Efficient Runahead Threads

Tanausú Ramírez
Dept. Arquitectura de Computadores
Universitat Politecnica de Catalunya
Barcelona, Spain
tramirez@ac.upc.edu

Alex Pajuelo
Dept. Arquitectura de Computadores
Universitat Politecnica de Catalunya
Barcelona, Spain
mpajuelo@ac.upc.edu

Oliverio J. Santana
Dept. Informática y Sistemas
University of Las Palmas de Gran Canaria
Las Palmas de GC, Spain
ojsantana@dis.ulpgc.es

Onur Mutlu
Dept. of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA, USA
onur@cmu.edu

Mateo Valero
DAC - UPC
BSC - Centro Nacional Supercomputación
Barcelona, Spain
mateo@ac.upc.edu

ABSTRACT

Runahead Threads (RaT) is a promising solution that enables a thread to speculatively run ahead and prefetch data instead of stalling for a long-latency load in a simultaneous multithreading processor. With this capability, RaT can reduces resource monopolization due to memory-intensive threads and exploits memory-level parallelism, improving both system performance and single-thread performance. Unfortunately, the benefits of RaT come at the expense of increasing the number of executed instructions, which adversely affects its energy efficiency.

In this paper, we propose Runahead Distance Prediction (RDP), a simple technique to improve the efficiency of Runahead Threads. The main idea of the RDP mechanism is to predict how far a thread should run ahead speculatively such that speculative execution is useful. By limiting the runahead distance of a thread, we generate efficient runahead threads that avoid unnecessary speculative execution and enhance RaT energy efficiency. By reducing runahead-based speculation when it is predicted to be not useful, RDP also allows shared resources to be efficiently used by non-speculative threads. Our results show that RDP significantly reduces power consumption while maintaining the performance of RaT, providing better performance and energy balance than previous proposals in the field.

Categories and Subject Descriptors

B.8 [**Hardware**]: Performance and Reliability—*General, Performance Analysis and Design Aids*; C.1.3 [**Processor Architectures**]: Other Architecture Styles—*SMT processors, Out-of-order processors*

PACT'10, September 11–15, 2010, Vienna, Austria.

General Terms

Measurement, Performance, Experimentation

Keywords

Simultaneous multithreading, Runahead, energy-efficiency

1. INTRODUCTION

Multi-threaded processors provide support for many simultaneous hardware threads of execution in various ways, including Simultaneous Multithreading (SMT) [22][25] and Chip Multiprocessing (CMP) [15]. Regardless of how many cores are available in the future, the ability to deal with varying (due to novel on-chip memory hierarchies) and large memory latencies will continue to be very important, especially for multithreading processors due to different threads' performance and interference in shared resources.

In this scenario, processor architects and manufacturers have noticed the importance of the memory wall problem [24] and resource contention in SMT/CMT processors. Recently, several proposed techniques exploit memory-level parallelism (MLP) [10] of memory-intensive threads while preventing them from clogging shared resources. An interesting solution is Runahead Threads (RaT) [16]. RaT detects when a thread executes a long-latency load, and then it turns the offending thread into a *runahead thread* while the miss is outstanding. SMT processors have two main advantages using RaT: 1) by allowing runahead execution on the L2-missing thread, RaT prefetches data and improves the individual performance of that thread, 2) by allowing a thread to execute speculatively rather than stall on an L2 miss, RaT ensures that this thread (that incurs an L2 miss) recycles faster the resources it uses instead of clogging up shared resources that could be used by other threads.

However, while RaT is an interesting solution that both improves thread performance and reduces resource clogging in SMT processors, this mechanism has an important shortcoming: its benefits come at the cost of executing a large number of instructions due to runahead speculative execution. When this speculative runahead execution does not expose prefetching through MLP, it does not contribute to

performance improvement. As a consequence, that runahead thread execution is not useful from the thread performance point of view and degrades energy efficiency of the processor by executing a large number of useless instructions.

In this paper, we aim to improve the efficiency of the runahead threads to alleviate this drawback. We focus on reducing the useless speculative work done by runahead threads as much as possible without degrading their performance benefits. To this end, we introduce a prediction mechanism which is guided by the "useful runahead distance", a new concept that indicates how far a thread should run ahead such that speculative runahead execution is useful. Based on this distance information obtained dynamically, our technique takes two actions. First, it decides *whether or not* a thread should start runahead execution, (i.e. whether or not runahead execution is useful) to avoid the execution of useless runahead periods. Second, it predicts *how long* the thread should execute in runahead mode to reduce unnecessary extra work done by useful runahead periods. With these capabilities, this new mechanism indirectly predicts the maximum MLP achievable by a particular runahead thread while at the same time reducing the extra speculative work.

This paper also provides new, quantitative full-chip power and energy-delay results for runahead execution, unlike previous research in runahead efficiency that report only the reduction in speculative instructions. According to these results, our new proposal reduces the power consumption of RaT on SMT processors by 14% on average, without significantly affecting its performance. Moreover, it provides the best energy-delay square ratio compared to other efficient runahead execution techniques used with RaT.

The remainder of this paper is organized as follows. We provide a brief discussion of the previous work in Section 2. In Section 3 we describe in detail our technique to optimize runahead threads. Section 4 describes our experimental framework. We provide a detailed evaluation comparing our proposal to prior work in Section 5 and analysis and data about our proposal in Section 6. Finally, we present our conclusions in Section 7.

2. RELATED WORK

Memory-Level Parallelism (MLP) refers to the idea of generating multiple cache misses in parallel such that their latencies are overlapped [10]. Previous research has shown that exploiting MLP is an effective approach for improving the performance of memory bound applications [5][11]. Recent techniques focus on exploiting MLP of threads to address the long-latency load problem on SMT processors. The prevailing idea behind these approaches is that a thread can continue executing after it incurs a long-latency cache miss to overlap multiple independent long-latency loads. The goal is to preserve single thread performance in the presence of long-latency loads within the context of an SMT processor while ensuring that a memory bound thread does not hold on to too many processor resources.

Two MLP-aware techniques [8] were built on top of stall and flush fetch policies [20] to achieve this objective by using an MLP predictor. Using this MLP predictor, these fetch policies decide to (1) stall or flush the L2-missing thread if there is no MLP or (2) continue executing as many instructions as predicted by the MLP predictor. After that, the thread is either stalled (MLP stall) or flushed (MLP flush). Due to their design, the disadvantage of these techniques is that they are limited by the reorder buffer size in the amount of MLP they can exploit. Thus, a thread that misses in the L2 cache cannot execute more instructions than the reorder buffer size permits, which cannot be scaled without introducing large additional complexity.

The Runahead Threads (RaT) approach [16] exploits MLP by applying runahead execution [14] to any running thread when a long-latency load is pending. RaT allows memory-intensive threads to advance speculatively in a multithreaded environment instead of stalling the thread, doing beneficial work (prefetching) to improve the performance. RaT reaches further MLP going speculatively beyond the processor's instruction window, overcoming the limitation of MLP-aware techniques aforementioned. However, RaT has no information about the existence of MLP for a particular runahead period and therefore it performs useless extra work if there is no available MLP.

Previous research [13] proposed techniques to improve the efficiency of runahead execution in single-thread processors. That work identified three causes of inefficiency in runahead execution (short, overlapping, and useless runahead periods) and proposed several techniques to reduce their occurrences. To eliminate short and overlapping periods, threshold-based heuristics are used. To eliminate useless runahead periods, that work proposes a technique with a binary MLP predictor based on two-bit saturating counters. In case there is no MLP to be exploited, a runahead period is not initiated at all. In parallel to our work, another approach [7] combines the advantages of both MLP-aware flush and RaT mechanisms. This proposal predicts the MLP only in order to decide whether the thread goes into full runahead execution (far-distance MLP) or the thread is flushed/stalled (short-distance MLP). However, in these previous proposals, if a runahead period is activated, the processor will stay in runahead mode until the load that misses in the L2 is completely serviced.

We develop a different mechanism for improving the energy efficiency of RaT. The new idea in our proposal is to predict *how long* a thread should stay in runahead mode. In contrast, existing proposals predict only *whether or not* to enter in a full runahead execution episode (i.e., a runahead episode that ends when the runahead-causing miss is serviced). For this very reason, as we will show, our mechanism is able to eliminate more useless speculative instructions (even in useful runahead periods) than previous proposed techniques.

3. A TECHNIQUE FOR EXECUTING EFFICIENT RUNAHEAD THREADS

Previous studies [16][7] have shown that RaT provides better SMT performance and fairness than prior SMT resource management policies. Nevertheless, its advantages come at the cost of executing a large amount of useless speculative work. The efficiency of Runahead threads can be indirectly related to the performance improvement provided by prefetching and the number of additional speculative instructions due to runahead execution. This efficiency relation is illustrated in Figure 1. This figure shows the increase in executed instruction (bars) and the corresponding increase in performance (triangular marks) due to RaT over a baseline SMT processor with ICOUNT fetch policy [21] for

different evaluated workloads (see Section 4 for the experimental framework). Looking at the averages (AVG) for each set of multithreaded workloads, RaT increases the number of executed instructions by 41% for 2-thread and 42% for 4-thread workloads to improve the throughput performance by 52% and 21% respectively compared to ICOUNT.

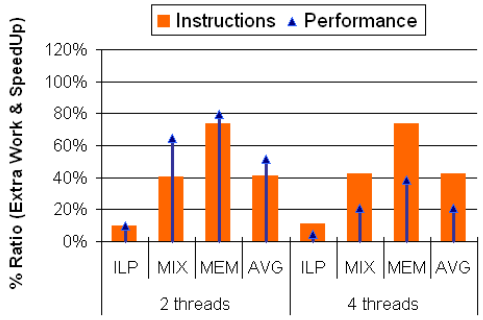


Figure 1: Extra executed instructions vs. performance SpeedUp due to Runahead Threads.

Since the execution of every instruction requires energy, all of this extra work directly affects the processor's power consumption. We measure and analyze this extra work in terms of energy (see Figure 7 in Section 5.2), and based on our studies, RaT consumes on average 57% more power for 2-thread workloads and 43% for 4-thread workloads than the basic ICOUNT fetch policy. Consequently, RaT requires more energy to increase the performance of the SMT processor. Even though the SMT processor using RaT is higher performance than ICOUNT, it is desirable to control the number of useless instructions executed by runahead threads to get improve energy-efficiency.

3.1 Runahead Distance

The usefulness of a runahead thread depends on the total amount of prefetching that a particular runahead thread is able to do during its speculative execution. Hence, in order to improve RaT efficiency, we would like to stop executing the runahead thread when it finishes generating useful prefetch requests, that is, there is no more MLP to be exploited. Therefore, we want to determine the useful lifetime of a runahead thread to expose as much MLP as possible with the minimum number of speculative instructions.

A full runahead thread execution consists of the total number of instructions that a particular thread executes from the L2-miss load that triggers a runahead execution to the last runahead instruction executed before flushing the pipeline when that L2-miss is resolved. Figure 2 illustrates an example of a runahead thread execution generated by a long-latency load miss. This figure depicts all instructions executed by a particular runahead thread, indicating the long-latency loads (LLLi). After the execution of LLL4, there are no more long-latency loads until the runahead-causing load is serviced. Intuitively, executing a runahead thread beyond the LLL4 instruction does not provide significant performance benefit and, in addition, it wastes energy. Based on this observation, we define *useful runahead distance* as the number of executed instructions between the L2-miss load that triggers the runahead thread and the last speculatively-executed load instruction that caused an L2 miss in a particular runahead thread execution (i.e., LLL4 in the example illustrated in the figure).

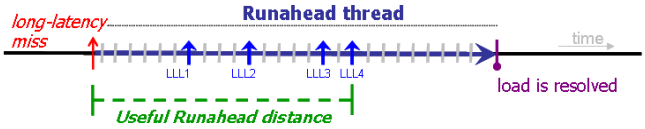


Figure 2: Illustration of the "useful runahead distance" under a runahead thread execution

The useful runahead distance can be tracked on a per static load basis, and the runahead distance of the next runahead episode caused by that load can be predicted by the useful distance recorded in previous runahead instances caused by that load. One important data in our studies is that only 12% of static loads generate 90% of all runahead threads (similar results for cache misses were shown in [6]). According to this analysis, it would be possible to track the useful runahead distance of the vast majority of runahead threads by tracking a small number of static loads.

3.2 Runahead Distance Prediction

To this end, we propose *Runahead Distance Prediction (RDP)*, a mechanism that predicts and decides how many instructions a runahead thread should execute. The idea of this mechanism is to capture dynamically the *useful runahead distance* of runahead threads caused by each static load. If the runahead distance is stable, the next runahead thread initiated by that load is run only up to the useful runahead distance with the hope that no useless instructions that do not benefit performance will be executed. In case the predicted useful distance is low, RDP does not initiate a runahead thread because the benefit of runahead would likely be lower than its cost (i.e., the runahead thread is very likely fully useless for performance).

With this ability, RDP has three major benefits: 1) it reduces the extra dynamic energy consumption due to executing useless speculative instructions, 2) it reduces the pressure that useless speculative instructions exert on shared SMT resources, thereby allowing other threads in the processor to make faster progress and thus improving the utilization of the SMT system, 3) it minimizes other possible causes of thread performance degradation (e.g., due to cache pollution or bandwidth interference) that might be caused by executing useless runahead instructions.

3.2.1 RDP Overview

RDP is based on learning the useful runahead distances of previous runahead thread executions caused by the same static load instruction to predict the useful runahead distance of future runahead periods for that load. Based on this prediction, RDP decides (1) whether or not a possible runahead thread should be initiated on an L2-miss load (avoiding the execution of useless runahead periods) and (2) if runahead thread is initiated, how long the runahead execution should be according to the useful runahead distance. With this prediction mechanism, (1) eliminate useless runahead threads that do not provide performance benefit, 2) not execute the final portions of runahead threads that do not provide performance benefit.

We propose an RDP predictor that keeps track of two runahead distance values per load: 1) the useful runahead distance obtained in the last *full*-length runahead execution

period, called *full useful distance*, 2) the useful runahead distance of the *last* runahead execution period, called *last useful distance*. The former is a register that is updated each time a runahead thread is fully executed (i.e., executed until the runahead-causing miss is fully serviced). The latter is a register that is always updated at the end of a runahead episode. To store the two useful runahead distance values associated with each L2-miss static load, we include a tagged table, called Runahead Distance Information Table (RDIT). As we show in Figure 3, each entry of the RDIT contains the load PC tag, the thread identifier, the two runahead distances: the *last* and the *full*, and a 2-bit counter that we explain below its functionality.

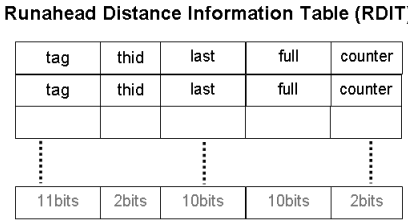


Figure 3: Runahead Distance Information Table

3.2.2 RDP Description

Figure 4 shows a flow diagram that summarizes the how the RDP mechanism works. First of all, when a load misses in the L2 cache, RDIT is accessed to obtain the runahead distance information (i.e., the full and last useful distance values) for a particular long-latency load (1). The first time a static load accesses the RDIT, the table has no distance information for that load. For this reason, when this load reaches the head of the reorder buffer, a runahead thread is activated and it is executed until the L2 miss is fully serviced. At the end of this runahead execution, the load that caused the runahead thread activation allocates a new entry in the RDIT and stores its observed useful runahead distance in both full and last useful distance fields because the runahead episode was executed in full.

Later, when the same load misses in the L2 cache again during normal thread execution, it accesses the RDIT to obtain the useful runahead distance associated with it. Initially, RDP checks whether the recorded distance values are reliable (2). To do this, RDP computes the difference between the *full* and the *last* runahead distance values and compares the result to a threshold value (called Distance Difference Threshold - DDT). If the distance difference is larger than DDT, the stored distance information is considered unreliable (or unstable). In this case, the runahead episode is executed in full (that is, until the runahead-causing L2-miss load is fully serviced). Both *last* and *full* useful distance values are updated again at the end of this runahead episode.[1] We have assessed several values for DDT and we found that a threshold value of 64 obtains the best tradeoff between performance and efficiency. We show this DDT analysis in section 6.1.

[1]Hence, RDP corrects the useful runahead distance value if the last distance has become significantly different than the previous ones.

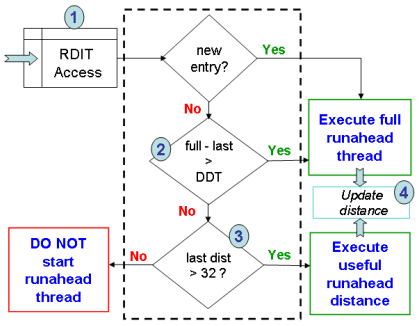


Figure 4: Flowchart of Runahead Distance Prediction technique

On the other hand, if the difference between last and full distances is smaller than DDT, the distance values are considered to be reliable. In this case, the *last* useful distance value is used to decide whether or not to start the runahead thread and to control how long the runahead thread should be executed. RDP decides to start a runahead thread if the value of the *last* useful runahead distance is above a particular activation threshold (3). We select this threshold based on a study of the number of instructions that can be executed from the time a candidate runahead load is detected to the time the load reaches the head of the ROB. We set this threshold value in 32 instructions based on the global average result of this study for the different workloads. Therefore, if the *last* useful distance is below this activation threshold value (32), it means that the possible (near) MLP can be exploited without entering runahead execution, since all overlapping loads would be issued by the out-of-order execution window. Otherwise (if the *last* useful distance value is greater than this threshold), RDP initiates runahead execution and ensures that the runahead threads executes at most as many instructions as indicated by the *last* useful distance value. Once the runahead thread executes as many instructions as the *last* useful runahead distance indicates, RDP updates the corresponding *last* useful runahead distance field in RDIT (4) for the runahead-causing load based on the observed useful distance in this very last period. We want to note that the distance comparison (between current distance and the last useful distance) needed to terminate the runahead thread to ensure it executes no more than last useful distance number of instructions is performed concurrently with the retirement stage during the runahead speculative commit process.

3.2.3 Ensuring Reliability of Small Useful Distances

According to the initial analysis with the proposed RDP design, we found that some runahead periods caused by a load initially have a small useful runahead distance (small enough to not start a runahead thread), but that useful runahead distance becomes larger later. Our study shows that 38% of static loads had a small useful runahead distance at first and 26% of static loads increase their useful runahead distance value later. As a result, if we use information from the early runahead instances of these loads, these loads would never initiate runahead later even though their runahead periods might become useful in the future. Consequently, performance degrades because useful runahead periods are eliminated. If RDP cannot take these cases into account, it could degrade performance compared to baseline RaT.

To prevent this, we use a simple heuristic that allows our mechanism to build confidence in a small useful runahead distance before enforcing. The key idea is to monitor how many times the static load instruction caused a runahead episode that is smaller than a threshold. In this case, we only allow RDP to enforce a small runahead distance if the static load has been very consistent in causing runahead episodes with small useful distances. To implement this idea, if the*last* useful runahead distance is lower than starting runahead thread condition (e.g. a distance less than 32) in the last N times for a load instruction, RDP initiates full runahead execution for that load instruction (instead of not starting runahead mode). Note that we enforce a limit of N for this heuristic because if we always enable full runahead execution, we will ignore when the runahead periods are truly useless, diminishing our technique's ability to reduce the useless extra work. After performing an exploration of the design space, our experiments show that a value of N=3 performs well. So, we use a 2-bit counter per-entry in RDIT to perform this countdown (Figure 3). This new heuristic essentially provides an additional confidence mechanism that determines whether or not a small distance is reliable.

3.2.4 Implementation issues

Runahead Distance Prediction described in this section is a simple mechanism that aims to improve the efficiency of runahead threads with low hardware cost and complexity. Basically, the processor needs to track the last-issued L2 miss load in each runahead thread execution to compute the useful runahead distance. To this end, we introduce the use of two new registers per hardware context: the *total-runahead-distance counter* and the *useful-runahead-distance register*. The first one is incremented per each instruction pseudo-retired during a runahead period. The second one holds the last-observed useful runahead distance for a particular thread execution. Thus, when a runahead thread starts, both registers are set to zero. Then, whenever a long-latency load is encountered during runahead execution, the total-runahead-distance counter value is copied into the useful-runahead-distance register. This procedure continues until the runahead thread ends. At this point in time, the value of the useful-runahead-distance register is used to update the RDIT for the corresponding load.

The RDIT configuration consists of a 4-way table of 1024 entries. So, the total RDIT size according to the different fields of each entry described in Figure 3 is around 4.5KB (which is much less than 1% of the L2 cache area). We used CACTI [18] to determine the access time to the RDIT. Assuming a 0.65nm technology and a 2GHz frequency, RDIT access/update can be performed in less than one cycle. As the useful runahead distance can depend on the program path leading to the L2-miss load instruction, the RDIT is indexed by an XOR function of the program counter of the load and 2 bits from the two previous conditional branches history. Access and update of RDIT are out of the critical path of execution. First, RDIT is accessed when a load misses in the L2 cache, which is a relatively infrequent event, and the required runahead distance prediction is not needed till the load reaches the head of the ROB. Second, RDIT is updated when the processor terminates a runahead thread. So, RDIT update latency can therefore be overlapped with the pipeline flush at the end of runahead thread.

4. EXPERIMENTAL SETUP

We use a dynamic resource partitioning model in our SMT processor. In this model, threads coexist in the different pipeline stages sharing the important hardware resources (issue queues, rename registers, functional units, caches, memory system). To prioritize threads in the fetch stage, we use the ICOUNT(2,8) as the underlying policy, which selects two different threads and fetches up to eight instructions from each thread in each cycle. The modeled memory subsystem includes a two level cache organization and the main memory with the corresponding latencies. We set the main memory latency to a minimum of 300 cycles to emulate the importance of the memory wall problem. The processor also employs a similar aggressive prefetcher based on the stream-based stride predictor proposed in [9], that can detect and generate prefetches for different access streams into the L2 cache. The main configuration parameters of the SMT processor model are listed in Table 1.

Processor core	
Pipeline depth	10 stages
Fetch/decode/issue/execute width	8 way
Reorder buffer size	64 entries per context
INT/FP registers	32 arch. + 48 phys. per context
INT/FP/LS issue queues	64 / 64 / 64
INT/FP/LdSt units	4 / 4 / 2
Branch predictor	perceptron (256-entry)
Hardware prefetcher	Stream buffers (16)
Memory subsystem	
Icache	64 KB, 4-way, 1-cycle access latency
Dcache	64 KB, 4-way, 3-cycle access latency
L2 Cache	1 MB, 8-way, 20-cycle access latency
Cache line size	64 bytes
Main memory latency	300-cycle minimum latency

Table 1: Baseline SMT processor configuration

We use an execution-driven SMT simulator derived from SMTSIM [19] including an enhanced power model based on Wattch [2]. We model the power for all the main hardware structures of the processor (functional units, registers, branch predictor, queues, rob, memory system, etc), including also the clock. With this accurate power model, we report power and energy consumption of the different mechanisms evaluated.

The experiments have been performed with multiprogramming workloads created by combining single-threaded benchmarks from the SPEC CPU2000 suite. All benchmarks were compiled on an Alpha AXP-21264 using the Compaq C/C++ compiler with the -O3 optimization level to obtain Alpha ISA binaries. For each benchmark, we simulate 300 million representative instructions using the reference input set. To identify representative simulation segments, we analyzed the distribution of basic blocks using SimPoint [17]. Measurements are then taken using the FAME [23] methodology. FAME re-executes all traces in a multiprogrammed workload until all of them are fairly represented in the final measurements.

To create the workloads, we characterized each program based on the L2 cache miss rate it attains when run on a single-threaded processor. A benchmark that has a high L2 cache miss rate is classified as a memory intensive benchmark, otherwise it is classified as computing intensive. We compose mixes of 2 and 4 threads workloads and create three different kinds of workload mixes: those consisting of computing-intensive benchmarks (ILP), those consisting of

ILP2	MIX2	MEM2	ILP4	MIX4	MEM4
apsi,eon	applu,vortex	applu,art	apsi,eon,fma3d,gcc	ammp,applu,apsi,eon	art,mcf,swim,twolf
apsi,gcc	art,gzip	art,mcf	apsi,eon,gzip,vortex	art,gap,twolf,crafty	art,mcf,vpr,swim
bzip2,vortex	bzip2,mcf	art,twolf	apsi,gap,wupwise,perl	art,mcf,fma3d,gcc	art,twolf,equake,mcf
fma3d,gcc	equake,bzip2	art,vpr	crafty,fma3d,apsi,vortex	gzip,twolf,bzip2,mcf	equake,parser,mcf,lucas
fma3d,mesa	galgel,equake	equake,swim	fma3d,gcc,gzip,vortex	lucas,crafty,equake,bzip2	equake,vpr,applu,twolf
gcc,mgrid	lucas,crafty	mcf,twolf	gzip,bzip2,eon,gcc	mcf,mesa,lucas,gzip	mcf,twolf,vpr,parser
gzip,bzip2	mcf,eon	parser,mcf	mesa,gzip,fma3d,bzip2	swim,fma3d,vpr,bzip2	parser,applu,swim,twolf
gzip,vortex	swim,mgrid	swim,mcf	wupwise,gcc,mgrid,galgel	swim,twolf,gzip,vortex	swim,applu,art,mcf
mgrid,galgel	twolf,apsi	swim,vpr			
wupwise,gcc	wupwise,twolf	twolf,swim			

Table 2: Multithreaded workloads grouped by categories

memory-intensive benchmarks (MEM), and those consisting of a mixture of the two (MIX). Table 2 shows these workloads.

5. EVALUATION

We quantitatively evaluate the performance and energy efficiency of the RDP technique and other related mechanisms through different metrics. Regarding SMT related techniques, we include the MLP-aware flush policy [8] (MLP-Flush) and the recent proposal MLP-aware runahead thread [7] (MLP-RaT). In addition, we apply the techniques in the single-threaded context for efficient runahead execution [13] to Runahead Threads in SMT. We label this combination as RaT-SET (Single-thread Execution Techniques -SET).

5.1 Performance

Figure 5(a) and 5(b) show the performance results of the mentioned techniques in terms of throughput (total IPC) and Hmean speedup [12] (harmonic mean of individual thread speed-ups) metrics respectively. Using these metrics, we evaluate the overall system performance of our SMT processor model, measuring the overall IPC and the performance-fairness balance of each technique regarding the individual thread performance.

Figure 5(a) demonstrates the performance improvement achievable using RaT mechanisms. In general, all RaT based mechanisms provide high performance, especially for MIX and MEM workloads in which there are memory-intensive threads involved. RDP obtains 33% speedup over the baseline ICOUNT, which is the closest throughput improvement to RaT (35%). RaT+SET and MLP-RaT provide 23% and 28% performance improvement respectively. In the RaT-SET case, the processor either executes the whole runahead period until the L2 miss returns or it does not enter runahead mode due to the heuristics and the binary prediction applied. So, RaT+SET eliminates extra work avoiding full runahead periods but it also eliminates useful runahead periods, thereby degrading performance by reducing prefetching benefits. In contrast, RDP is more fine-grained since the processor executes a runahead thread until the predicted runahead distance (i.e., until the runahead execution stops being useful), with the special case that if the predicted distance is smaller than the activation threshold, the processor does not enter runahead mode. On the other hand, while MLP-Flush improves the throughput over ICOUNT by 12% on average, this technique cannot improve the performance as much as RaT proposals mainly because it cannot go beyond the reorder buffer size to exploit prefetching due to distant MLP.

For Hmean metric, the performance results are similar as for throughput. RaT-SET, MLP-RaT and RDP obtain relatively close Hmean performance results to RaT. RaT provides 31% and 14% higher hmean speedup than ICOUNT for 2- and 4-thread workloads respectively. MLP-RaT and RDP provide similar results with slightly lower Hmean speedup (4% and 5% respectively). The predictor misspredictions in each technique degrades sometimes the individual IPC of memory-intensive threads compared to the original RaT mechanism, which reduces Hmean speedup. RaT-SET shows 6% slowdown for Hmean compared to RaT. We therefore conclude that RDP preserves the performance of runahead threads better than previous mechanisms.

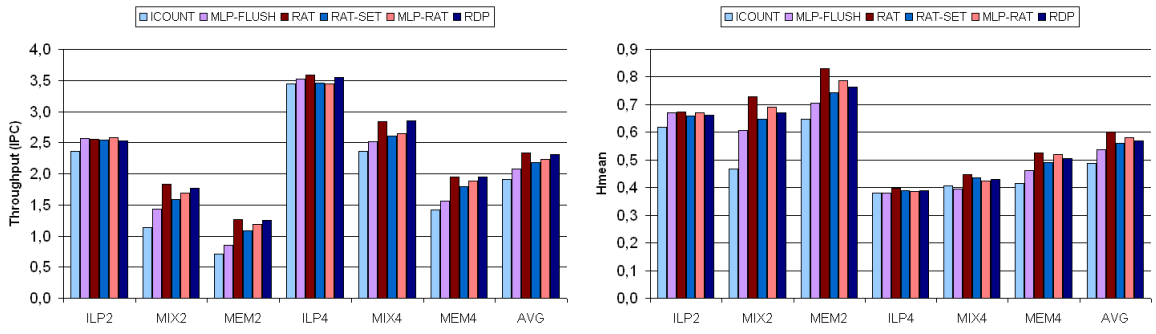
5.2 Energy Efficiency

We now study the implications of using the different techniques in terms of energy efficiency. We quantify the extra work reduction, the power consumption and performance-energy balance to show a detailed energy/power evaluation.

Figure 6 shows the normalized number of speculatively executed instructions of different techniques compared to baseline RaT (note that we only show the mechanisms that employ runahead threads in this figure). We also show the optimum instruction reduction ratio using an oracle technique (RDP Oracle) that is unimplementable in real hardware. This optimum reduction is obtained by executing each runahead thread for its ideal useful runahead distance, i.e. until the last instruction that generates an L2 miss in the episode is executed. As Figure 6 shows, the optimum reduction in speculative instructions is 64% on average.

According to Figure 6, MLP-RaT reduces the speculative work by 10% (extra flushed instructions due to flush mechanism also count for speculative instructions) whereas applying RaT-SET this reduction is 31%. In both mechanisms (MLP-RaT and RaT-SET), if a runahead thread is predicted to be executed, the processor fully executes the runahead thread until the L2 miss that caused entry into runahead is serviced. For this reason, even if runahead execution does not provide any benefits beyond some point in the runahead execution (that we called useful runahead distance), these mechanisms continue executing speculative useless instructions.

In contrast, the reduction in speculative instructions using RDP comes from eliminating both the useless runahead threads (as other techniques also try to do) and the useless runahead instructions at the end of useful runahead threads (thanks to the fine-grain runahead distance prediction). As



(a) Throughput (b) Hmean

Figure 5: Performance of our RDP versus efficient runahead and MLP-aware SMT fetch policies.

a result, RDP achieves the highest reduction in the amount of speculatively executed instructions: 44% on average according to Figure 6.

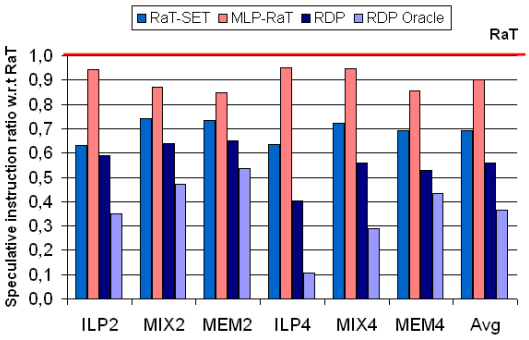


Figure 6: Fraction of speculative instructions normalized to RaT

To quantify the impact of this extra work reduction in terms of energy, Figure 7 shows the SMT processor power consumption when each different evaluated technique is employed. For this evaluation, we get the power for all important processor core and memory components using our power model during all workload executions. We faithfully model the power consumption of the additional structures required by each techniques, e.g. the RDIP in our proposal. To this end, we use CACTI to get the power consumption estimation per access of each additional hardware structure and integrate this value in our Wattch model. For instance, the basic power for the RDIT is 1.09 watts, which is less than 1% of total processor power consumption and it is much less than caches (4,7W for Icache and 9,4W for Dcache).

Our results show that the average dynamic power of the processor tends to be correlated with the number of executed instructions, similarly to Annavaram *et al.*'s observation [1]. For instance, the average power consumption for 4-thread workloads is higher than 2-thread ones mainly because IPC of the former is significantly higher. Looking the Figure7, all techniques consume more power than baseline SMT with ICOUNT. According to these power results, MLP-Flush mechanism consumes 16% whereas RaT consumes 42% more power than ICOUNT on average.

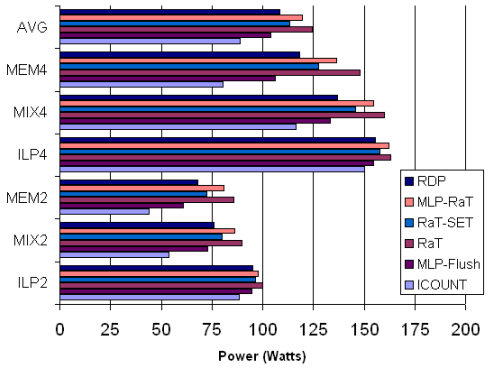


Figure 7: Average Power consumption

Among the techniques that control runahead thread executions, RaT-SET reduces the power requirements by 9% over RaT, although this reduction comes with a corresponding performance degradation as we have seen in Figure 5(a) (9% less performance compared to RaT). MLP-RaT achieves 4% average power reduction. In comparison, RDP effectively reduces the power consumption by 14% on average compared to RaT (up to 20% for MEM2 workloads) and by 10% compared to MLP-RaT. Hence, we conclude that the RDP mechanism is the most efficient technique among all RaT-based mechanisms.

Finally, we measure the energy-delay square product (ED^2) [3] of MLP-Flush, RaT-SET, MLP-RaT and RDP to assess energy efficiency. This voltage-invariant metric indicates the balance between performance and power and it gives us an energy efficiency measure for high-performance processors. A lower value of ED^2 is better, since it indicates a more efficient mechanism which minimizes the relation of power consumption and cycles per instruction.

Figure 8 shows the ED^2P ratio compared to the ICOUNT mechanism and then normalized that results to the original RaT mechanism. This figure shows that RDP provides the best ED^2 product results among all techniques (10% on average better than RaT). Particularly, RDP provides 7% lower ED^2P than RaT for 2-thread workloads. The results are even better for 4 threads, since RDP provides the best average improving ED^2P by 13% over RaT in these workloads. RaT-SET and MLP-RaT energy efficiency are lower,

with 24% and 33% ED^2 ratio degradation respectively compared to RDP. The reason RDP improves energy efficiency is twofolds: 1) RDP executes the lowest number of speculative instructions (44% compared to RaT) among all techniques, thereby causing less power consumption (14% reduction), 2) RDP provides similar performance to RaT. We conclude that RDP provides the highest energy efficiency compared to all other evaluated techniques.

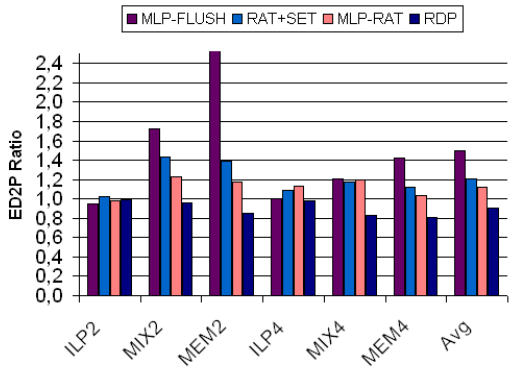


Figure 8: Energy-Delay^2 of evaluated Runahead thread techniques

6. RDP ANALYSIS

In this section, we analyze some features and relevant data about the proposed RDP technique. We provide insights about how this technique works and why RDP scheme is effective to support the previous results.

6.1 Distance difference threshold study

As we described in Section 3.2.2, the value of the distance difference threshold (DDT) fixes the confidence limit of the two runahead distances (last and full) for a particular load for the RDP mechanism. Figure 9 shows the relation of performance improvement and extra work reduction in terms of speculative instructions compared to RaT mechanism for different evaluated DDT values for RDP. The curve of performance represents the average performance throughput ratio for all workloads between RaT and RDP configurations with each DDT value. The curve of extra work depicts the percentage of speculative instruction reduction for the same experiments compared to RaT as well. We evaluate the DDT with values that range from 0 to 128. A DDT=0 is a very strict threshold in which only when the last and full runahead distance are exactly the same, they are considered reliable. Larger values involve a more flexible condition in this sense, therefore allowing runahead threads with some variability on computed runahead distances.

According to these results, when the DDT is increased, the extra work is reduced since larger DDT values make the obtained useful distances fulfill the DDT condition easily. So, executed runahead threads become shorter due to a less strict control that allows more differences among full and last useful runahead distances. However, for DDT values larger than 64, the RDP starts to lose performance with regard to RaT. If the distance threshold is bigger, the difference between full and last useful distances can be bigger as well. In these cases, the last useful runahead distance can be decremented with larger margins. This results in

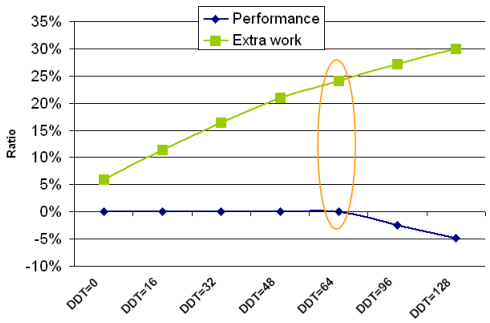


Figure 9: Effect of DDT value on the increase in performance and extra instructions compared to RaT

inaccurate useful runahead distances which cause the RDP technique to reduce the prefetching opportunities for the long-latency loads that cannot be overlapped in the same runahead thread. As a consequence, this effect generates more runahead threads due to future L2 cache misses that could not be avoided by useful previous prefetches which impact on the performance. Therefore, in order to keep the performance of the original RaT mechanism, we chose the DDT=64 as the threshold value with the best tradeoff between performance and efficiency based on the results for these experiments. We are studying as future work to generate and recalibrate dynamically this DDT in function of certain processor features and work load conditions.

6.2 Length of runahead threads

Figure 10 shows the average length of runahead threads (measured as number of instructions executed per runahead thread) using RaT or using RDP. We also show a striped part for RaT bars which indicates the average ideal useful distance for the executed runahead threads in this case. This *ideal* runahead distance represents the optimum distance calculated for each executed runahead thread with RaT according to our useful distance definition. This data provides valuable information about how well RDP manages the executed runahead threads to control the instruction execution with respect to the ideal computed useful runahead distances.

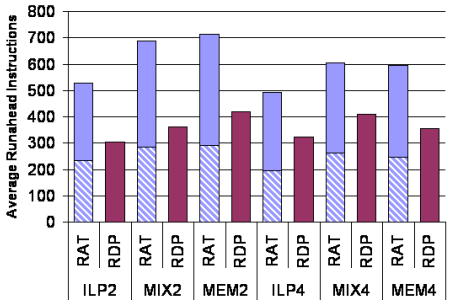


Figure 10: Avg number of instructions executed per runahead thread for RAT and RDP

The figure shows that RDP is successful at reducing the length of runahead threads. This is especially true for the MIX and MEM workloads. In addition, RDP executes more instructions than the ideal mechanism, which is expected because RDP cannot always accurately predict the useful

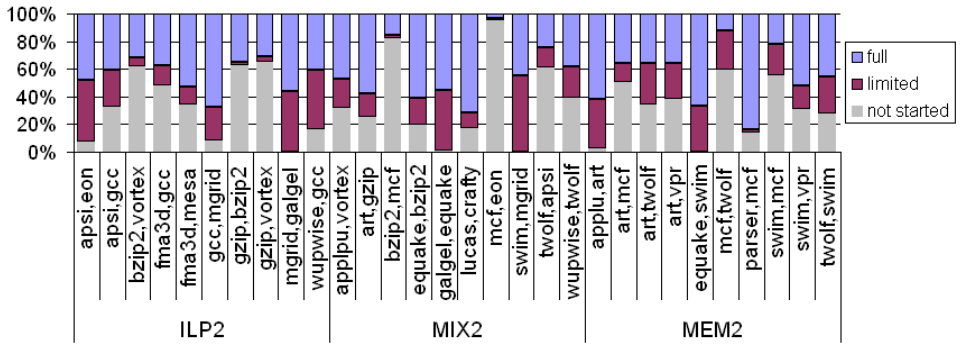


Figure 11: Distribution of runahead threads execution based on what RDP decided to do with them

distance of a runahead thread. With RDP, he average length reduction per runahead thread for 2-thread workload is 282 instructions whereas for 4-threads is 201 instructions. This reduction directly results in the energy savings of RDP.

6.3 Distribution of runahead threads

We show how RDP manages the different runahead threads according to the predicted useful distance. Figure 11 illustrates a more detailed analysis of the runahead thread distribution in terms of RDP decision for 2-thread workloads. Each bar is split in three parts: the percentage of runahead threads that were fully executed (full), the percentage of runahead threads whose execution length was limited by the predicted useful runahead distance (limited), and the percentage of runahead threads that were not even started (because their predicted useful distances do not fulfill the activation threshold, i.e. the distance was less than 32).

Figure 11 indicates diverse behavior among the workloads. For example, if we analyze how many times our technique eliminates a runahead thread completely, we can observe there are workloads with a high percentage of runahead threads suppressed ('not started' label). These are the cases of *bzip2-mcf* and *mcf-eon*. Because mcf is a benchmark with a huge number of dependent loads, it causes invalid runahead loads that do not issue prefetches which are not taken into account for useful distance computation. On average, the percentage of not initiated runahead threads due to small distances for 2-thread workloads is 34% (in the case of four threads this percentage is 37%). On the other hand, there are workloads which have a high ratio of runahead threads limited by the corresponding useful distance, for instance, *mgrid-galgel*, *galgel-equake*,*swim-mgrid* or *applu-art* (around 40% for them). This percentage is 22% for overall 2-thread workloads. Note that limiting the runahead distance leads to more efficient runahead threads because those threads execute fewer instructions to achieve the same benefit as if they were executed in full length.

To provide an in-depth examination of the previous results, Figure 12 shows a histogram of how many runahead threads execute between N and M instructions (in a range of 32 instructions) for a particular workload (*art,gzip*) using RaT and RDP[2]. The bar for RDP between 0-32 instructions indicates the number of runahead threads that were not started for this workload. RDP has fewer long runahead threads compared to RaT, but it has more short runahead threads than RaT. Effectively, RDP converts long runahead threads into shorter ones by eliminating their useless portions at the end that do not provide any prefetching benefits. The end result is a significant reduction in executed instructions and hence energy, as we have shown previously.

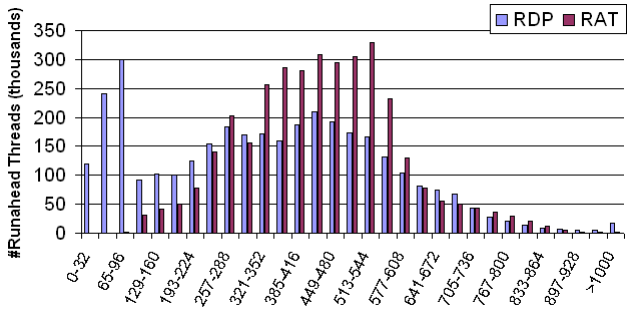


Figure 12: Runahead threads distance histogram for RDP and RaT

7. CONCLUSIONS

This paper developed a new technique, called Runahead Distance Prediction (RDP), to make runahead threads more efficient. The key idea is to predict how long a runahead thread should execute before it stops providing useful performance benefits and execute a runahead thread only for that useful "distance." The proposed mechanism also decides whether or not a runahead thread should be executed at all: if the predicted useful distance is too small, the execution of the runahead thread is suppressed. RDP is able to eliminate more instructions than previous runahead techniques because it can eliminate at a fine-grain the useless portion of runahead threads/episodes.

Our results show that our runahead distance prediction technique effectively reduces the speculatively executed instructions by 44% and thereby the dynamic power consumption due to runahead threads by 14% on average. RDP provides the best energy-efficiency ratio compared to previously proposed efficient runahead techniques with a 10% ED^2 better than the baseline Runahead Threads mechanism. We conclude that RDP is a new technique that can improve the efficiency of runahead mechanisms in isolation or in conjunction with other previous runahead efficiency techniques.

[2]Although we only show one example here, the remaining workloads follow a similar trend.

Acknowledgments

This work has been developed and supported by the Ministry of Education of Spain under contract TIN-2004-07739-C02-01 whereas the registration fees and travel costs by contract TIN2007-61763. We would like to thank Francisco Cazorla and Augusto Vega from Barcelona Supercomputing Center (BSC-CNS) for their useful help on integrating the Wattch power model with our simulator.

8. REFERENCES

[1] M. Annavaram, E. Grochowski, and J. Shen. Mitigating amdahls law through epi throttling. In *32nd International Symposium on Computer Architecture, (ISCA-32).*, pages 298–309, 2005.

[2] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *27th International Symposium on Computer Architecture (ISCA-27)*, 2000.

[3] D. M. Brooks, P. Bose, S. E. Schuster, H. Jacobson, P. N. Kudva, A. Buyuktosunoglu, J.-D. Wellman, V. Zyuban, M. Gupta, and P. W. Cook. Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. *IEEE Micro*, 20(6):26–44, 2000.

[4] S. Chaudhry, P. Caprioli, S. Yip, and M. Tremblay. High-performance throughput computing. *Micro, IEEE*, 25(3), 2005.

[5] Y. Chou, B. Fahs, and S. G. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *31th International Symposium on Computer Architecture, ISCA-31*, pages 76–89, 2004.

[6] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen. Speculative precomputation: long-range prefetching of delinquent loads. In *28th annual International Symposium on Computer Architecture (ISCA-28)*, 2001.

[7] K. Craeynest, S. Eyerman, and L. Eeckhout. Mlp-aware runahead threads in a simultaneous multithreading processor. In *4th International Conference on High Performance Embedded Architectures and Compilers (HiPEAC'09)*, pages 110–124, 2009.

[8] S. Eyerman and L. Eeckhout. A memory-level parallelism aware fetch policy for smt processors. In *International Symposium on High-Performance Computer Architecture, HPCA07*, 2007.

[9] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. Memory-system design considerations for dynamically-scheduled processors. In *24th Annual International Symposium on Computer Architecture (ISCA-24)*, pages 133–143, 1997.

[10] A. Glew. Mlp yes! ilp no! In *In Wild and Crazy Ideas Session, 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[11] T. Karkhanis and J. E. Smith. A day in the life of a data cache miss. *In Workshop on Memory Performance Issues.*, 2002.

[12] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in smt processors. In *International Symposium on Performance Analysis of Systems and Software (ISPASS'01)*, Tucson,AZ,USA,2001.

[13] O. Mutlu, H. Kim, and Y. N. Patt. Techniques for efficient processing in runahead execution engines. In *32th International Symposium on Computer Architecture, ISCA-32*, pages 370–381, 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 *International Symposium on High-Performance Computer Architecture, HPCA03*, page 129, Washington, DC, USA, 2003.

[15] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. *SIGOPS Oper. Syst. Rev.*, 30(5):2–11, 1996.

[16] T. Ramirez, A. Pajuelo, O. J. Santana, and M. Valero. Runahead threads to improve smt performance. In *International Symposium on High-Performance Computer Architecture, HPCA08*, page 129, Salt Lake City, UT, USA, 2008.

[17] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques, PACT'01*, pages 3–14, Washington, DC, USA, 2001.

[18] D. Tarjan, S. Thoziyoor, and N. P. Jouppi. Cacti 4.0. *Technical Report HPL-2006-86, HP Labs*, 2006.

[19] D. M. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *Int. Annual Computer Measurement Group Conference*, pages 819–828, 1996.

[20] D. M. Tullsen and J. A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *34rd International Symposium on Microarchitecture, (MICRO-34)*, Washington, DC, USA, 2001.

[21] 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 *23th International Symposium on Computer Architecture, (ISCA-23)*, NY, USA, 1996.

[22] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *22nd International Symposium on Computer Architecture, (ISCA-22)*, New York, NY, USA, 1995.

[23] J. Vera, F. J. Cazorla, A. Pajuelo, O. J. Santana, E. Fernandez, and M. Valero. A novel evaluation methodology to obtain fair measurements in multithreaded architectures. In *Workshop on Modeling, Benchmarking and Simulation*, 2006.

[24] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Computer Architecture News*, 23(1), 1995.

[25] W. Yamamoto and M. Nemirovsky. Increasing superscalar performance through multistreaming. In *International Conference on Parallel Architectures and Compilation Techniques, PACT'95*, pages 49–58, Manchester, UK, 1995.