

A Memory-Level Parallelism Aware Fetch Policy for SMT Processors

Stijn Eyerman Lieven Eeckhout

ELIS Department, Ghent University, Belgium
Email: {seyerman, leeckhou}@elis.UGent.be

Abstract

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 given thread from fetching more instructions — and in some implementations, instructions beyond the long-latency load may even be flushed which frees allocated resources.

This paper proposes an SMT fetch policy that takes into account the available memory-level parallelism (MLP) in a thread. The key idea proposed in this paper 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. The proposed MLP-aware fetch policy achieves better performance for MLP-intensive threads on an SMT processor and achieves a better overall balance between performance and fairness than previously proposed fetch policies.

1 Introduction

A thread experiencing a long-latency load (a last cache level miss or D-TLB miss) in a simultaneous multithreading processor [22, 25, 26] will stall while holding execution resources without making progress. This affects the performance of the co-scheduled thread(s) because the co-scheduled thread(s) cannot make use of the resources allocated by the stalled thread.

Tullsen and Brown [24] and Cazorla *et al.* [2] 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 in [2, 24], instructions may even be flushed in order to free execution resources allocated by the stalled

thread such as reorder buffer space, instruction queue entries, *etc.*, in favor of the non-stalled threads. A limitation of these long-latency aware fetch policies is that they do not preserve the memory-level parallelism (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 paper proposes a fetch policy for SMT processors that takes into account memory-level parallelism for determining when to fetch stall or flush a thread executing a long-latency load. More in particular, we predict the amount of MLP for a given load and based on the predicted amount of MLP, 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 of MLP, we only allocate as many resources as required to expose the available memory-level parallelism. The end result is that in the no-MLP case, the other thread(s) can allocate all the available resources improving 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 [2, 24]. Our experimental results using SPEC CPU2000 show that the MLP-aware fetch policy achieves an average 34.6% better performance/fairness balance for MLP-intensive workloads compared to the previously proposed load-latency aware fetch policies [2, 24] and a 28.6% better performance/fairness balance compared to ICOUNT [25]. For mixed ILP/MLP-intensive workloads, our MLP-aware fetch policy achieves an average 8.5% and 21% better performance/fairness balance compared to load-latency aware fetch policies and ICOUNT, respectively.

This paper is organized as follows. We first revisit memory-level parallelism and quantify the amount of memory-level parallelism 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 sec-

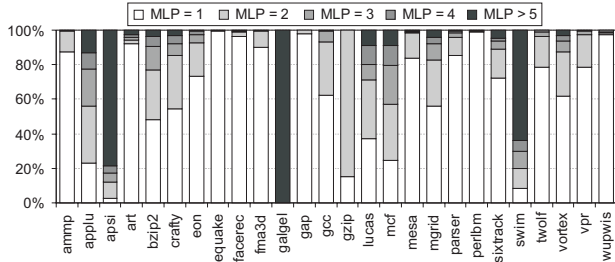


Figure 1. Amount of MLP for all of the benchmarks.

tion 6. Before concluding in section 8 we also detail on 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 of long-latency loads is typically quite large — on the order of a hundred or more cycles. (Note that the term long-latency load collectively refers 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 [9]. When the miss data returns from memory, instruction issuing resumes.

In contemporary superscalar processors though, multiple long-latency loads can be outstanding at a given point in time. This is made possible through various microarchitecture techniques such as non-blocking caches, miss status handling registers (MSHRs), *etc.* 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 [9, 10]. The reason is that as the first long-latency load blocks the ROB head, the ROB will 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 MSHRs and associated structures available. In other words, in case multiple independent long-latency loads occur within W instructions from each other, with W being the size of the reorder buffer (ROB), their penalties will overlap [5, 9]. This is called *memory-level parallelism (MLP)* [8]; the latency from one long-latency load is hidden by the latency from another long-latency load.

Following this reasoning, we define the amount of memory-level parallelism (MLP) in a particular program as the average number of long-latency loads outstanding

benchmark	input	LLL	MLP	MLP impact	type
bzip2	program	0.7	1.75	8.40%	MLP
crafty	ref	0.5	1.92	9.52%	MLP
eon	rushmeier	0.0	1.33	0.01%	ILP
gap	ref	0.2	1.02	0.15%	ILP
gcc	166	0.5	1.55	3.97%	ILP
gzip	graphic	0.1	1.84	1.31%	ILP
mcf	ref	64.1	2.81	54.68%	MLP
parser	ref	0.9	1.29	4.73%	ILP
perlbmk	makerand	0.3	1.01	0.07%	ILP
twolf	ref	1.9	1.26	6.99%	MLP
vortex	ref2	1.6	1.56	13.25%	MLP
vpr	route	2.5	1.26	8.92%	MLP
ammp	ref	18.2	1.14	7.11%	MLP
applu	ref	8.5	3.21	47.13%	MLP
apsi	ref	12.1	8.95	69.99%	MLP
art	ref-110	1.1	1.09	2.27%	ILP
equake	ref	0.0	1	0.00%	ILP
facerec	ref	1.1	1.04	0.93%	ILP
fma3d	ref	0.0	1.1	0.00%	ILP
galgel	ref	169.7	22.79	95.24%	MLP
lucas	ref	9.1	2.15	39.76%	MLP
mesa	ref	0.1	1.19	0.72%	ILP
mgrid	ref	5.9	1.72	25.06%	MLP
sixtrack	ref	0.1	1.34	0.55%	ILP
swim	ref	40.6	6.46	79.39%	MLP
wupwise	ref	0.5	1.05	0.46%	ILP

Table 1. 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.

when there is at least one long-latency load outstanding [5]. Figure 1 shows the amount of MLP in all of the SPEC CPU2000 benchmarks. Table 1 (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 *gap*, *perlbmk*, *equake* and *wupwise*. Other benchmarks exhibit a fair amount of MLP, see for example *applu*, *apsi*, *swim* and the most notable example being *galgel*. The amount of MLP for *galgel* is extremely high, namely 22.79. The reason for this high MLP is the very large number of independent loads that index very large arrays that span multiple pages which results in a very large number of D-TLB misses. Note that for most benchmarks the amount of MLP correlates well with the number of long-latency loads; the latter is shown in the third column in Table 1. This is to be expected in case the long-latency loads are independent of each other. There are some exceptions though, such as *mcf* and *ammp*, that show a large number of long-latency loads with a moderate

amount of MLP. The reason is that the long-latency loads are dependent on each other.

The second but last column in Table 1 shows the impact MLP has on overall performance. More in particular, it quantifies the amount of performance improvement due to MLP, *i.e.*, the reduction in overall execution time by enabling the parallel processing of independent long-latency loads. For several benchmarks, the amount of MLP has a substantial impact on overall performance with performance improvements ranging from 10% up to 95%. For `apsi` for example, MLP reduces the execution time by almost 70%; for `galgel`, MLP even reduces the execution time by 95%. Based on this observation we can classify the various benchmarks according to their memory behavior, see the rightmost column in Table 1. We classify a benchmark as an MLP-intensive benchmark in case the impact of the MLP on overall performance is larger than 5%. The other benchmarks are classified as ILP-intensive benchmarks. We will use this benchmark classification later in this paper 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 how cache behavior affects overall performance. First, co-scheduled 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 deteriorating the performance of the other co-scheduled threads. Second, memory-bound threads can hold critical execution resources while not making any progress because of the long-latency memory accesses. More in particular, a long-latency load cannot be committed as long as the miss is not resolved. In the meanwhile though, 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 paper deals with the latter problem of long-latency threads holding execution resources.

The ICOUNT fetch policy [25], which fetches instructions from the thread(s) least represented in the pipeline, 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 approximate equal number of instructions in the pipeline. 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 [24] recognize the problem of long-latency loads and therefore propose two mechanisms to free the allocated resources by the stalled thread. In the first approach, they prevent the thread executing a long-

latency load to fetch any new instructions until the miss is resolved. The second mechanism 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.* [2] 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 memory-level parallelism. 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 memory-level parallelism 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 the next section, 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 paper consists of three mechanisms. First, we need to identify long-latency loads, or alternatively, we need to predict whether a given load is likely to be long-latency. Second, once the long-latency load is identified or predicted, we need to determine the amount of MLP associated with this load. Third, we need to drive the fetch policy with the obtained MLP information. These three mechanisms will now be discussed in more detail in the following subsections.

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 in case the load is found to be an L3 miss or a D-TLB miss.

The second mechanism is to predict whether a load is going to be a long-latency load. The predictor is placed at the pipeline frontend and long-latency loads are predicted as they pass through the frontend pipeline. We use the miss pattern predictor proposed in [12]. The miss pattern predictor consists of a table indexed by the load PC; each table entry records (i) the number of load hits between the two most recent long-latency loads, and (ii) the number of

load hits since the last long-latency load. In case the latter matches the former, *i.e.*, 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 long-latency. The predictor table is updated when a load executes. The predictor used in our experiments is a 2K-entry table; 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 [7]. We concluded though that the miss pattern predictor outperforms the other predictors — this conclusion was also reached in [2]. Note that a load hit/miss predictor has been implemented in commercial processors as is the case in the Alpha 21264 microprocessor [11] 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 exhibiting memory-level parallelism. The MLP predictor that we propose consists of a table indexed by the load PC. Each entry in the table contains the number of instructions one needs to go down the dynamic instruction stream in order to observe the maximum MLP for the given reorder buffer size. We assume one MLP predictor per thread.

Updating the MLP predictor is done using a structure called the *long-latency shift register* (LLSR) which, in our implementation, has as many entries as there are reorder buffer entries divided by the number of threads. Upon committing an instruction from the reorder buffer, we shift the LLSR over one bit position from tail to head, and then insert one 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 reorder buffer size. The MLP predictor is updated by inserting the computed MLP distance in the predictor table entry pointed to by the load PC.

Note that this MLP predictor is a fairly simple last value predictor: the most recently observed MLP distance is stored in the predictor table. According to our experimental results, this predictor performs well for our purpose. As part of our future work, we plan to study more advanced MLP predictors though.

4.3 MLP-Aware Fetch Policy

We consider two mechanisms for driving the MLP-aware fetch policy, namely fetch stall and flush — these two mechanisms are similar to the ones proposed by [24] and [2], however, these previous approaches did not consider memory-level parallelism.

In the *fetch stall* approach, we first predict in the frontend 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, we flush the instructions after m instructions since the long-latency load. 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 [11] effectively support checkpointing at all instructions. If the microprocessor would only support checkpointing at branches, our flush mechanism could continue fetching instructions until the next branch after the m instructions.

Our fetch policies of fetch stall and flush implement the ‘continue the oldest thread’ (COT) mechanism proposed in [2]. COT means that in case multiple 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 our MLP-aware fetch policy resorts to the ICOUNT fetch policy in the absence of long-latency loads.

5 Experimental Setup

The processor model being simulated is the superscalar out-of-order SMT processor as shown in Table 2. We use the SMTSIM simulator [23] in all of our experiments. We use the SPEC CPU2000 benchmarks in this paper with reference inputs, see Table 1. These benchmarks are compiled for the Alpha ISA. For all of these benchmarks we fastforward the first 1B instructions and subsequently simulate the next 500M instructions in detail. We consider various ILP-intensive, ILP/MLP-intensive and MLP-intensive mixes of two-thread and four-thread workloads.

We use two metrics in our evaluation. Our first metric is the average weighted speedup compared to single-threaded execution:

$$speedup = \frac{\sum_{i=1}^n \frac{IPC_{MT,i}}{IPC_{ST,i}}}{n},$$

fetch width	8 instructions per cycle
pipeline depth	8 stages
branch misprediction penalty	6 cycles
branch predictor	2K-entry gshare
branch target buffer	256 entries, 4-way set associative
active list entries	256 per thread
functional units	6 int ALUs, 4 ld/st units and 3 FP units
instruction queues	64 entries in total (32 int and 32 fp)
rename registers	100 integer and 100 floating-point
L1 instruction cache	64KB, 2-way, 64-byte lines
L1 data cache	64KB, 2-way, 64-byte lines
unified L2 cache	512KB, 2-way, 64-byte lines
unified L3 cache	4MB, 2-way, 64-byte lines
cache hierarchy latencies	L2 (10), L3 (20), MEM (100)

Table 2. The SMT processor model assumed in the evaluation.

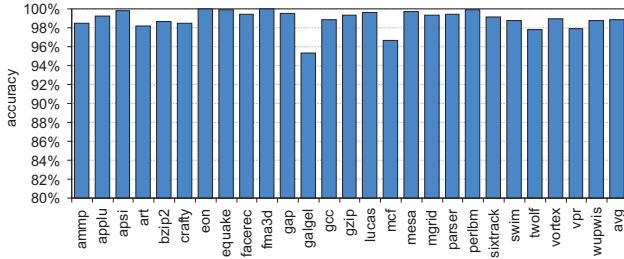


Figure 2. The accuracy of the long-latency load predictor.

with $IPC_{MT,i}$ and $IPC_{ST,i}$ being the IPC for thread i in multithreaded mode and single-threaded mode, respectively, and n being the number of threads. The pitfall with this metric is that it favors fetch policies that improve overall performance at the expense of degrading the performance of particular threads. Therefore, we also use a second metric called *hmean* which is the harmonic average speedup:

$$hmean = \frac{n}{\sum_{i=1}^n \frac{IPC_{ST,i}}{IPC_{MT,i}}}$$

This metric balances performance and fairness [14].

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.

6.1 Long-latency load predictor

The MLP-aware fetch stall approach requires that we can predict long-latency loads in the frontend stages of the processor pipeline. Figure 2 shows the prediction accuracy for

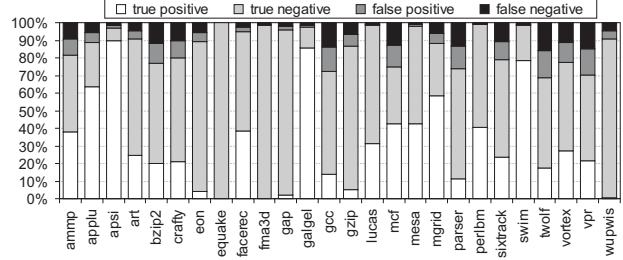


Figure 3. Evaluating the accuracy of the MLP predictor for predicting MLP.

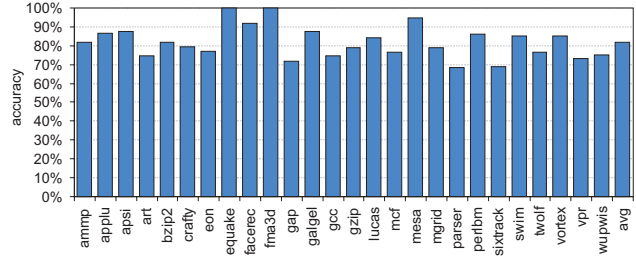


Figure 4. Evaluating the accuracy of the MLP predictor for predicting the MLP distance.

the long-latency load predictor. We observe that the accuracy achieved is very high, no less than 95% with an average prediction accuracy of 98.9%.

6.2 MLP predictor

Figure 3 evaluates the ability of the MLP predictor for predicting whether a long-latency load is going to expose MLP. A true positive means the MLP predictor predicts MLP in case there is MLP; a true negative means 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 average prediction accuracy equals 87.1%. The average fraction false negatives equals 6.5% 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, *i.e.*, the thread will be fetch stalled or flushed although there is MLP. The average fraction false positives equals 6.4% 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 4 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. The average MLP distance prediction accuracy equals 81.8%.

6.3 MLP-aware fetch policy

We now evaluate our MLP-aware fetch policy in terms of the speedup and hmean 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 pipeline. The following fetch policies extend upon the ICOUNT policy.
- The *stall fetch* approach proposed by Tullsen and Brown [24], *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 [2], 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 amount of MLP for predicted long-latency loads 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 [24] 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 in [24]) and flushes starting from the instruction following the long-latency load (this is the ‘next’ approach in [24]).
- The *MLP-aware flush* approach predicts the amount of MLP for a long-latency load, and fetch stalls or flushes the thread after m instructions since the long-latency load, with m the MLP distance predicted by the MLP predictor.

6.3.1 Two-thread workloads

Figures 5 and 6 show the speedup and hmean metrics for the various SMT fetch policies for the 2-thread workloads. There are three graphs in Figure 6, one for the ILP-intensive workloads, one for the MLP-intensive workloads and one for the mixed ILP/MLP-intensive workloads. There are several interesting observations to be made from these graphs. First, the flush policies generally outperform the fetch stall policies. This is in line with [24] and is explained by the fact that the flush policy is able to free resources allocated by a stalled thread. Second, the fetch policy does

not have a significant impact on performance and fairness for ILP-intensive workloads. Third, MLP-aware fetch policies improve the hmean metric for the MLP-intensive workloads by an average 28.6% improvement over ICOUNT and an average 34.6% improvement over flush; speedup is improved by 18.7% on average compared to ICOUNT and by 17.1% on average compared to flush. Fourth, for mixed ILP/MLP-intensive workloads, the MLP-aware flush policy improves speedup by 12.3% over ICOUNT on average and by 2.7% over flush on average. Likewise, the MLP-aware flush policy improves the hmean metric by 21.0% on average over ICOUNT and by 8.5% over flush. The bottom-line is that an MLP-aware fetch policy improves the performance of MLP-intensive threads. This is also illustrated in Figures 7 and 8 where IPC stacks are shown for MLP-intensive and mixed ILP/MLP-intensive workloads, respectively. These graphs show that the MLP-intensive thread typically achieves better performance under an MLP-aware fetch policy. The improved MLP-intensive thread performance leads to a better overall balance between performance and fairness.

6.3.2 Four-thread workloads

Figures 9 and 10 shows the speedup and hmean metrics for the various fetch policies for the 4-thread workloads. Here we obtain fairly similar results as for the 2-thread workloads. The MLP-aware fetch policies achieve better hmean numbers than non-MLP-aware fetch policies. Especially, the MLP-aware flush policy achieves a better balance between performance and fairness: the average hmean score for the MLP-aware flush policy is 9.0% better than for ICOUNT and 13.6% better than for flush.

7 Related Work

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

7.1 Memory-Level Parallelism

Karkhanis and Smith [9, 10] propose an analytical model that provides fundamental insight into how memory-level parallelism affects overall performance. An isolated long-latency load typically blocks the head of the reorder buffer 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 instructions with W being the reorder buffer size, the penalties completely overlap in time.

Chou *et al.* [5] study the impact of the microarchitecture on the amount of memory-level parallelism. Therefore

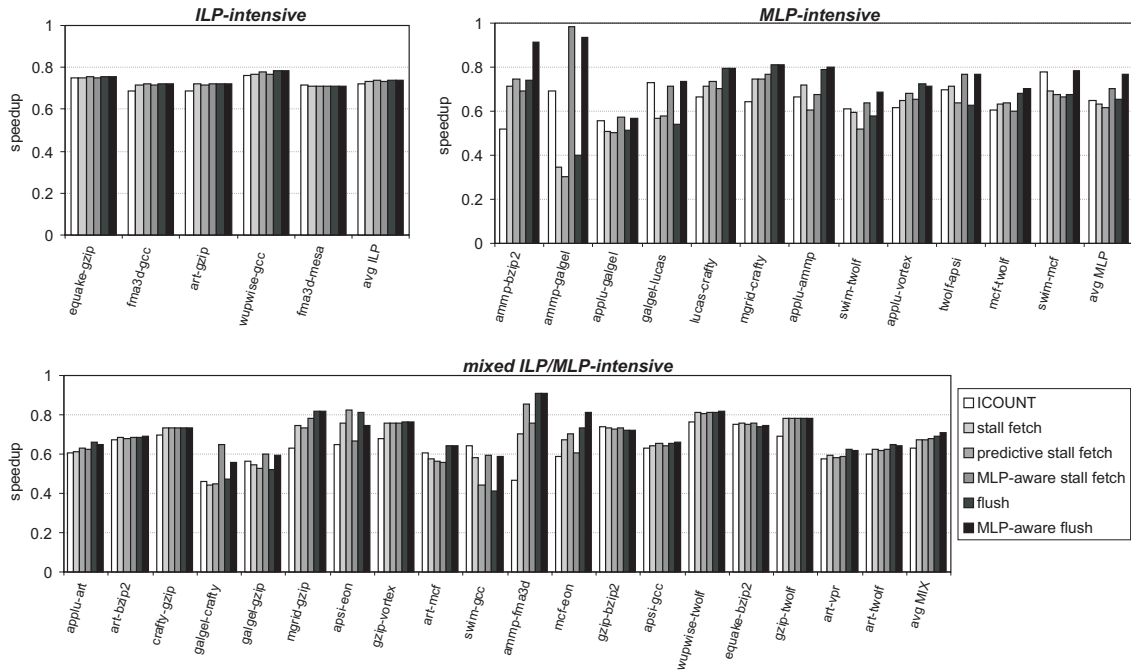


Figure 5. The speedup for the various SMT fetch policies compared to single-threaded execution for the 2-thread workloads.

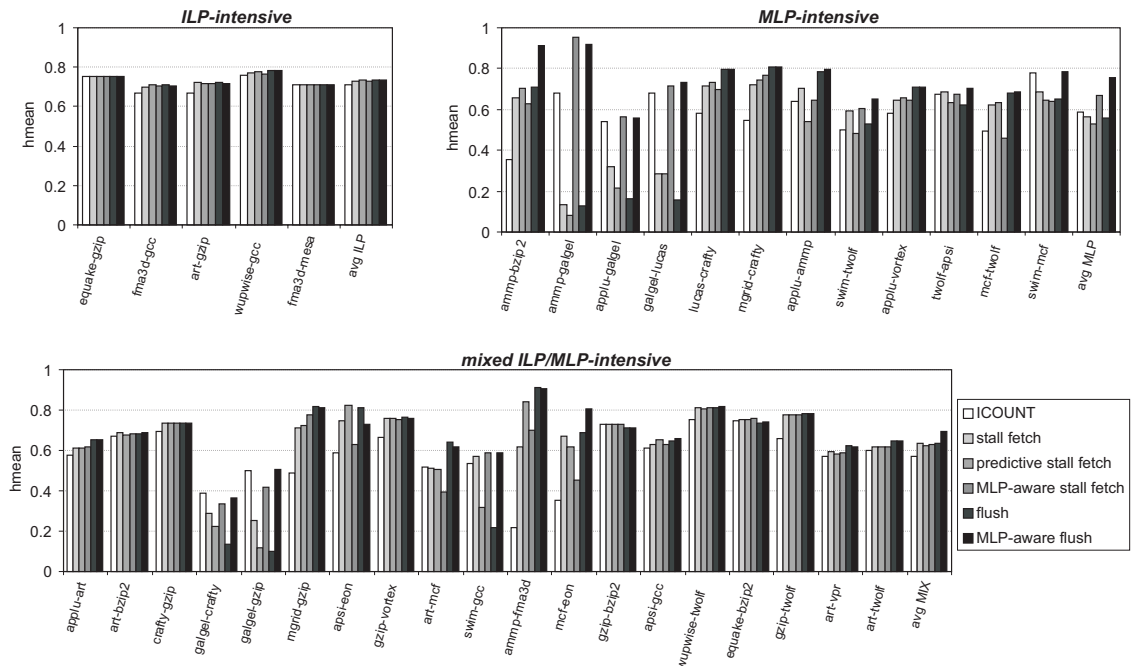


Figure 6. The hmean metric that balances performance and fairness for the various SMT fetch policies compared to single-threaded execution for the 2-thread workloads.

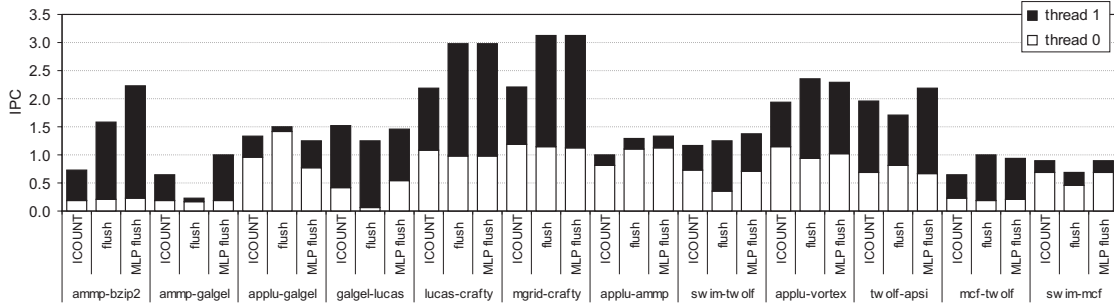


Figure 7. IPC values for the two threads ‘thread 0 – thread 1’ for the MLP-intensive workloads.

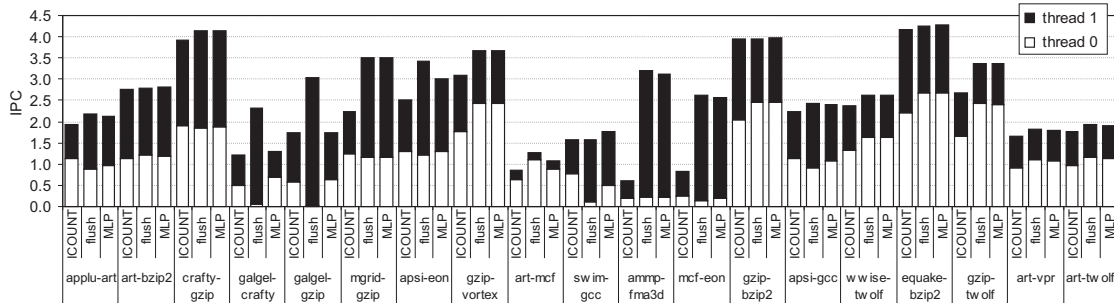


Figure 8. IPC values for the two threads ‘thread 0 – thread 1’ for the mixed ILP/MLP-intensive workloads.

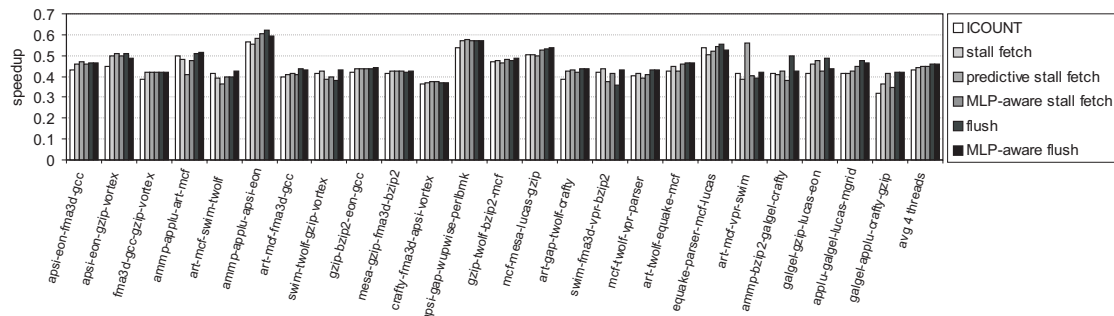


Figure 9. The speedup for the various SMT fetch policies compared to single-threaded execution for the 4-thread workloads.

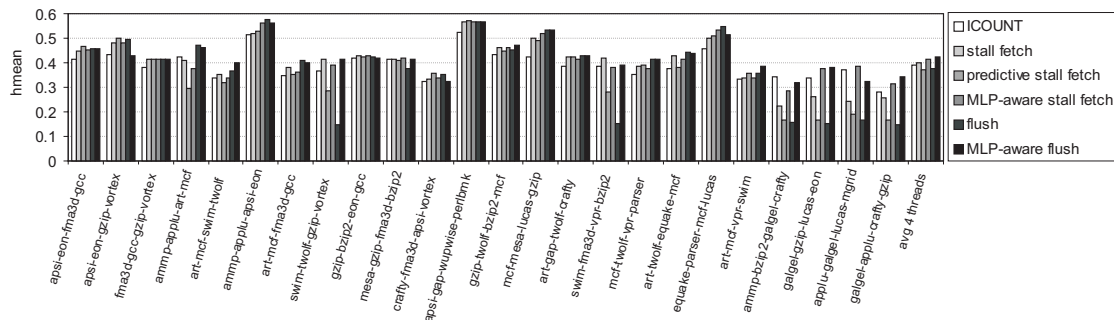


Figure 10. The hmean metric that balances performance and fairness for the various SMT fetch policies compared to single-threaded execution for the 4-thread workloads.

they evaluate the effectiveness of various microarchitecture techniques such as out-of-order execution, value prediction [21, 28], runahead execution [16] on the amount of MLP. Mutlu *et al.* [15] 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.

Qureshi *et al.* [18] 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.

MLP can also be exposed through compiler optimizations. Read miss clustering for example is a compiler technique proposed by Pai and Adve [17] that transforms the code in order to increase the amount of MLP. Read miss clustering strives at scheduling likely long-latency independent 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 reorder buffer. One solution is to statically partition [19] the available resources. 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 a 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 [25] 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 that cannot be allocated by the other thread(s). In response to this problem, Tullsen and Brown [24] proposed two basic 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.* [2] improved upon the work done by Tullsen and Brown by predicting long-latency loads along with the ‘continue the oldest thread (COT)’ mechanism that prioritizes the oldest thread in case multiple threads wait for a long-latency load.

Other fetch policies have been proposed as well, however these fetch policies do not address the long-latency load problem and are orthogonal to our MLP-aware fetch policy. For example, El-Moursy and Albonesi [7] propose to give fewer resources to threads that experience many data

cache misses. They propose two schemes for doing that, namely data miss gating (DG) and predictive data miss gating (PDG). DG 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. The more L1 data cache misses observed, the fewer resources the thread can allocate. PDG strives at overcoming the delay between observing the L1 data cache miss and the actual fetch gating in the DG scheme by predicting L1 data cache misses in the frontend pipeline stages. Another scheme by Cazorla *et al.* [3] proposes to monitor the dynamic usage of resources by each thread and strives at giving a fair share of the available resources to all the 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 observed L1 data cache misses. Choi and Yeung [4] go one step further and use a learning-based resource partitioning policy.

7.3 Coarse-Grained Multithreading

Coarse-Grained Multithreading (CGMT) [1, 20] 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, *i.e.*, a context switch is performed when a long-latency load is observed. Tune *et al.* [27] combined CGMT with SMT into Balanced Multithreading (BMT) in order to combine the best of both worlds, *i.e.*, the ability of SMT for hiding short latencies versus the ability of CGMT for hiding long latencies. Tune *et al.* 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 here in this paper provide a way to further develop this idea: a context switch should 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 reorder buffer size from each other. The MLP predictor proposed in this paper can be used to drive this mechanism.

7.4 Prefetching

Prefetching [6, 13] is a technique that addresses the long-latency load problem in a different way, namely 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 paper, *i.e.*, the MLP-aware fetch policy can be applied on the long-latency loads that are not adequately handled through prefetching.

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. However, none of this prior work fully addresses the long-latency load problem. The key notion lacking in these fetch policies is memory-level parallelism.

This paper showed that being aware of the available memory-level parallelism allows for improving SMT fetch policies. The key insight from this paper 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 better performance for MLP-intensive threads, and a better overall balance between performance and fairness.

Acknowledgements

The authors would like to thank the anonymous reviewers for their feedback. This research is supported by the Fund for Scientific Research–Flanders (Belgium) (FWO–Vlaanderen), Ghent University, the HiPEAC Network of Excellence and the European FP6 SARC project No. 27648.

References

- [1] A. Agarwal, J. Kubiawicz, D. Kranz, B.-H. Lim, D. Yeung, G. D'Souza, and M. Parkin. Sparcle: An evolutionary processor design for large-scale multiprocessors. *IEEE Micro*, 13(3):48–61, 1993.
- [2] F. J. Cazorla, E. Fernandez, A. Ramirez, and M. Valero. Optimizing long-latency-load-aware fetch policies for SMT processors. *IJHPCN*, 2(1):45–54, 2004.
- [3] F. J. Cazorla, A. Ramirez, M. Valero, and E. Fernandez. Dynamically controlled resource allocation in SMT processors. In *MICRO*, pages 171–182, Dec. 2004.
- [4] S. Choi and D. Yeung. Learning-based SMT processor resource distribution via hill-climbing. In *ISCA*, pages 239–250, June 2006.
- [5] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *ISCA*, pages 76–87, June 2004.
- [6] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *ISCA*, pages 14–25, July 2001.
- [7] A. El-Moursy and D. H. Albonesi. Front-end policies for improved issue efficiency in SMT processors. In *HPCA*, pages 31–40, Feb. 2003.
- [8] A. Glew. MLP yes! ILP no! In *ASPLOS Wild and Crazy Idea Session*, Oct. 1998.
- [9] T. Karkhanis and J. E. Smith. A day in the life of a data cache miss. In *WMPI held in conjunction with ISCA-29*, May 2002.
- [10] T. S. Karkhanis and J. E. Smith. A first-order superscalar processor model. In *ISCA*, pages 338–349, June 2004.
- [11] R. E. Kessler, E. J. McLellan, and D. A. Webb. The Alpha 21264 microprocessor architecture. In *ICCD*, pages 90–95, Oct. 1998.
- [12] C. Limousin, J. Sbot, A. Vartanian, and N. Drach-Temam. Improving 3D geometry transformation on a simultaneous multithreaded SIMD processor. In *ICS*, pages 236–245, June 2001.
- [13] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *ISCA*, pages 40–51, July 2001.
- [14] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. In *ISPASS*, pages 164–171, Nov. 2001.
- [15] O. Mutlu, H. Kim, and Y. N. Patt. Techniques for efficient processing in runahead execution engines. In *ISCA*, pages 370–381, June 2005.
- [16] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *HPCA*, pages 129–140, Feb. 2003.
- [17] V. S. Pai and S. V. Adve. Code transformations to improve memory parallelism. In *MICRO*, pages 147–155, Nov. 1999.
- [18] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A case for MLP-aware cache replacement. In *ISCA*, pages 167–177, June 2006.
- [19] S. E. Raasch and S. K. Reinhardt. The impact of resource partitioning on SMT processors. In *PACT*, pages 15–26, Sept. 2003.
- [20] R. Shekath and S. J. Eggers. The effectiveness of multiple hardware contexts. In *ASPLOS*, pages 328–337, Oct. 1994.
- [21] N. Tuck and D. Tullsen. Multithreaded value prediction. In *HPCA*, pages 5–15, Feb. 2005.
- [22] N. Tuck and D. M. Tullsen. Initial observations of the simultaneous multithreading Pentium 4 processor. In *PACT*, pages 26–34, Sept. 2003.
- [23] D. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *Proceedings of the 22nd Annual Computer Measurement Group Conference*, Dec. 1996.
- [24] D. M. Tullsen and J. A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *MICRO*, pages 318–327, Dec. 2001.
- [25] 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 *ISCA*, pages 191–202, May 1996.
- [26] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *ISCA*, pages 392–403, June 1995.
- [27] E. Tune, R. Kumar, D. Tullsen, and B. Calder. Balanced multithreading: Increasing throughput via a low cost multithreading hierarchy. In *MICRO*, pages 183–194, Dec. 2004.
- [28] H. Zhou and T. M. Conte. Enhancing memory level parallelism via recovery-free value prediction. In *ICS*, pages 326–335, June 2003.