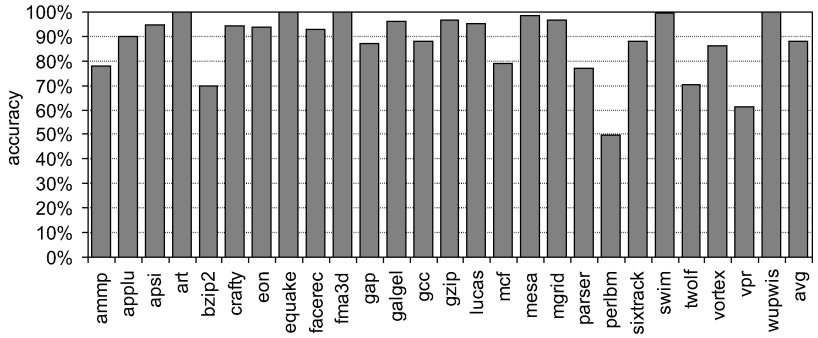

Fig. 8. Evaluating the accuracy of the MLP predictor for predicting the MLP distance: Predicting a far enough MLP distance is counted as a correct prediction.

the MLP predictor predicts no-MLP in case there is no MLP. The sum of the fraction of true positives and true negatives is the prediction accuracy of the MLP predictor in predicting MLP. The predictor evaluated in this experiment is a 2K-entry table in which each entry contains $log_2\lceil\frac{ROB\_size}{n}\rceil$ bits (7 bits in our setup); the entire predictor thus requires 14Kbits of storage. The average prediction accuracy equals 91.5%. The average fraction of false negatives equals 4.8% and corresponds to the case where the MLP predictor fails to predict MLP. This case will lead to performance loss for the MLP-intensive thread, that is, the thread will be fetch stalled or flushed although there is MLP to be exploited. The average fraction of false positives equals 3.7% and corresponds to the case where the MLP predictor fails to predict there is no MLP. In this case, the fetch policy will allow the thread to allocate additional resources although there is no MLP to be exposed; this may hurt the performance of the other thread(s).

Figure 8 further evaluates the MLP predictor and quantifies the probability for the MLP predictor to predict a far enough MLP distance. In other words, a prediction is classified as a misprediction if the predicted MLP distance is smaller than the actual MLP distance (i.e., the maximum available MLP is not fully exposed by the MLP predictor). A prediction is classified as a correct prediction if the predicted MLP distance is at least as large as the actual MLP distance. This classification of correct versus incorrect predictions gives emphasis on the ability of the MLP predictor to expose MLP rather than to preserve resources for the other thread(s). The average MLP distance prediction accuracy equals 87.8%.

6.3 MLP-Aware Fetch Policy

We now evaluate the proposed MLP-aware fetch policies in terms of the STP and ANTT metrics. For doing so, we compare the following SMT fetch policies:

—ICOUNT, which strives at having an equal number of instructions from all threads in the front-end pipeline and instruction queues. The following fetch policies extend upon the ICOUNT policy.

—The stall fetch approach proposed by Tullsen and Brown [2001] (i.e., a thread that experiences a long-latency load is fetch stalled until the data returns from memory).
—The predictive stall fetch approach, following Cazorla et al. [2004a], extends the above stall fetch policy by predicting long-latency loads in the front-end pipeline. Predicted long-latency loads trigger fetch stalling a thread.
—The MLP-aware stall fetch approach predicts long-latency loads, predicts the MLP distance, and fetch stalls threads when the number of instructions has been fetched as predicted by the MLP predictor.
—The flush approach proposed by Tullsen and Brown [2001] flushes on long-latency loads. Our implementation flushes when a long-latency load is detected (this is the "TM" or trigger on long-latency miss by Tullsen and Brown [2001]) and flushes starting from the instruction following the long-latency load (this is the "next" approach by Tullsen and Brown [2001]).
—The MLP-aware flush approach predicts the MLP distance m for a long-latency load, and fetch stalls or flushes the thread after m instructions since the long-latency load.

Note that all of these fetch policies also include the COT mechanism proposed by Cazorla et al. [2004a] in case all threads stall on a long-latency load.

6.3.1 *Two-Thread Workloads.* Figures 9 and 10 show STP and ANTT, for the various SMT fetch policies for the 2-thread workloads, respectively. There are three graphs in Figures 9 and 10: one for the ILP-intensive workloads (top graph), one for the MLP-intensive workloads (middle graph), and one for the mixed ILP/MLP-intensive workloads (bottom graph). There are several interesting observations to be made from these graphs. First, the flush policies generally outperform the stall fetch policies. This is in line with the observations made by Tullsen and Brown [2001] and is explained by the fact that the flush policy is able to free resources allocated by a stalled thread. Second, for ILP-intensive workloads, the MLP-aware flush policy achieves a similar STP and ANTT as flush and achieves an, on average, 6.4% higher STP and 5.1% lower ANTT than ICOUNT. Third, for MLP-intensive workloads, the MLP-aware flush policy achieves an, on average, 20.2% better STP and 21.0% better ANTT than ICOUNT; and a 5.1% better STP and 18.8% better ANTT than flush. Fourth, for mixed ILP/MLP-intensive workloads, the MLP-aware flush policy improves STP by 22.4% over ICOUNT on average and by 4.0% over flush, on average. Likewise, the MLP-aware flush policy improves ANTT by 19.2%, on average, over ICOUNT and by 13.9% over flush. The bottom line from the performance data presented in Figures 9 and 10 is that an MLP-aware fetch policy improves the performance of MLP-intensive threads and does not hurt the performance of ILP-intensive workload mixes. Or, in other words, for MLP-intensive and mixed ILP/MLP-intensive workloads, the MLP-aware flush policy improves STP slightly over flush (4.5%, on average) while improving a program's turnaround time substantially over flush (15.9%, on average).

This is further illustrated in Figures 11 and 12 where IPC stacks are shown for MLP-intensive and mixed ILP/MLP-intensive workloads, respectively.

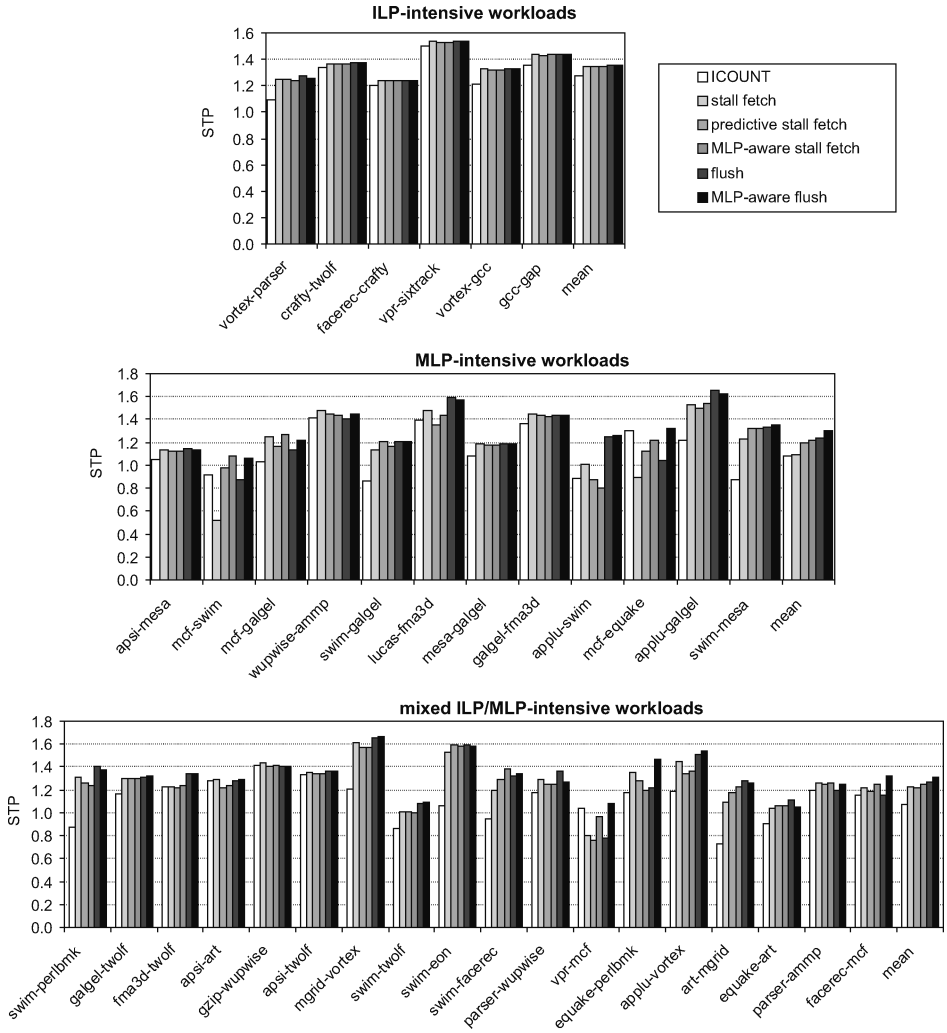

Fig. 9. STP for the various SMT fetch policies compared to single-threaded execution for the two-thread workloads: ILP-intensive workloads (top), MLP-intensive workloads (middle), and mixed ILP/MLP-intensive workloads (bottom).

These graphs show that an MLP-intensive thread typically achieves better performance under an MLP-aware fetch policy. One example illustrating the improved performance for an MLP-intensive thread is mcf-galgel (see Figure 11 [third cothread example]). The flush policy severely affects mcf's performance by not exploiting the MLP available for mcf. MLP-aware flush, on the other hand, enables exploiting mcf's MLP while giving more resources to galgel. As a result, the performance for mcf under MLP-aware flush is comparable to under ICOUNT, and the performance for galgel improves substantially compared to ICOUNT. This results in a 7.4% better STP (see Figure 9) as well as a 53% better ANTT (see Figure 10).

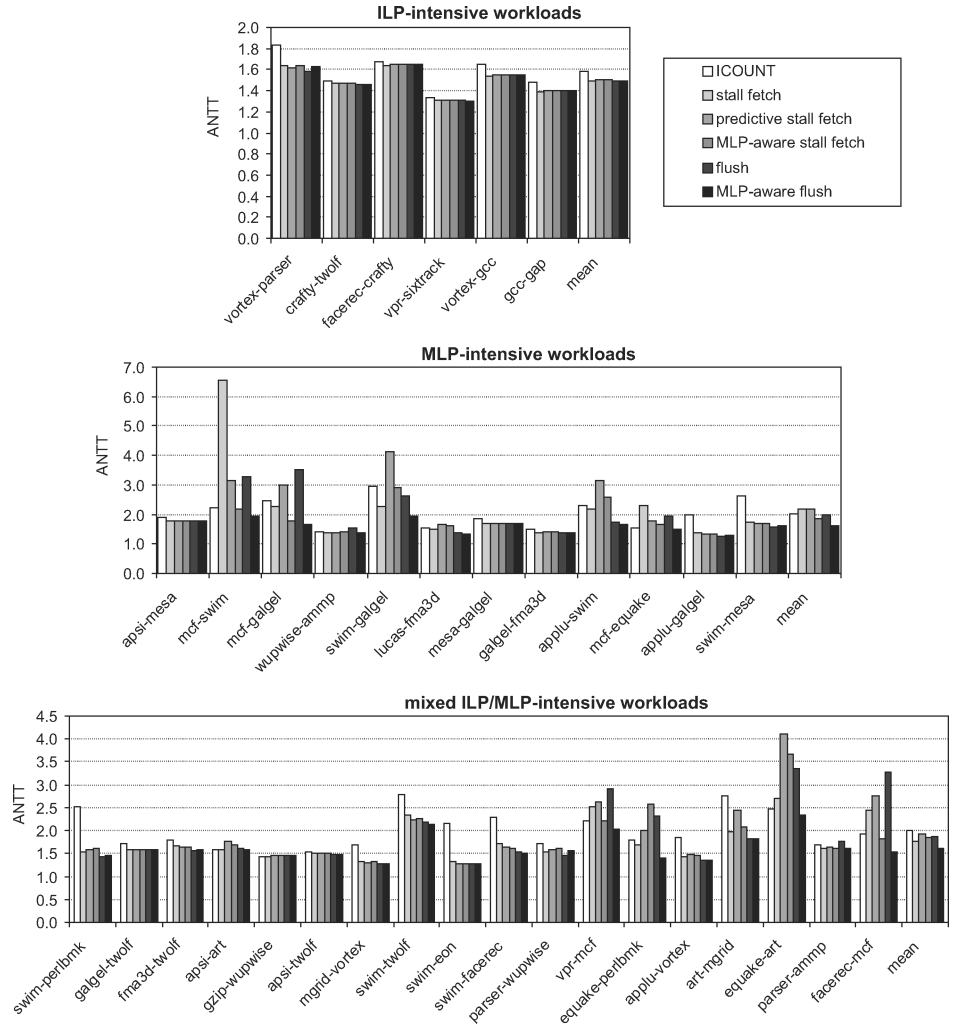

Fig. 10. ANTT for the various SMT fetch policies compared to single-threaded execution for the two-thread workloads: ILP-intensive workloads (top), MLP-intensive workloads (middle), and mixed ILP/MLP-intensive workloads (bottom).

6.3.2 *Four-Thread Workloads.* Figures 13 and 14 show STP and ANTT for the various fetch policies for the 4-thread workloads, respectively. Here we obtain fairly similar results as for the two-thread workloads. The MLP-aware fetch policies achieve a better normalized turnaround time than non-MLP-aware fetch policies. In particular, the MLP-aware flush policy achieves the overall best normalized turnaround time: ANTT for the MLP-aware flush, policy is 12.4% better than for ICOUNT, and 9.5% better than for flush; STP is comparable for flush and MLP-aware flush, which is approximately 16% better than for ICOUNT.

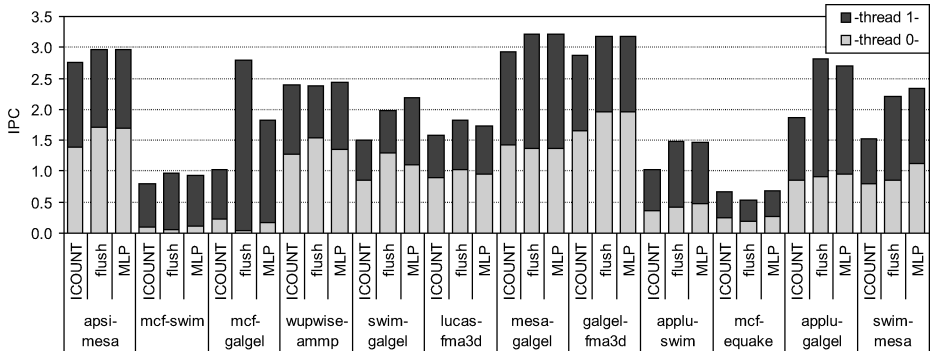

Fig. 11. IPC values for the two threads "thread 0—thread 1" for the MLP-intensive workloads.

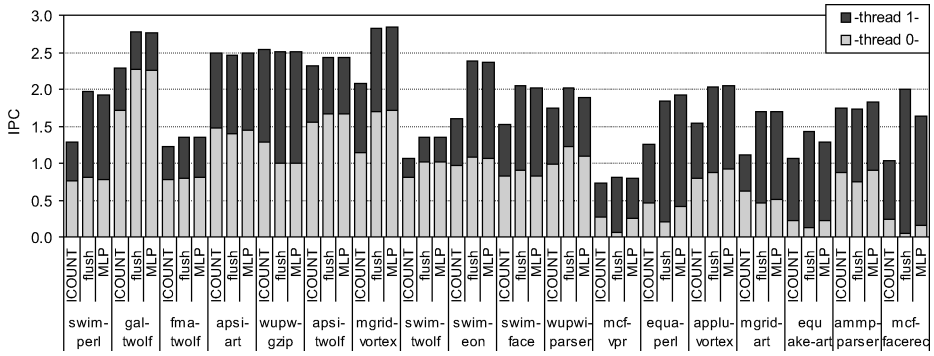

Fig. 12. IPC values for the two threads "thread 0—thread 1" for the mixed ILP/MLP-intensive workloads. Thread 0 is the MLP-intensive thread; thread 1 is the ILP-intensive thread.

6.4 Impact of Microarchitecture Parameters

In order to gain more insight into how an MLP-aware fetch policy is affected by the SMT processor's microarchitecture, we now study the impact of two major microarchitecture parameters that potentially have a large impact on the MLP-aware fetch policy's performance, namely main memory access latency and processor core buffer sizes.

6.4.1 *Memory Latency.* In our first experiment, we vary the main memory access latency while keeping the rest of the baseline processor configuration unchanged. We vary main memory access latency from 200 processor cycles up to 800 processor cycles, in steps of 200 cycles. The results are shown in Figures 15 and 16 for STP and ANTT relative to ICOUNT, respectively. MLP-aware flush policy is the clear winner, and its achieved throughput improves compared to ICOUNT with increasing main memory access latency. The reason is that a long-latency thread under ICOUNT holds more allocated resources for a longer period of time as main memory access latency increases. The MLP-aware flush policy, on the other hand, gives more resources to the other thread, yielding a better overall STP. A program's turnaround time achieved by the MLP-aware

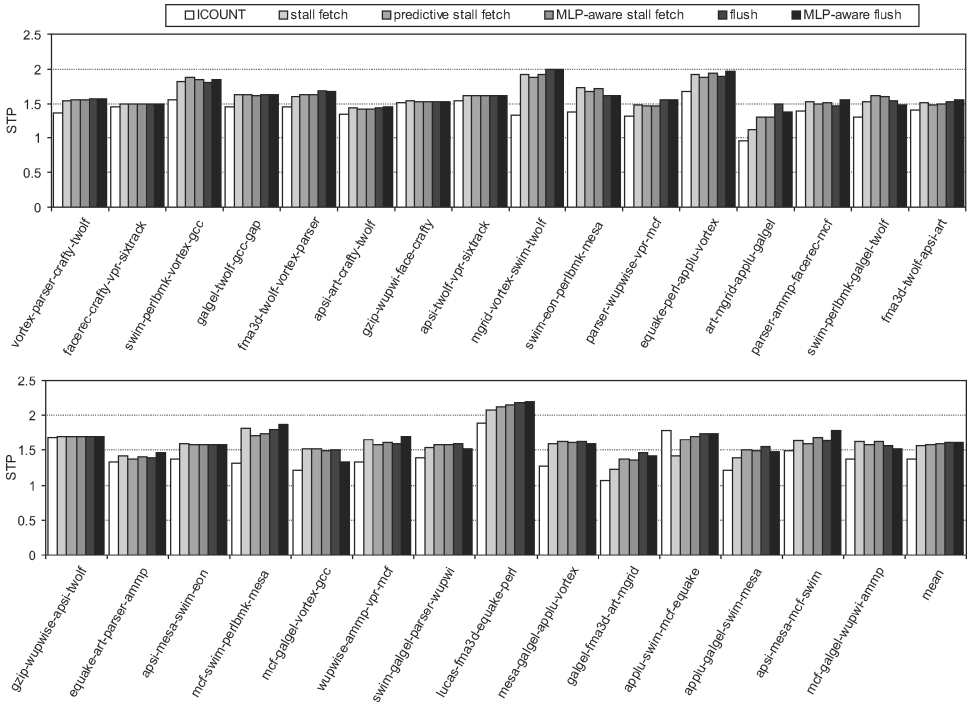

Fig. 13. STP for the various SMT fetch policies compared to single-threaded execution for the four-thread workloads.

flush policy improves compared to flush because the MLP-aware flush policy does not penalize MLP-intensive threads as much as the flush policy.

6.4.2 *Processor Core Structures.* In our second experiment, we vary the size of a number of processor core structures. We vary the ROB size, the load/store queue size, the integer and floating-point issue queue sizes, and the number of integer and floating-point rename registers. We consider four design points and vary the ROB size from 128, 256, 512, up to 1,024; we simultaneously vary the load/store queue size from 64, 128, 256, up to 512, the integer and floating-point issue queue sizes from 32, 64, 128, up to 256, as well as the number of integer and floating-point rename registers from 50, 100, 200, up to 400. The large sizings do not correspond to a realistic design point for a conventional ROB-based out-of-order processor, but merely serve as a proxy for a microarchitecture that strives at enlarging the instruction window size at reasonable hardware cost such as runahead execution [Mutlu et al. 2003; Mutlu et al. 2005], continual flow pipelines [Srinivasan et al. 2004], and kilo-instruction processors [Cristal et al. 2004].

Figures 17 and 18 show the results for STP and ANTT, respectively. The various fetch policies are compared relative to ICOUNT. We observe that the performance improvement of a long-latency load aware fetch policy improves with fewer resources (relative to ICOUNT). This is to be expected because the

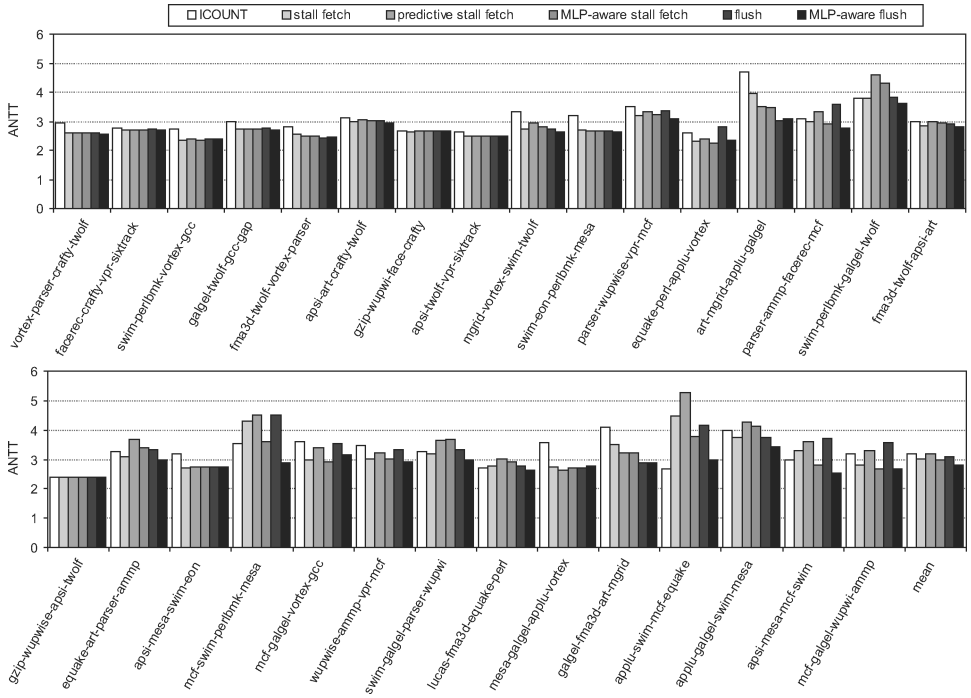

Fig. 14. ANTT for the various SMT fetch policies compared to single-threaded execution for the four-thread workloads.

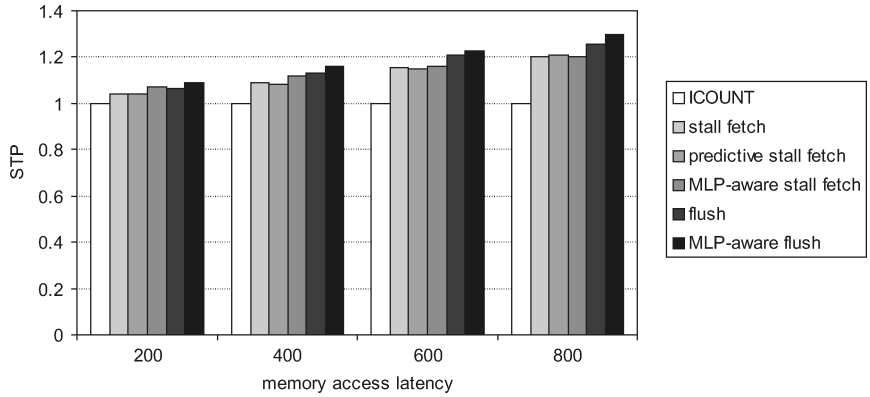

Fig. 15. STP for the various SMT fetch policies as a function of memory access latency.

goal of the long-latency aware fetch policies is to allocate fewer resources in case of a long-latency load. Also, the performance of an MLP-aware fetch policy improves compared to a non-MLP-aware fetch policy with increased window resources; compare the relative ANTT performance differences between MLP-aware stall fetch versus predictive stall fetch, and MLP-aware flush versus flush. The reason is that as the window resources increase, there is more MLP to be exploited and the MLP-aware fetch policy better exploits the available MLP.

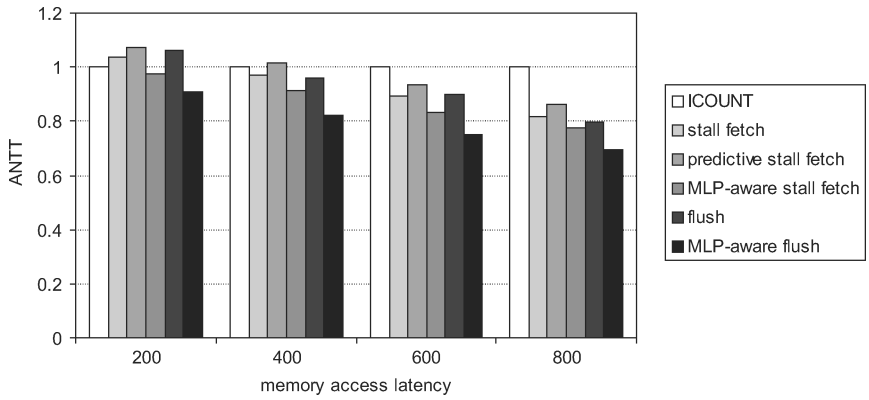

Fig. 16. ANTT for the various SMT fetch policies as a function of memory access latency.

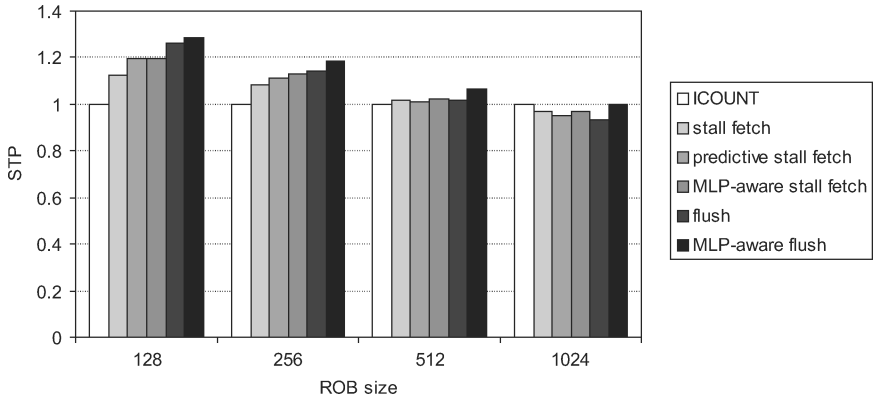

Fig. 17. STP for the various SMT fetch policies as a function of processor window size; the load/store queue, issue queues, and register files are scaled proportionally.

6.5 Alternative MLP-Aware Fetch Policies

To gain more insight into the design trade-offs for an MLP-aware fetch policy, we now consider a number of MLP-aware fetch policy alternatives. To facilitate the discussion, the five alternatives considered here are schematically presented in Figure 19. We consider the following five alternatives:

(a) The first alternative is the flush fetch policy as proposed by Tullsen and Brown [2001]. Upon the detection of a long-latency load, the instructions fetched after the long-latency load are flushed from the pipeline.

(b) The second alternative is the MLP distance + flush policy, which is the MLP-aware fetch policy evaluated throughout this article: Upon the detection of a long-latency load, the MLP distance is predicted, and the pipeline is fetch stalled or flushed per the predicted MLP distance.

(c) The third alternative, MLP + flush assumes a binary MLP predictor that predicts whether there is MLP to be exploited but does not predict the

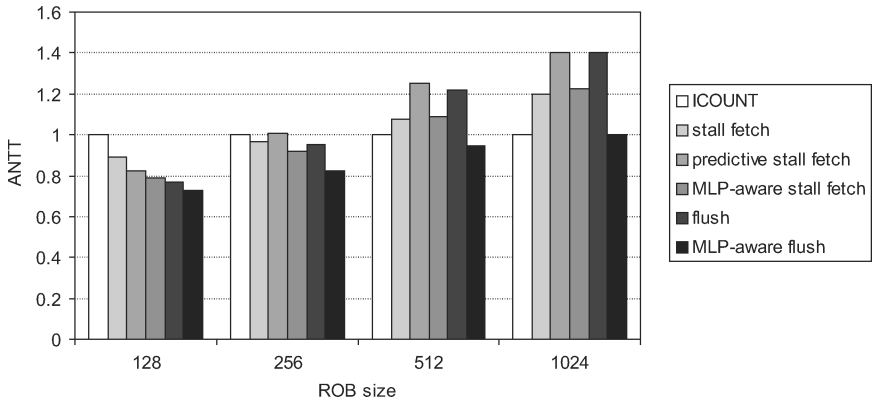

Fig. 18. ANTT for the various SMT fetch policies as a function of processor window size; the load/store queue, issue queues, and register files are scaled proportionally.

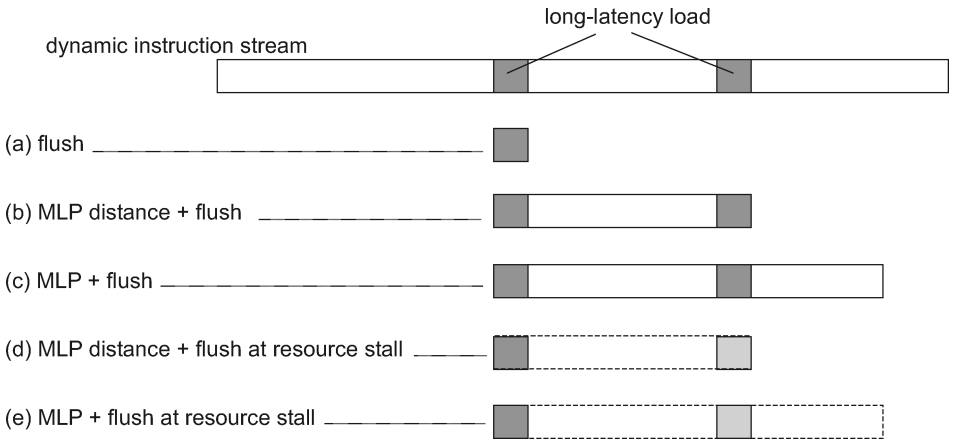

Fig. 19. A schematic representation of five alternative MLP-aware fetch policies.

MLP distance. Each entry in the binary MLP predictor is a 1-bit entry that keeps track of whether MLP was observed in the previous occurrence of a long-latency miss of that same static load. In case no MLP is predicted, the long-latency thread is flushed. In case MLP is predicted, no flush will occur and fetching will continue past long-latency loads following the ICOUNT principle.

(d) The fourth alternative, MLP distance + flush at resource stall, uses an MLP distance predictor that predicts how far down the instruction stream we need to fetch instructions. Once the number of instructions determined by the predicted MLP distance are fetched, we fetch stall the given thread. If at some later cycle, a resource stall occurs—none of the threads can make progress because of a full issue queue, ROB, or no more available rename registers—the thread is flushed past the initial long-latency load; this is illustrated in Figure 19 through the dashed lines. The intuition behind this

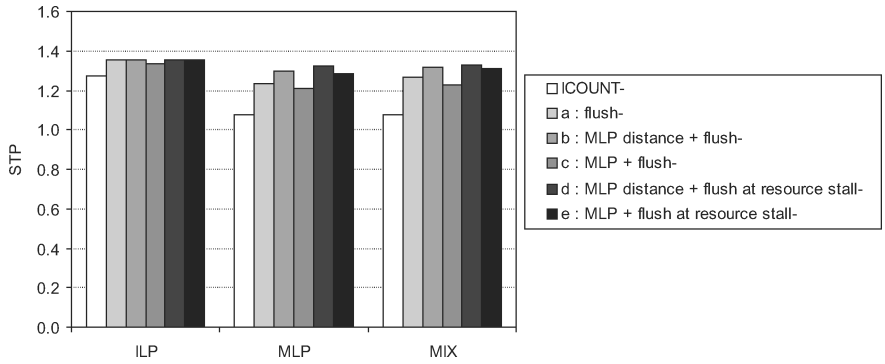

Fig. 20. Evaluating alternative MLP-aware flush policies in terms of STP.

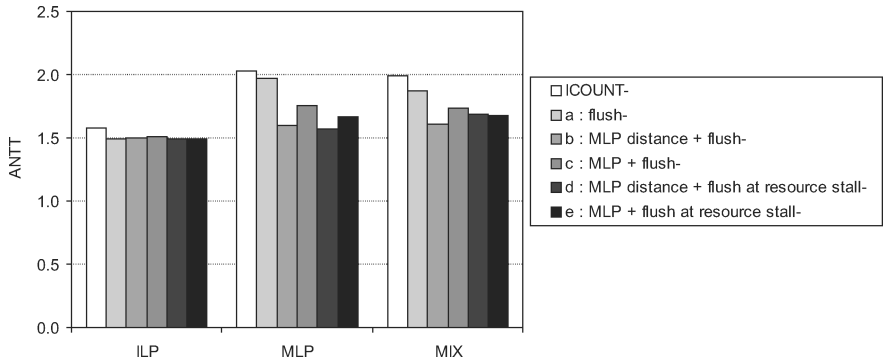

Fig. 21. Evaluating alternative MLP-aware flush policies in terms of ANTT.

scheme is to free resources to be used by other threads, while still exploitings MLP: Independent long-latency loads most likely will have started execution and their latencies will overlap. When the initial long-latency load returns, fetching resumes and the load instruction, which was a long-latency load previously (the light gray box in Figure 19), is likely going to be a hit—there is a prefetching effect. Comparing (d) against (b), the trade-off is that, under (d), instructions need to be refetched and reexecuted, which is not the case under (b). On the other hand, under (d), more resources will be made available to the other threads upon a resource stall.

(e) The fifth alternative, MLP + flush at resource stall, combines the binary MLP predictor with the flush at resource stall policy.

Figures 20 and 21 evaluate these alternative MLP-aware fetch policies and quantify STP and ANTT, respectively. The various bars represent the alternative fetch policies for the three two-thread workload groups, ILP-intensive, MLP-intensive, and mixed ILP/MLP-intensive. There are three interesting observations to be made. First, for the *flush* policies, (b) and (c), it is important to predict the MLP distance rather than to resort to a binary MLP prediction:

Predicting the MLP distance and fetch stalling (or flushing) past the predicted MLP distance, as done under fetch policy (b), prevents the long-latency thread from allocating and holding more resources compared to (c). As such, fetch policy (b) is a better design option than (c). Second, also for the flush at resource stall fetch policies, (d) and (e), predicting the MLP distance is a good design option (in general), however, the reason why (d) outperforms (e) is different. Fetch policy (e) will continue fetching instructions past long-latency loads, even past the last long-latency load in a burst of long-latency loads. This will result in more resource stalls and, by consequence, more flushes than under (d). As a result, fetch policy (e) suffers more frequently from the overhead of refetching flushed instructions than (d). There are cases, however, where fetch policy (d) performs less than (e), namely in case of an incorrect MLP distance prediction: An incorrect MLP distance prediction under (d) leads to missed MLP exploitation opportunities, whereas (e) fully exploits these MLP exploitation opportunities. Third, and finally, comparing the (winner) fetch policies (b) "MLP distance + flush" against (d) "MLP distance + flush at resource stall," it follows that (d) outperforms (b) for the MLP-intensive workloads. Under (d), an MLP-intensive thread will be able to exploit the available MLP and will then flush the allocated resources on a resource stall so that the other MLP-intensive thread can allocate as many resources as possible to exploit its MLP. For mixed ILP/MLP-intensive workloads, on the other hand, the ILP-intensive thread does not require as many resources as an MLP-intensive thread and does not require flushing all the allocated resources by the MLP-intensive thread, and thus (b) is a better design option than (d).

6.6 Comparison Against Static and Dynamic Partitioning

So far, we assumed an SMT processor architecture in which the resources are managed implicitly by the fetch policy, that is, the fetch policy determines which thread to fetch instructions from, and, once fetched, the instructions compete for the shared resources such as ROB entries, issue queue entries, and so on. An alternative approach is to explicitly manage the available resources. There are two ways for explicit resource management. One approach is to statically partition the resources [Raasch and Reinhardt 2003], as done in the Intel Pentium 4, that is, each thread in an n-threaded SMT processor gets a $1/n$ share of the resources, and a thread cannot allocate more than its share. An alternative approach is to dynamically partition resources based on application demands. Different programs exercise different resource demands, and in addition, resource demands may even vary over time. The idea behind dynamic resource partitioning is identify resource demands at runtime, and allocate resources accordingly, while preventing resource-hungry programs to monopolize a shared resource.

We now compare the MLP-aware flush policy against static resource partitioning and dynamic resource partitioning. Static resource partitioning provides an equal share of the buffer resources (ROB, load/store queue, issue queues, and physical register files) to each thread, while sharing the functional units among the threads. The dynamic resource partitioning approach that we

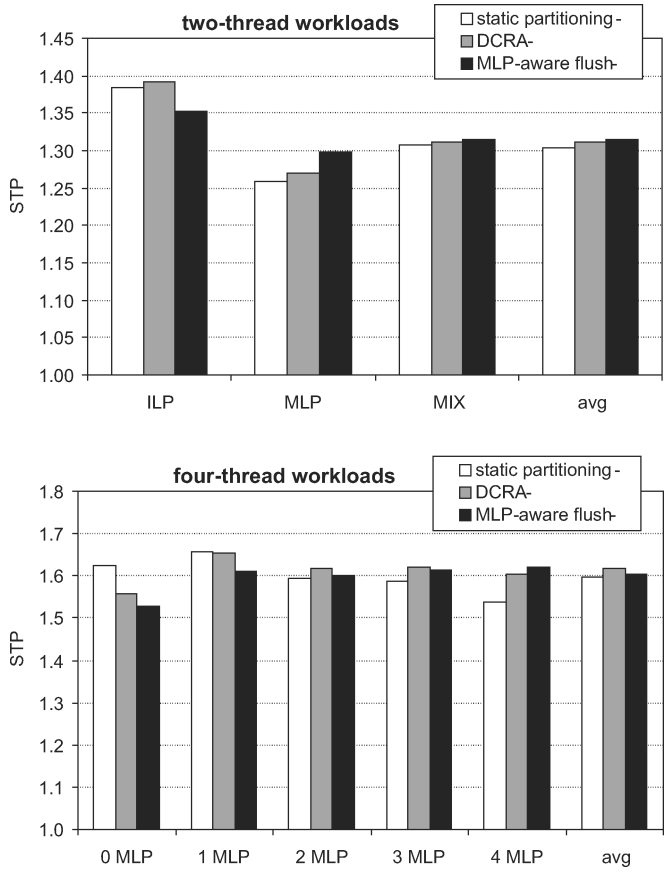

Fig. 22. Evaluating MLP-aware flush against static partitioning and dynamic particitioning (DCRA) in terms of STP for the two-thread workloads (top graph) and four-thread workloads (bottom graph).

compare against is dynamically controlled resource allocation (DCRA) proposed by Cazorla et al. [2004b], which manages the shared resources based on the occupancy counts in the issue queues, the number of allocated physical registers, and the number of L1 data cache misses. The idea of DCRA is to give more resources to memory-intensive threads for MLP exploitation. Figures 22 and 23 show STP and ANTT for the MLP-aware flush policy compared to static resource partitioning and dynamic resource partitioning (DCRA); results are shown for the two-thread and four-thread workloads. Although DCRA achieves a better STP (2.9%) and ANTT (3.3%) than MLP-aware flush for the ILP-intensive workloads, MLP-aware flush achieves a 5.4% better ANTT than DCRA for MLP-intensive and mixed ILP/MLP-intensive workloads for a comparable or slightly better STP (up to 2.1% for the MLP-intensive workloads). For MLP-intensive four-thread workload mixes, MLP-aware flush achieves a 8.5% better ANTT than DCRA. From this result, we conclude that DCRA is an effective

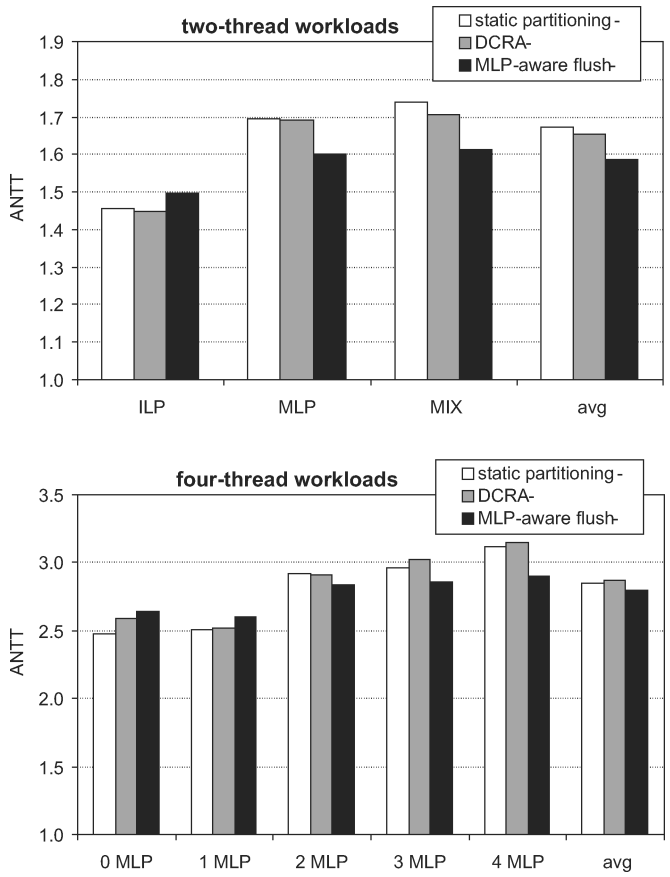

Fig. 23. Evaluating MLP-aware flush against static partitioning and dynamic particitioning (DCRA) in terms of ANTT for the two-thread workloads (top graph) and four-thread workloads (bottom graph).

approach for dynamically managing SMT processor resources; however, for memory-intensive workloads, MLP-aware flush is more effective than DCRA leading to shorter job turnaround times. The reason why MLP-aware flush outperforms DCRA is as follows: DCRA is oblivious to the amount of MLP that is, a thread is classified as a memory-intensive thread if at least one L1 cache miss is outstanding, and allocates a fixed amount of resources for the memory-intensive thread. On the other hand, MLP-aware flush allocates just enough resources to exploit the available MLP, leaving the rest of the resources to the other thread(s).

7. RELATED WORK

There are four avenues of research related to this work: (i) MLP, (ii) SMT fetch policies and resource partitioning, (iii) coarse-grained multithreading, and (iv) prefetching.

7.1 Memory-Level Parallelism

Glew [1998] pointed out the performance impact of MLP and advocated microarchitectural modifications for increasing the number of outstanding cache misses during single-program execution.

Karkhanis and Smith [2004] propose an analytical superscalar processor model that provides fundamental insight into how MLP affects overall performance. An isolated long-latency load typically blocks the head of the ROB after which performance drops to zero; when the data returns, performance ramps up again to steady-state performance. The overall penalty for an isolated long-latency load is approximately the access time to the next level in the memory hierarchy. For independent long-latency loads that occur within W instructions from each other in the dynamic instruction stream with W being the ROB size, the penalties completely overlap in time.

Chou et al. [2004] study the impact of the microarchitecture on the amount of MLP. Therefore, they evaluate the effectiveness of various microarchitecture techniques such as out-of-order execution, value prediction [Zhou and Conte 2003; Tuck and Tullsen 2005], and runahead execution [Mutlu et al. 2003]. Mutlu et al. [2005] propose using MLP predictors to improve the efficiency of runahead processors by not going into runahead mode in case there is no MLP to be exploited. The MLP predictor proposed by Mutlu et al. [2005] is a load PC indexed table with a two-bit counter in each entry; a two-bit counter keeps track of the amount of MLP associated with a given long-latency load. If there is MLP to be exploited for a long-latency load blocking the head of the ROB, the processor will enter runahead mode; if no MLP is to be exploited, the processor does not enter runahead. The MLP predictor proposed by Mutlu et al. [2005] is a binary predictor that predicts whether there is MLP to be exploited, or whether it is valuable to enter runahead mode. The Mutlu et al. [2005] predictor does not predict the MLP distance though. The MLP distance predictor proposed in this article could improve on this scheme by determining whether or not runahead mode should be entered *and*, if runahead mode is entered, how long runahead mode should go on, based on the predicted MLP distance (runahead should stop after the last long-latency load in the long-latency load burst).

Qureshi et al. [2006] propose an MLP-aware cache replacement policy. They propose to augment traditional recency-based cache replacement policies with MLP information. The goal is to reduce the number of isolated cache misses and if needed, to interchange isolated cache misses for overlapping cache misses, thus exposing MLP and improving overall performance. The MLP-aware cache replacement policy proposed by Qureshi et al. [2006] requires that an MLP-based cost is computed per cache miss. Isolated cache misses get a large cost assigned; overlapping cache misses get a smaller cost assigned. This MLP-based cost is different from what we use in our MLP-aware SMT fetch policy. We need to know how far we need to go in the dynamic instruction stream in order to expose the maximum available MLP.

MLP can also be exposed through compiler optimizations. Read miss clustering for example is a compiler technique proposed by Pai and Adve [1999]

that transforms the code in order to increase the amount of MLP. Read miss clustering strives at scheduling independent, likely long-latency memory accesses as close to each other as possible. At execution time, these long-latency loads will then overlap improving overall performance.

7.2 SMT Fetch Policies and Resource Partitioning

An important design issue for SMT processors is how to partition the available resources such as the instruction issue buffer and ROB. One solution is to statically partition the available resources [Raasch and Reinhardt 2003]. The downside of static partitioning is the lack of flexibility. Static partitioning prevents resources from a thread that does not require all of its available resources to be used by another thread that may benefit from the additional resources.

Dynamic partitioning, on the other hand, allows multiple threads to share resources. In dynamic partitioning, the use of the common pool of resources is determined by the fetch policy. The fetch policy determines from what thread instructions need to be fetched in a given cycle. Several fetch policies have been proposed in the recent literature. ICOUNT [Tullsen et al. 1996] prioritizes threads with few instructions in the pipeline. The limitation of ICOUNT is that in case of a long-latency load, ICOUNT may continue allocating resources for a stalled thread; by consequence, these resources will be held by the stalled thread and will prevent the other thread from allocating these resources. In response to this problem, Tullsen and Brown [2001] proposed two schemes for handling long-latency loads, namely (i) fetch stall the thread executing the long-latency thread, and (ii) flush instructions fetched passed the long-latency load in order to deallocate resources. Cazorla et al. [2004a] improved on the work done by Tullsen and Brown [2001] by predicting long-latency loads along with the COT mechanism that prioritizes the oldest thread in case all threads wait for a long-latency load.

El-Moursy and Albonesi [2003] propose to give fewer resources to threads that experience many data cache misses in order to minimize issue queue occupancy for saving energy. They propose two schemes for doing so, namely data miss gating and predictive data miss gating (PDG). Data miss gating drives the fetching based on the number of observed L1 data cache misses (i.e., by counting the number of L1 data cache misses in the execute stage of the pipeline). When the number of L1 data cache misses exceeds a given threshold, the thread is fetch gated. Predictive data miss gating strives at overcoming the delay between observing the L1 data cache miss and the actual fetch gating in the data miss gating scheme by predicting L1 data cache misses in the front-end pipeline stages.

A number of other fetch policies have been proposed driven by explicit resource partitioning. For example, Cazorla et al. [2004b] propose DCRA, which monitors the dynamic usage of resources by each thread and strives at giving a higher share of the available resources to memory-intensive threads. The input to their scheme consists of various usage counters for the number of instructions in the instruction queues, the number of allocated physical registers and the number of L1 data cache misses. Using these counters, they dynamically

determine the amount of resources required by each thread and prevent threads from using more resources than they are entitled to use. As mentioned earlier, DCRA drives the resource partitioning mechanism using imprecise MLP information and allocates a fixed amount of additional resources for memory-intensive workloads irrespective of the amount of MLP; the MLP-aware fetch policies proposed in this article, on the other hand, predict the amount of resources needed for exploiting the available MLP. Choi and Yeung [2006] go one step further than DCRA, and use a learning-based resource partitioning policy: They introduce a feedback loop that monitors the impact that resource partitioning decisions have on overall performance and feed that back into the resource partitioning mechanism for driving future decisions. Because of the performance feedback loop, learning-based resource partitioning is less responsive to dynamic workload behavior than an MLP-aware fetch policy. An interesting avenue for future work may be to make these explicit resource partitioning mechanisms MLP-aware.

Subsequent to our earlier work [Eyerman and Eeckhout 2007] and parallel with this article, Ramirez et al. [2008] proposed runahead threads for mitigating the performance impact of long-latency loads. The idea of a runahead thread [Mutlu et al. 2003] is to not block commit on a long-latency load but to speculatively execute instructions ahead in order to expose MLP through prefetching. Runahead threads have the important benefit not to clog any resources. Ramirez et al. [2008] report significant performance improvements using runahead threads in an SMT processor. We believe that the insights obtained in this article can help improve the effectiveness of runahead threads. The MLP predictor can be used to predict whether or not to go in runahead mode. If the predicted MLP distance is small, it may be beneficial to apply MLP-aware flush and not to go in runahead mode; only in case the predicted MLP distance is large, runahead execution should be initiated. Studying SMT architectures, which combine MLP-aware flush with runahead threads, is part of our future work.

7.3 Coarse-Grained Multithreading

Coarse-grained multithreading (CGMT) [Agarwal et al. 1993; Thekkath and Eggers 1994] is a form of multithreaded execution that executes one thread at a time but can switch relatively quickly (on the order of tens of cycles) to another thread. This makes CGMT suitable for hiding long-latency loads that is, a context switch is performed when a long-latency load is observed. Tune et al. [2004] combined CGMT with SMT into balanced multithreading in order to combine the best of both worlds, that is, balanced multithreading combines the ability of SMT to hide short latencies versus the ability of CGMT to hide long latencies. Tune et al. [2004] provided the intuition that for some applications a context switch should not be performed as soon as the long-latency load is detected in order to exploit MLP. The insights obtained in this article provide a way to further develop this idea: A context switch should not be done for all long-latency loads, but should rather be performed at isolated long-latency loads and at the last long-latency load in a burst of long-latency loads that occur

within a ROB size from each other. The MLP predictor proposed in this article can be used to drive this mechanism.

7.4 Prefetching

Prefetching [Anderson et al. 1967; Collins et al. 2001; Luk 2001] is a technique that addresses the long-latency load problem by seeking to eliminate the latency itself by bringing the miss data ahead of time to the appropriate cache level. Prefetching is orthogonal to the MLP-aware fetch policy proposed in this article, that is, the MLP-aware fetch policy can be applied on the long-latency loads that are not hidden through prefetching. The results presented in this article confirm this: Our baseline processor configuration includes an aggressive hardware prefetcher, and we still achieve performance speedups through an MLP-aware fetch policy.

8. CONCLUSION

Long-latency loads are particularly challenging in an SMT processor because, in the absence of an adequate fetch policy, they cause the thread to allocate critical resources without making forward progress. This limits the achievable performance for the other thread(s). Previous work proposed a number of SMT fetch policies for addressing the long-latency load problem. The key notion lacking in these fetch policies, however, is MLP.

This article showed that being aware of the available MLP allows for improving SMT fetch policies. The key insight from this article is that in case of an isolated long-latency load, the stalling thread should indeed be fetch stalled or flushed, as proposed by previous work. However, in case multiple independent long-latency loads overlap—there is MLP—the fetch policy should not fetch stall or flush the thread. Instead, the fetch policy should continue fetching instructions up to the point where the maximum available MLP for the given ROB size can be achieved. Our experimental results showed that an MLP-aware fetch policy achieves substantially higher STP and substantially smaller turnaround times.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable feedback and suggestions.

REFERENCES

Agarwal, A., Kubiatowicz, J., Kranz, D., Lim, B.-H., Yeung, D., D'Souza, G., and Parkin, M. 1993. Sparcle: An evolutionary processor design for large-scale multiprocessors. *IEEE Micro 13*, 3, 48–61.

Anderson, D. W., Sparacio, F. J., and Tomasulo, R. M. 1967. The IBM system/360 model 91: machine philosophy and instruction-handling. *IBM J. Res. Dev. 11*, 1, 8–24.

Cazorla, F. J., Fernandez, E., Ramirez, A., and Valero, M. 2004a. Optimizing long-latency-load-aware fetch policies for SMT processors. *Int. J. High Perform. Comput. Network. 2*, 1, 45–54.

Cazorla, F. J., Ramirez, A., Valero, M., and Fernandez, E. 2004b. Dynamically controlled resource allocation in SMT processors. In *Proceedings of the 37th Annual IEEE/ACM*

International Symposium on Microarchitecture (MICRO-37). IEEE, Los Alamitos, CA, 171–182.

CHOI, S. AND YEUNG, D. 2006. Learning-based SMT processor resource distribution via hill-climbing. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA'06)*. ACM, New York 239–250.

CHOU, Y., FAHS, B., AND ABRAHAM, S. 2004. Microarchitecture optimizations for exploiting memory-level parallelism. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA'04)*. ACM, New York, 76–87.

COLLINS, J., WANG, H., TULLSEN, D., HUGHES, C., LEE, Y.-F., LAVERY, D., AND SHEN, J. P. 2001. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA'01)*. ACM, New York, 14–25.

CRISTAL, A., SANTANA, O. J., VALERO, M., AND MARTINEZ, J. F. 2004. Toward kilo-instruction processors. *ACM Trans. Archit. Code Opt. 1*, 4, 389–417.

EL-MOURSY, A. AND ALBONESI, D. H. 2003. Front-end policies for improved issue efficiency in SMT processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA'03)*. IEEE, Los Alamitos, CA, 31–40.

EYERMAN, S. AND EECKHOUT, L. 2007. A memory-level parallelism aware fetch policy for SMT processors. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'07)*. IEEE, Los Alamitos, CA, 240–249.

EYERMAN, S. AND EECKHOUT, L. 2008. System-level performance metrics for multi-program workloads. *IEEE Micro* 28, 3, 42–53.

GLEW, A. 1998. MLP yes! ILP no! In *Proceedings of the Architectural Support for Programming Languages and Operating Systems Wild and Crazy Idea Session*.

JOHN, L. K. 2006. Aggregating performance metrics over a benchmark suite. In *Performance Evaluation and Benchmarking*, L. K. John and L. Eeckhout Eds. CRC Press, Boca Raton, FL, 47–58.

KARKHANIS, T. AND SMITH, J. E. 2002. A day in the life of a data cache miss. In *Proceedings of the 2nd Annual Workshop on Memory Performance Issues (WMPI'02)*. ACM, New York.

KARKHANIS, T. S. AND SMITH, J. E. 2004. A first-order superscalar processor model. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA'04)*. ACM, New York, 338–349.

KESSLER, R. E., MCLELLAN, E. J., AND WEBB, D. A. 1998. The Alpha 21264 microprocessor architecture. In *Proceedings of the 1998 International Conference on Computer Design (ICCD'98)*. IEEE, Los Alamitos, CA, 90–95.

LIMOUSIN, C., SBOT, J., VARTANIAN, A., AND DRACH-TEMAM, N. 2001. Improving 3D geometry transformation on a simultaneous multithreaded SIMD proceesor. In *Proceedings of the 13th International Conference on Supercomputing (ICS'01)*. ACM, New York, 236–245.

LUK, C.-K. 2001. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA'01)*. ACM, New York, 40–51.

LUO, K., GUMMARAJU, J., AND FRANKLIN, M. 2001. Balancing throughput and fairness in SMT processors. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'01)*. IEEE, Los Alamitos, CA, 164–171.

MUTLU, O., KIM, H., AND PATT, Y. N. 2005. Techniques for efficient processing in runahead execution engines. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA'05)*. ACM, New York, 370–381.

MUTLU, O., STARK, J., WILKERSON, C., AND PATT, Y. N. 2003. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA'03)*. IEEE, Los Alamitos, CA, 129–140.

PAI, V. S. AND ADVE, S. V. 1999. Code transformations to improve memory parallelism. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-32)*. IEEE, Los Alamitos, CA, 147–155.

Perelman, E., Hamerly, G., and Calder, B. 2003. Picking statistically valid and early simulation points. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT'03)*. IEEE, Los Alamitos, CA, 244–256.

QURESHI, M. K., LYNCH, D. N., MUTLU, O., AND PATT, Y. N. 2006. A case for MLP-aware cache replacement. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA'06)*. ACM, New York, 167–177.

RAASCH, S. E. AND REINHARDT, S. K. 2003. The impact of resource partitioning on SMT processors. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT'03)*. IEEE, Los Alamitos, CA, 15–26.

RAMIREZ, T., PAJUELO, A., SANTANA, O. J., AND VALERO, M. 2008. Runahead threads to improve SMT performance. In *Proceedings of the 14th International Symposium on High-Performance Computer Architecture (HPCA'08)*. IEEE, Los Alamitos, CA, 149–158.

SHERWOOD, T., PERELMAN, E., HAMERLY, G., AND CALDER, B. 2002. Automatically characterizing large scale program behavior. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'02)*. ACM, New York, 45–57.

SHERWOOD, T., SAIR, S., AND CALDER, B. 2000. Predictor-directed stream buffers. In *Proceedings of the 33rd Annual International Symposium on Microarchitecture (MICRO-33)*. IEEE, Los Alamitos, CA, 42–53.

SNAVELY, A. AND TULLSEN, D. M. 2000. Symbiotic jobscheduling for simultaneous multithreading processor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'00)*. ACM, New York, 234–244.

SRINIVASAN, S. T., RAJWAR, R., AKKARY, H., GANDHI, A., AND UPTON, M. 2004. Continual flow pipelines. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'04)*. ACM, New York, 107–119.

THEKKATH, R. AND EGGERS, S. J. 1994. *The effectiveness of multiple hardware contexts. In Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'94)*. ACM, New York, 328–337.

TUCK, N. AND TULLSEN, D. 2005. Multithreaded value prediction. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA'05)*. IEEE, Los Alamitos, CA, 5–15.

TUCK, N. AND TULLSEN, D. M. 2003. Initial observations of the simultaneous multithreading Pentium 4 processor. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'03)*. IEEE, Los Alamitos, CA, 26–34.

TULLSEN, D. 1996. Simulation and modeling of a simultaneous multithreading processor. In *Proceedings of the 22nd Annual Computer Measurement Group Conference*. Curran Associates, Red Hook, NY.

TULLSEN, D. M. AND BROWN, J. A. 2001. Handling long-latency loads in a simultaneous multithreading processor. In *Proceedings of the 34th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-34)*. IEEE, Los Alamitos, CA, 318–327.

TULLSEN, D. M., EGGERS, S. J., EMER, J. S., LEVY, H. M., LO, J. L., AND STAMM, R. L. 1996. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA'96)*. ACM, New York, 191–202.

TULLSEN, D. M., EGGERS, S. J., AND LEVY, H. M. 1995. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95)*. ACM, New York, 392–403.

TUNE, E., KUMAR, R., TULLSEN, D., AND CALDER, B. 2004. Balanced multithreading: Increasing throughput via a low cost multithreading hierarchy. In *Proceedings of the 37th Annual International Symposium on Microarchitecture (MICRO-37)*. IEEE, Los Alamitos, CA, 183–194.

ZHOU, H. AND CONTE, T. M. 2003. Enhancing memory level parallelism via recovery-free value prediction. In *Proceedings of the 17th Annual International Conference on Supercomputing (ICS)*. ACM, New York, 326–335.

Received June 2007; revised May 2008, September 2008; accepted September 2008