MLP-Aware Runahead Threads in a Simultaneous Multithreading Processor

Kenzo Van Craeynest, Stijn Eyerman, and Lieven Eeckhout

Department of Electronics and Information Systems (ELIS), Ghent University, Belgium
kevcraey,seyerman,leeckhou@elis.UGent.be

Abstract. Threads experiencing long-latency loads on a simultaneous multithreading (SMT) processor may clog shared processor resources without making forward progress, thereby starving other threads and reducing overall system throughput. An elegant solution to the long-latency load problem in SMT processors is to employ runahead execution. Runahead threads do not block commit on a long-latency load but instead execute subsequent instructions in a speculative execution mode to expose memory-level parallelism (MLP) through prefetching. The key benefit of runahead SMT threads is twofold: (i) runahead threads do not clog resources on a long-latency load, and (ii) runahead threads exploit far-distance MLP.
This paper proposes MLP-aware runahead threads: runahead execution is only initiated in case there is far-distance MLP to be exploited. By doing so, useless runahead executions are eliminated, thereby reducing the number of speculatively executed instructions (and thus energy consumption) while preserving the performance of the runahead thread and potentially improving the performance of the co-executing thread(s). Our experimental results show that MLP-aware runahead threads reduce the number of speculatively executed instructions by 13.9% and 10.1% for two-program and four-program workloads, respectively, compared to MLP-agnostic runahead threads while achieving comparable system throughput and job turnaround time.

1 Introduction

Long-latency loads (last D-cache level misses and D-TLB misses) have a big performance impact on simultaneous multithreading (SMT) processors [23]. In particular, in an SMT processor with dynamically shared resources, a thread experiencing a long-latency load will eventually stall while holding resources (reorder buffer entries, issue queue slots, rename registers, etc.), thereby potentially starving the other thread(s) and reducing overall system throughput.

Tullsen and Brown [21] recognized this problem and proposed to limit the amount of resources allocated by threads that are stalled due to long-latency loads. In their *flush* policy, fetch is stalled as soon as a long-latency load is detected and instructions are flushed from the pipeline in order to free resources allocated by the long-latency thread. The flush policy by Tullsen and Brown, however, does not preserve memory-level parallelism (MLP) [3,8], but instead serializes independent long-latency loads. This may hurt the performance of memory-intensive (or, more precisely, MLP-intensive) threads.

Eyerman and Eeckhout [6] therefore proposed the *MLP-aware flush* policy which first predicts the MLP distance for a long-latency load, i.e., it predicts the number of instructions one needs to go down the dynamic instruction stream for exposing the available MLP. Subsequently, based on the predicted MLP distance, MLP-aware flush decides to (i) 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 flush a thread only in case there is no MLP; in case there is MLP, MLP-aware flush allocates as many resources as required to expose the available memory-level parallelism.

Ramirez et al. [17] proposed runahead threads in an SMT processor which avoid resource clogging on long-latency loads while exposing memory-level parallelism. The idea of runahead execution [14] 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 are particularly interesting in the context of an SMT processor because they solve two issues: (i) they do not clog resources on long-latency loads, and (ii) they preserve MLP, and even allow for exploiting far-distance MLP (beyond the scope of the reorder buffer).

A limitation of runahead threads in an SMT processor though is that they consume execution resources (functional unit slots, issue queue slots, reorder buffer entries, etc.) even if there is no MLP to be exploited, i.e., runahead execution does not contribute to the performance of the runahead thread in case there is no MLP to be exploited, and in addition, may hurt the performance of the co-executing thread(s) and thus overall system performance. In this paper, we propose *MLP-aware runahead threads*. The key idea of MLP-aware runahead threads is to enter runahead execution only in case there is far-distance MLP to be exploited. In particular, the MLP distance predictor first predicts the MLP distance upon a long-latency load, and in case the MLP distance is large, runahead execution is initiated. If not, i.e., in case the MLP distance is small, we fetch stall the thread after having fetched as many instructions as predicted by the MLP distance predictor, or we (partially) flush the long-latency thread if more instructions have been fetched than predicted by the MLP distance predictor.

MLP-aware runahead threads reduce the number of speculatively executed instructions significantly over MLP-agnostic runahead threads while not affecting overall SMT performance. Our experimental results using the SPEC CPU2000 benchmarks on a 4-wide superscalar SMT processor configuration report that MLP-aware runahead threads reduce the number of speculatively executed instructions by 13.9% and 10.1% on average for two-program and four-program workloads, respectively, compared to MLP-agnostic runahead threads, while yielding comparable system throughput and job turnaround time. Binary MLP prediction (using the previously proposed MLP predictor by Mutlu et al. [13]) along with an MLP-agnostic flush policy, further reduces the number of speculatively executed instructions under runahead execution by 13% but hurts system throughput (STP) by 11% and job turnaround time (ANTT) by 2.3% on average.

This paper is organized as follows. We first revisit the MLP-aware flush policy (Section 2) and runahead SMT threads (Section 3). Subsequently, we propose MLP-aware runahead threads in Section 4. After detailing our experimental setup in Section 5, we

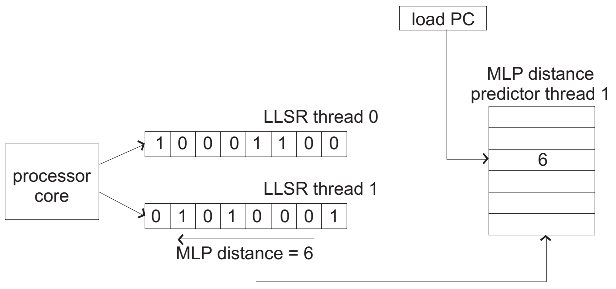

Fig. 1. Updating the MLP distance predictor.

then present our evaluation in Section 6. Finally, we describe related work (Section 7), and conclude (Section 8).

2 MLP-Aware Flush

The MLP-aware flush policy proposed in [6] consists of three mechanisms: (i) it identifies long-latency loads, (ii) it predicts the load's MLP distance, and (iii) it stalls fetch or flushes the long-latency thread based on the predicted MLP distance. The first step is trivial (i.e., a load instruction is labeled as a long-latency load as soon as the load is found out to be an off-chip memory access, e.g., an L3 miss or a D-TLB miss). We now discuss the second and third steps in more detail.

2.1 MLP Distance Prediction

Once a long-latency load is identified, the MLP distance predictor predicts the *MLP distance*, or the number of instructions one needs to go down the dynamic instruction stream in order to expose the maximum exploitable MLP for the given reorder buffer size. The MLP distance predictor consists of a table indexed by the load PC, and each entry in the table records the MLP distance for the corresponding load. There is one MLP distance predictor per thread.

Updating the MLP distance predictor is done using a structure called the *long-latency shift register* (LLSR), see Figure 1. The LLSR has as many entries as there are reorder buffer entries divided by the number of threads (assuming a shared reorder buffer), and there are as many LLSRs as there are threads. Upon committing an instruction from the reorder buffer, the LLSR is shifted over one bit position from tail to head, and one bit is inserted at the tail of the LLSR. A '1' is inserted in case the committed instruction is a long-latency load, and a '0' is inserted otherwise. 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 distance 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. In the example given in Figure 1, 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 is a simple last value predictor, i.e., the most recently observed MLP distance is stored in the predictor table.

2.2 MLP-Aware Fetch Policy

The best performing MLP-aware fetch policy reported in [6] is the MLP-aware flush policy and operates as follows. Say the predicted MLP distance equals *m*. Then, 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, and we then fetch stall the thread.

The flush mechanism requires checkpointing support by the microarchitecture. Commercial processors such as the Alpha 21264 [11] effectively support checkpointing at all instructions. If the microprocessor would only support checkpointing at branches for example, the flush mechanism could flush the instructions past the first branch after the next *m* instructions. The MLP-aware flush policy resorts to the ICOUNT fetch policy [22] in the absence of long-latency loads. The MLP-aware flush policy also implements the 'continue the oldest thread' (COT) mechanism proposed by Cazorla et al. [1]. 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 is likely to be the first thread to get the data back from memory and continue execution.

3 Runahead Threads

Runahead execution [4,14] avoids the processor from stalling when a long-latency load hits the head of the reorder buffer. When a long-latency load that is still being serviced, reaches the reorder buffer head, the processor takes a checkpoint (which includes the architectural register state, the branch history register and the return address stack), records the program counter of the blocking long-latency load, and initiates runahead execution. The processor then continues to execute instructions in a speculative way past the long-latency load: these instructions do not change the architectural state. Long-latency loads executed during runahead send their requests to main memory but their results are identified as invalid; and an instruction that uses an invalid argument also produces an invalid result. Some of the instructions executed during runahead execution (those that are independent of the long-latency loads) may miss in the cache as well. Their latencies then overlap with the long-latency load that initiated runahead execution. And this is where the performance benefit of runahead comes from: it exploits memory-level parallelism (MLP) [3,8], i.e., independent memory accesses are processed in parallel. When, eventually, the initial long-latency load returns from memory, the processor exits runahead execution, flushes the pipeline, restores the checkpoint, and resumes normal execution starting with the load instruction that initiated runahead execution. This normal execution will make faster progress because some of the data has already been prefetched in the caches during runahead execution.

Whereas Mutlu et al. [14] proposed runahead execution for achieving high performance on single-threaded superscalar processors, Ramirez et al. [17] integrate runahead threads in an SMT processor. The reason for doing so is twofold. First, runahead threads seek for exploiting MLP thereby improving per-thread performance. Second, runahead threads do not stall on commit and thus do not clog resources in an SMT processor. This appealing solution to the shared resource partitioning problem in SMT processors yields substantial SMT performance improvements, especially for memory-intensive workloads according to Ramirez et al. (and we confirm those results in our evaluation). The runahead threads proposal by Ramirez et al., however, initiates runahead execution upon a long-latency load irrespective of whether there is MLP to be exploited. As a result, in case there is no MLP, runahead execution will consume resources without contributing to performance, i.e., the runahead execution is useless because it does not exploit MLP. This is the problem being addressed in this paper and for which we propose MLP-aware threads as described in the next section.

4 MLP-Aware Runahead Threads

An MLP-aware fetch policy as well as runahead threads come with their own benefits and limitations. The limitation of an MLP-aware fetch policy is that it cannot exploit MLP over large distances, i.e., the exploitable MLP is limited to (a fraction of) the reorder buffer size. Runahead threads on the other hand can exploit MLP at large distances, beyond the scope of the reorder buffer, which improves performance substantially for memory-intensive workloads. However, if MLP-agnostic — as in the original description of runahead execution by Mutlu et al. [14] as well as in the follow-on work by Ramirez et al. [17] — runahead execution is initiated upon every in-service long-latency load that hits the reorder buffer head irrespective of whether there is MLP to be exploited. As a result, runahead threads may consume execution resources without any performance benefit for the runahead thread. Moreover, runahead execution may even hurt the performance of the co-executing thread(s). Another disadvantage of runahead execution compared to the MLP-aware flush policy is that more instructions need to be re-fetched and re-executed upon the return of the initiating long-latency load. In the MLP-aware flush policy on the other hand, instructions reside in the reorder buffer and issue queues and need not be re-fetched, and, in addition, the instructions that are independent of the blocking long-latency load need not be re-executed, potentially saving execution resources and energy consumption.

To combine the best of both worlds, we propose *MLP-aware runahead threads* in this paper. We distinguish two approaches to MLP-aware runahead threads.

Runahead threads based on binary MLP prediction. The first approach is to employ binary MLP prediction. We therefore use the MLP predictor proposed by Mutlu et al. [13] which was originally developed for limiting the number of useless runahead periods, thereby reducing the number of speculatively executed instructions under runahead execution in order to save energy. The idea of employing the MLP predictor is to enter runahead mode only in case the MLP predictor predicts there is far-distance MLP to be exploited.

The MLP predictor by Mutlu et al. is a load-PC indexed table with a two-bit saturating counter per table entry. Runahead mode is entered only in case the counter is in the '10' or '11' states. A long-latency load which has no counter associated with it, allocates a counter and resets the counter (to the state '00'). Runahead execution is not entered in the '00' and '01' states; instead, the counter is incremented. During runahead execution, the processor keeps track of the number of long-latency loads generated. (Mutlu et al. count the number of loads generated beyond the reorder buffer; in the SMT context with a shared reorder buffer, this translates to the reorder buffer size divided by the number of hardware threads.) When exiting runahead mode, if at least one long-latency load was generated during runahead mode, the associated counter is incremented; if not, the counter is decremented if in the '11' state, and is reset if in the '10' state.

Runahead threads based on MLP distance prediction. The second approach to MLP-aware runahead threads is to predict the MLP distance rather than to rely on a binary MLP prediction. We first predict the MLP distance upon a long-latency load. In case the predicted MLP distance is smaller than half the reorder buffer size for a two-thread SMT processor and one fourth the reorder buffer size for a four-thread SMT processor (i.e., this is what the MLP-aware flush policy can exploit), we apply the MLP-aware flush policy. In case the predicted MLP distance is larger than half (or one fourth) the reorder buffer size, we enter runahead mode. In other words, if there is no MLP or if there is exploitable MLP over a short distance only, we reside to the MLP-aware flush policy; if there is large-distance MLP to be exploited, we initiate runahead execution.

5 Experimental Setup

5.1 Benchmarks and Simulator

We use the SPEC CPU2000 benchmarks in this paper with their reference inputs. 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 [15,18]. We use a wide variety of randomly selected two-thread and four-thread workloads. The two-thread and four-thread workloads are classified as ILP-intensive, MLP-intensive or mixed ILP/MLP-intensive workloads.

We use the SMTSIM simulator v1.0 [20] in all of our experiments. The processor model being simulated is the 4-wide superscalar out-of-order SMT processor shown in Table 1. The default fetch policy is ICOUNT 2.4 [22] 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 reorder buffer upon commit and wait in 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.

5.2 Performance Metrics

We use two system-level performance metrics in our evaluation: system throughput (STP) and average normalized turnaround time (ANTT) [7]. System throughput (STP)

parameter	value
fetch policy	ICOUNT 2.4
pipeline depth	14 stages
(shared) reorder buffer size	128 entries
(shared) load/store queue	64 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, 4-way, 64-byte lines
L1 data cache	64KB, 4-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 (500)

Table 1. The baseline SMT processor configuration.

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 multi-threaded execution, respectively; there are n threads running simultaneously. STP is a higher-is-better metric and equals the weighted speedup metric proposed by Snavely and Tullsen [19].

Average normalized turnaround time (ANNT) 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 in [12], and is a lower-is-better metric. Eyerman and Eeckhout [7] argue 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 multi-program workload, simulation stops when 400 million instructions have been executed. At that point, program i will have executed x_i million instructions. The single-threaded CPI_i^{ST} used in the above formulas equals single-threaded CPI after x_i million instructions. When we report average STP and ANTT numbers across a number of multi-program workloads, we use the harmonic and arith-

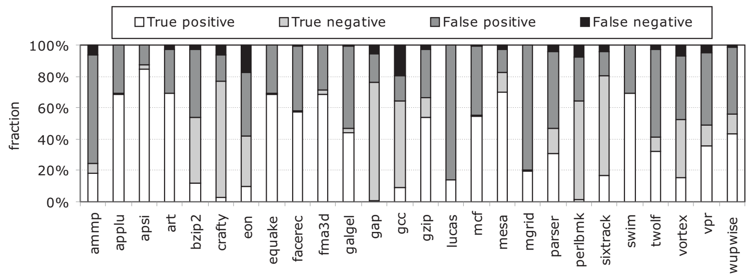

Fig. 2. Quantifying the accuracy of the MLP distance predictor.

metic mean for computing the average STP and ANTT, respectively, following the recommendations on the use of averages by John [10].

5.3 Hardware Cost

The performance numbers reported in the evaluation section assume the following hardware costs. For both the binary MLP predictor and the MLP distance predictor we assume a PC-indexed 2K-entry table. (We experimented with a number of predictor configurations, including the tagged set-associative table organization proposed by Mutlu et al. [13] and we found the untagged 2K-entry to slightly outperform the tagged organization by Mutlu et al.) An entry in the binary MLP predictor is a 2-bit field following Mutlu et al. [13]. An entry in the MLP distance predictor is a 3-bit field; one bit encodes whether long-distance MLP is to be predicted, and the other two bits encode the MLP distance within the reorder buffer in buckets of 16 instructions. The hardware cost for a run-length encoded LLSR equals 0.7Kbits in total: 32 (maximum number of outstanding long-latency loads) times 22 bits (11 bits for keeping track of the load PC index in the 2K-entry MLP distance predictor, plus 11 bits for the encoded run length — maximum of 2048 instructions — since the prior long-latency load miss). In summary, the total hardware cost for the binary MLP predictor equals 4Kbits; the total hardware cost for the MLP distance predictor (predictor table plus LLSR) equals 6.7Kbits.

6 Evaluation

6.1 MLP distance predictor

Key to the success of MLP-aware runahead threads is the accuracy of the MLP distance predictor. The primary concern is whether the predictor can accurately estimate far-distance MLP in order to decide whether or not to go in runahead mode.

Figure 2 shows the accuracy of the MLP distance predictor. A true positive denotes correctly predicted long-distance MLP and a true negative denotes correctly predicted short-distance or no MLP; the false positives and false negatives denote mispredictions. The prediction accuracy equals 61% on average, and the majority of mispredictions are false positives. In spite of this relatively low prediction accuracy, MLP-aware runahead threads are effective as will be demonstrated in the next few paragraphs. Improving MLP distance prediction will likely lead to improved effectiveness of MLP-aware runahead threads, i.e., reducing the number of false positives will reduce the number of speculatively executed instructions and will thus increase energy saving opportunities — this is left for future work though.

6.2 Two-program workloads

We compare the following SMT fetch policies and architectures:

– ICOUNT [22] 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 *MLP-aware flush* approach [6] 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.
– *Runahead threads*: threads go in runahead mode when the oldest instruction in the reorder buffer is a long-latency load that is still being serviced [17].
– *Binary MLP-aware runahead threads w/ ICOUNT*: the binary MLP predictor by Mutlu et al. [13] predicts whether there is far-distance MLP to be exploited, and a thread only goes in runahead mode in case MLP is predicted. In case there is no (predicted) MLP, we resort to ICOUNT.
– *Binary MLP-aware runahead threads w/ flush*: this is the same policy as the one above, except that in case of no (predicted) MLP, we perform a flush. The trade-off between this policy and the latter is that ICOUNT may exploit short-distance MLP whereas flush does not, however, flush prevents resource clogging.
– *MLP-distance-aware runahead threads*: the MLP distance predictor by Eyerman and Eeckhout [6] predicts the MLP distance. If there is far-distance MLP to be exploited, the thread goes in runahead mode. If there is only short-distance MLP to be exploited, the thread is fetch stalled and/or flushed according to the predicted MLP distance.

Figures 3 and 4 compare these six fetch policies in terms of the STP and ANTT performance metrics, respectively, for the two-program workloads. These results confirm the results presented in prior work by Ramirez et al. [17]: runahead threads improve both system throughput and job turnaround time significantly over both ICOUNT and MLP-aware flush: STP and ANTT improve by 70.1% and 43.8%, respectively, compared to ICOUNT; and STP and ANTT improve by 44.3% and 26.8%, respectively, compared to MLP-aware flush. These results also show that MLP-aware runahead threads (rightmost bars) achieve comparable performance as MLP-agnostic runahead threads. Moreover, MLP-aware runahead threads achieve a slight improvement in both STP and ANTT for some workloads over MLP-agnostic runahead threads, e.g., mesa-galgel achieves a 3.3% higher STP and a 3.2% smaller ANTT under MLP-aware runahead threads compared to MLP-agnostic runahead threads. The reason for this performance improvement is that preventing one thread from entering runahead mode gives

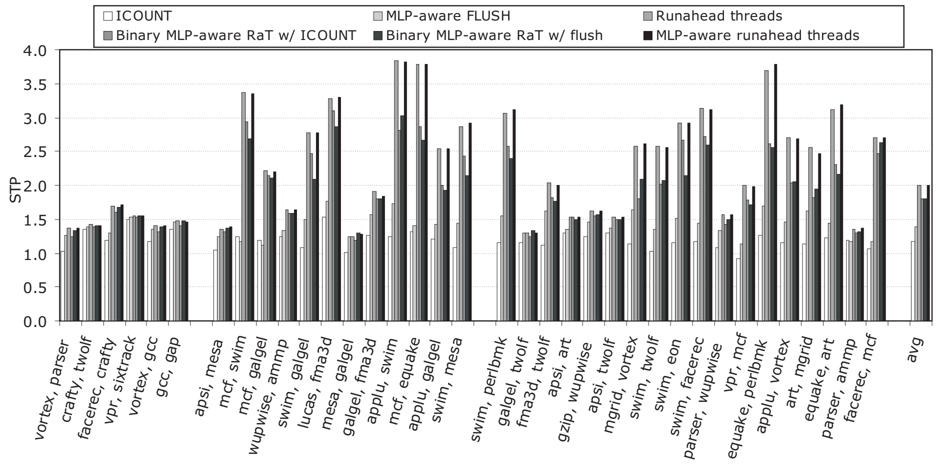

Fig. 3. Comparing MLP-aware runahead threads against other fetch SMT policies in terms of STP for two-program workloads: ILP-intensive workloads are shown on the left, MLP-intensive workloads are shown in the middle and mixed ILP/MLP-intensive workloads are shown on the right.

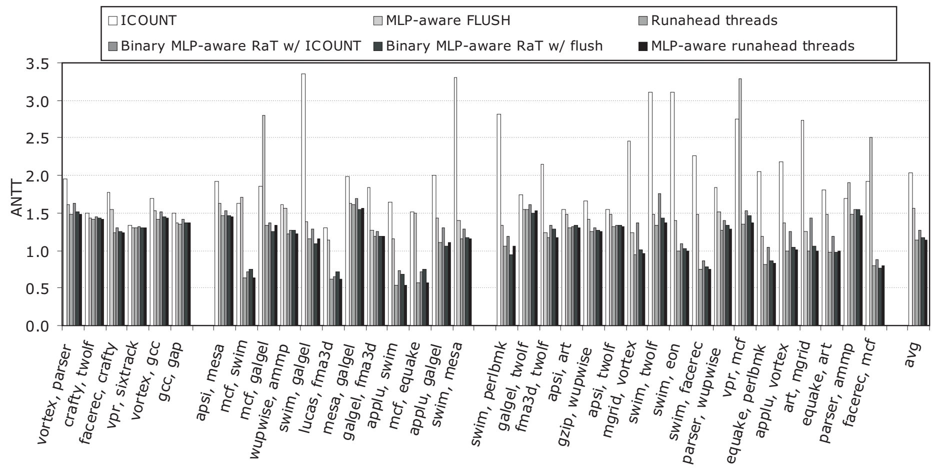

Fig. 4. Comparing MLP-aware runahead threads against other fetch SMT policies in terms of ANTT for two-program workloads: ILP-intensive workloads are shown on the left, MLP-intensive workloads are shown in the middle and mixed ILP/MLP-intensive workloads are shown on the right.

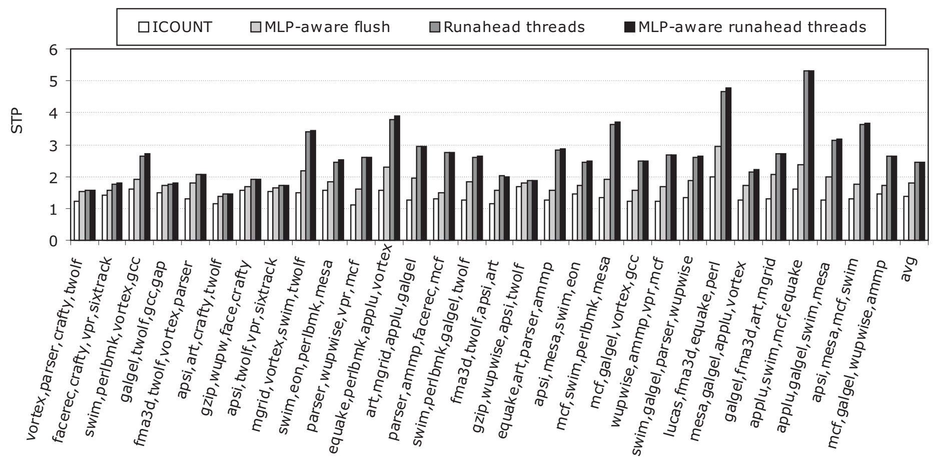

Fig. 5. Comparing MLP-aware runahead threads against other fetch SMT policies in terms of STP for four-program workloads.

more resources to the co-executing thread thereby improving the performance of the co-executing thread. For other workloads, on the other hand, MLP-aware runahead threads result in slightly worse performance compared to MLP-agnostic runahead threads, e.g., the worst performance is observed for art-mgrid: 3% reduction in STP and 0.3% increase in ANTT. These performance degradations are due to incorrect MLP distance predictions.

Figures 3 and 4 also clearly illustrate the effectiveness of MLP distance prediction versus binary MLP prediction. The MLP distance predictor is more effective than the binary MLP predictor proposed by Mutlu et al. [13]: i.e., STP improves by 11% on average and ANTT improves by 2.3% compared to the binary MLP-aware policy with flush; compared to the binary MLP-aware policy with ICOUNT, the MLP distance predictor improves STP by 11.5% and ANTT by 10%. The reason is twofold. First, the LLSR employed by the MLP distance predictor continuously monitors the MLP distance for each long-latency load. The binary MLP predictor by Mutlu et al. only checks for far-distance MLP through runahead execution; as runahead execution is not initiated for each long-latency load, it provides partial MLP information only. Second, the MLP distance predictor releases resources allocated by the long-latency thread as soon as the short-distance MLP (within half the reorder buffer) has been exploited. The binary MLP-aware policy on the other hand clogs resources (through the ICOUNT mechanism) or does not exploit short-distance MLP (through the flush policy).

6.3 Four-program workloads

Figures 5 and 6 show STP and ANTT, respectively, for the four-program workloads. The overall conclusion is similar as for two-program workloads: MLP-aware runahead threads achieve similar performance as MLP-agnostic runahead threads. The performance improvements are slightly higher though for the four-program workloads than

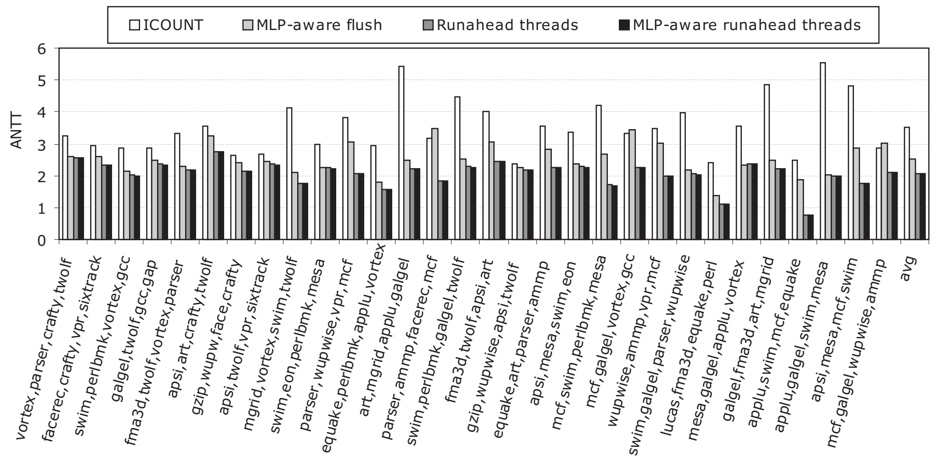

Fig. 6. Comparing MLP-aware runahead threads against other fetch SMT policies in terms of ANTT for four-program workloads.

for the two-program workloads because the co-executing programs compete more for the shared resources on a four-threaded SMT processor than on a two-threaded SMT processor. Making the runahead threads MLP-aware provides more shared resources for the co-executing programs which improves both single-program performance as well as overall system performance.

6.4 Reduction in speculatively executed instructions

As mentioned before, the main motivation for making runahead MLP-aware is to reduce the number of useless runahead executions, and thereby reduce the number of speculatively executed instructions under runahead execution in order to reduce energy consumption. Figure 7 quantifies the normalized number of speculatively executed instructions compared to MLP-agnostic runahead threads. MLP-aware runahead threads reduce the number of speculatively executed instructions by 13.9% on average; this is due to eliminating useless runahead execution periods. (We obtain similar results for the four-program workloads with an average 10.1% reduction in the number of speculatively executed instructions; these results are not shown here because of space constraints.) Binary MLP-aware runahead threads with ICOUNT and flush achieve higher reductions in the number of speculatively executed instructions (23.7% and 27%, respectively), however, this comes at the cost of reduced performance (by 11% to 11.5% in STP and 2.3% to 10% in ANTT) as previously shown.

7 Related Work

There are two ways of partitioning the resources in an SMT processor. One approach is static partitioning [16] as done in the Intel Pentium 4 [9], in which each thread gets an equal share of the resources. Static partitioning solves the long-latency load problem:

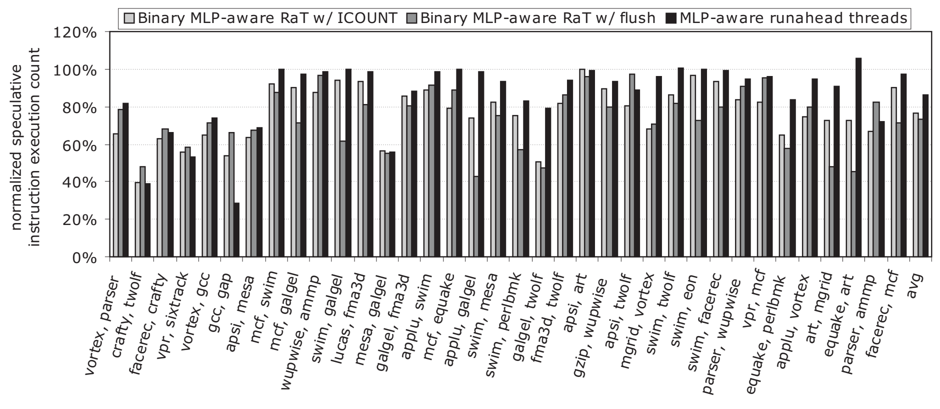

Fig. 7. Normalized speculative instruction count compared to MLP-agnostic runahead threads for the two-program workloads.

a long-latency thread cannot clog resources, however, it does not provide flexibility: a resource that is not being used by one thread cannot be used by the other thread(s).

The second approach, called dynamic partitioning, on the other hand provides flexibility by allowing multiple threads to share resources, however, preventing long-latency threads from clogging resources is a challenge. In dynamic partitioning, the fetch policy typically determines what thread to fetch instructions from in each cycle and by consequence, the fetch policy also implicitly manages the shared resources. Several fetch policies have been proposed in the recent literature. ICOUNT [22] prioritizes threads with fewer instructions in the pipeline. The limitation of ICOUNT is that in case of a long-latency load, ICOUNT may continue allocating resources for the blocking long-latency thread; by consequence, these resources will be hold by the blocking thread and will prevent the other thread(s) from allocating these resources. In response to this problem, Tullsen and Brown [21] proposed two schemes for handling long-latency loads, namely (i) fetch stall the long-latency thread, and (ii) flush instructions fetched passed the long-latency load in order to deallocate resources. Cazorla et al. [1] 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 all threads wait for a long-latency load. Eyerman and Eeckhout [6] made the flush policy MLP-aware in order to preserve the available MLP upon a flush or fetch stall on a long-latency thread.

An alternative approach is to drive the fetch policy through explicit resource partitioning. For example, Cazorla et al. [2] 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, DCRA dynamically determines the amount of resources required by each thread and prevents threads from using more resources than they are entitled to. However, 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.

El-Moursy and Albonesi [5] propose to give fewer resources to threads that experience many data cache misses in order to minimize issue queue occupancies for saving energy. They propose two schemes for doing so, 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. When the number of L1 data cache misses exceeds a given threshold, the thread is fetch gated. 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 front-end pipeline stages.

8 Conclusion

Runahead threads solve the long-latency load problem in an SMT processor elegantly by exposing (far-distance) memory-level parallelism while not clogging shared processor resources. A limitation though of existing runahead SMT execution proposals is that runahead execution is initiated upon a long-latency load irrespective of whether there is far-distance MLP to be exploited. A useless runahead execution, i.e., one along which there is no exploitable MLP, thus wastes execution resources and energy.

This paper proposed MLP-aware runahead threads to reduce the number of useless runahead periods. In case the MLP distance predictor predicts there is far-distance MLP to be exploited, the long-latency thread enters runahead execution. If not, the long-latency thread is flushed or fetch stalled per the predicted MLP distance. By doing so, runahead execution consumes resources only in case of long-distance MLP; if not, the MLP-aware flush policy frees allocated resources while exposing short-distance MLP, if available. Our experimental results report an average reduction of 13.9% in the number of speculatively executed instructions compared to MLP-agnostic runahead threads for two-program workloads while incurring no performance degradation; for four-program workloads, we report a 10.1% reduction in the number of speculatively executed instructions. Previously proposed binary MLP prediction achieves greater reductions in the number of speculatively executed instructions (by 23.7% to 27% on average) compared to MLP-agnostic runahead threads, however, it incurs an average 11% to 11.5% reduction in system throughput and an average 2.3% to 10% reduction in average job turnaround time.

Acknowledgements

We would like to thank the anonymous reviewers for their valuable comments. Stijn Eyerman and Lieven Eeckhout are postdoctoral fellows with the Fund for Scientific Research in Flanders (Belgium) (FWO-Vlaanderen). Additional support is provided by the FWO projects G.0160.02 and G.0255.08.

References

1. F. J. Cazorla, E. Fernandez, A. Ramirez, and M. Valero. Optimizing long-latency-load-aware fetch policies for SMT processors. *International Journal of High Performance Computing and Networking (IJHPCN)*, 2(1):45–54, 2004.
2. F. J. Cazorla, A. Ramirez, M. Valero, and E. Fernandez. Dynamically controlled resource allocation in SMT processors. In *MICRO*, pages 171–182, Dec. 2004.
3. Y. Chou, B. Fahs, and S. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *ISCA*, pages 76–87, June 2004.
4. J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *ICS*, pages 68–75, July 1997.
5. A. El-Moursy and D. H. Albonesi. Front-end policies for improved issue efficiency in SMT processors. In *HPCA*, pages 31–40, Feb. 2003.
6. S. Eyerman and L. Eeckhout. A memory-level parallelism aware fetch policy for SMT processors. In *HPCA*, pages 240–249, Feb. 2007.
7. S. Eyerman and L. Eeckhout. System-level performance metrics for multi-program workloads. *IEEE Micro*, 28(3):42–53, May/June 2008.
8. A. Glew. MLP yes! ILP no! In *ASPLOS Wild and Crazy Idea Session*, Oct. 1998.
9. G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, Q1 2001.
10. L. K. John. Aggregating performance metrics over a benchmark suite. In L. K. John and L. Eeckhout, editors, *Performance Evaluation and Benchmarking*, pages 47–58. CRC Press, 2006.
11. R. E. Kessler, E. J. McLellan, and D. A. Webb. The Alpha 21264 microprocessor architecture. In *ICCD*, pages 90–95, Oct. 1998.
12. K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. In *ISPASS*, pages 164–171, Nov. 2001.
13. O. Mutlu, H. Kim, and Y. N. Patt. Techniques for efficient processing in runahead execution engines. In *ISCA*, pages 370–381, June 2005.
14. 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.
15. E. Perelman, G. Hamerly, and B. Calder. Picking statistically valid and early simulation points. In *PACT*, pages 244–256, Sept. 2003.
16. S. E. Raasch and S. K. Reinhardt. The impact of resource partitioning on SMT processors. In *PACT*, pages 15–26, Sept. 2003.
17. T. Ramirez, A. Pajuelo, O. J. Santana, and M. Valero. Runahead threads to improve SMT performance. In *HPCA*, pages 149–158, Feb. 2008.
18. T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS*, pages 45–57, Oct. 2002.
19. A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for simultaneous multithreading processor. In *ASPLOS*, pages 234–244, Nov. 2000.
20. D. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *Proceedings of the 22nd Annual Computer Measurement Group Conference*, Dec. 1996.
21. D. M. Tullsen and J. A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *MICRO*, pages 318–327, Dec. 2001.
22. 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.
23. D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *ISCA*, pages 392–403, June 1995.