A Flexible Heterogeneous Multi-Core Architecture

Miquel Pericàs[†⋆], Adrian Cristal[⋆], Francisco J. Cazorla[⋆],
Ruben González[†], Daniel A. Jiménez[‡] and Mateo Valero[†⋆]

[†]Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya
{mpericas,gonzalez,mateo}@ac.upc.edu

[⋆]Computer Sciences, Barcelona Supercomputing Center
{adrian.cristal,francisco.cazorla}@bsc.es

[‡]Department of Computer Science, The University of Texas at San Antonio
djimenez@acm.org

Abstract

Multi-core processors naturally exploit thread-level parallelism (TLP). However, extracting instruction-level parallelism (ILP) from individual applications or threads is still a challenge as application mixes in this environment are nonuniform. Thus, multi-core processors should be flexible enough to provide high throughput for uniform parallel applications as well as high performance for more general workloads. Heterogeneous architectures are a first step in this direction, but partitioning remains static and only roughly fits application requirements.

This paper proposes the Flexible Heterogeneous MultiCore processor (FMC), the first dynamic heterogeneous multi-core architecture capable of reconfiguring itself to fit application requirements without programmer intervention. The basic building block of this microarchitecture is a scalable, variable-size window microarchitecture that exploits the concept of Execution Locality *to provide large-window capabilities. This allows to overcome the memory wall for applications with high memory-level parallelism (MLP). The microarchitecture contains a set of small and fast* cache processors *that execute* high locality code *and a network of small in-order* memory engines *that together exploit* low locality code*. Single-threaded applications can use the entire network of cores while multi-threaded applications can efficiently share the resources. The sizing of critical structures remains small enough to handle current power envelopes.*

In single-threaded mode this processor is able to outperform previous state-of-the-art high-performance processor research by 12% on SpecFP. We show how in a quad-threaded/quad-core environment the processor outperforms a statically allocated configuration in both throughput and harmonic mean, two commonly used metrics to evaluate SMT performance, by around 2-4%. This is achieved while using a very simple sharing algorithm.

1 Introduction

Recent years have seen a new trend in the design of high performance microprocessors. Rather than continuing to improve performance through exploiting instruction-level parallelism (ILP), processors have begun to improve performance through thread-level parallelism (TLP). The shift in focus is driven by three factors limiting ILP-alone designs: the wide disparity between processor speeds and memory speeds (i.e. the *memory wall* [32]), and the increasing power budgets and design complexity of large monolithic designs. By contrast, *multi-core processors* take advantage of increasing transistor budget and can achieve high performance by running multiple threads simultaneously. For thread-parallel applications, the advantages of multi-core are obvious. However, by focusing on TLP, multi-core processors sacrifice performance for applications with a large sequential component. Despite the best efforts of the programming languages community, exploiting large numbers of threads for high performance is still a complex resource that most programmers do not know how to handle correctly.

We propose a microarchitecture capable of running a single thread or many threads with high performance and fairness. The architecture is based on a novel processor microarchitecture that allows the instruction window size to be changed at runtime. This is achieved by distributing the work among multiple small cores that can be reallocated

1089-795X/07 $25.00 © 2007 IEEE
DOI 10.1109/PACT.2007.28

IEEE computer society

to different threads. These processing elements are allocated to threads based on *execution locality*, i.e. the tendency of groups of instructions to have either high or low latency due to pending main memory accesses. Our microarchitecture exploits ILP by having an effective instruction window of thousands of instructions spread across the processing elements, largely overcoming the negative effects of long-latency memory operations. Our microarchitecture also exploits TLP for parallel workloads by allowing multiple threads to automatically allocate the processing elements it needs to achieve the best performance, rather than giving each thread the same kind of core regardless of its needs. As an additional benefit, all of these advantages are obtained without changes to the ISA, compiler or operating system.

Our variable window processor is based on a recent approach that has been proposed to build large instruction window processors [23]. It consists in using two processors to handle the instruction stream. A first processor, the *cache processor*, focuses on instructions whose inputs are available from registers or caches. A second, much simpler processor, the *memory processor*, focuses on main memory dependent instructions. The instruction window can be large, but the issue windows are small since both execution loops handle a relatively small amount of instructions. However, the design has several shortcomings. For instance, the intermediate buffer, being in-order, serializes all memory-dependent instructions regardless of their effective wakeup time. As will be shown, the penalty due to this serialization is significant, resulting in about a 10% performance loss. In addition, this design features only two execution modes, small window or full window, instead of offering a scalable performance range. This makes it undesirable as a building block for our flexible chip multiprocessor.

In this research, we borrow the decoupled nature of this approach but overcome its limitations and allow it to scale to many cores and many threads. The result is a processor with a variable window/issue size using a simple scalability mechanism. This variable-size window processor uses multiple small cores, called *memory engines*, linked by a network, to compute memory dependent instructions. The network introduces latency, but this additional latency has little impact on instructions already waiting hundreds of cycles due to a cache miss. The memory engine network can then be shared among threads to build a reconfigurable heterogeneous multi-core architecture.

The three main contributions of this paper are as follows:

1. We propose a new microarchitecture that significantly improves performance by overcoming memory latencies while keeping complexity within reasonable limits.
2. We propose a scalable microarchitecture with a variable window size that can be tuned by adding/removing memory engines.
3. We propose a multi-threaded implementation of the microarchitecture that can reconfigure itself resulting in the first heterogeneous multi-core architecture that adapts dynamically to the requirements of the threads. Most notable, reconfiguration happens dynamically without intervention of the operating system nor the programmer.

Through the rest of the paper we will describe the microarchitecture of our multi-core approach. This paper is organized as follows: Section 2 gives related work, Section 3 describes our proposal for a variable window single-threaded processor and Section 4 introduces an extension to run multiple threads efficiently on our microarchitecture. We conclude with Section 5.

2 Related Work

Much research has focused on designing processors capable of overcoming the memory wall [32]. Processor behavior in the event of L2 cache misses has been studied in detail [13]. Karkhanis *et al.* showed that many independent instructions can be fetched and executed in the shadow of a cache miss. This observation has fueled the proposal of microarchitectures supporting thousands of in-flight instructions.

One way to increase the number of instructions in flight is to scale existing processor structures. Work in this field has concentrated on the reorder buffer [7, 1], the instruction queues [7, 16, 30], on handling registers [19, 6] and on the load/store queue. Load queue resource requirements can be greatly reduced by using techniques for early release of load instructions [8, 18]. For the store queue, several recent proposals for dealing with scalability problems exist including two–level queue proposals [1, 21] or filtering proposals based on address hashing [26]. Another approach is to combine aggressive load optimizations with re-execution of load instructions at commit, using re-execution to verify the correctness of the optimizations [4, 24].

Other approaches to address the memory wall look ahead to find independent cache misses and thus provide an accurate prefetch stream. *Assisted threads* [29, 5, 25] rely on pre–executing future parts of the program, selected at compile time or generated dynamically at run time. *Runahead Execution* [9, 20] pre–executes future instructions while an L2 cache miss blocks the ROB. *Dual Core Execution* [33] is a technique that resembles runahead in that it can execute in advance to prefetch and improve branch prediction. It uses two cores: one that performs a runahead and another that is conservative. Because of the use of two cores, it does not suffer from refetching. However, it cannot achieve the

performance of a large window because it doesn't exploit distant parallelism. All instructions in the first core are processed again by the second core.

A different way to improve performance is to partition programs into tasks that can be executed in parallel. MultiScalar performs such a partitioning of the program and then executes the tasks speculatively on a set of processing units [28]. Our FMC architecture might seem similar to MultiScalar externally, but it performs the partitioning dynamically (based on execution locality) instead of being generated by the compiler. In MultiScalar, speed-ups are obtained by the parallel execution of tasks. Instead, in FMC, speed-ups are obtained by executing distant parallelism and by looking ahead in the program to execute loads early.

Recently multi-core architectures have become popular. In this paper we study a hybrid of a large window processor and multi-core architectures. Integrating multiple cores on a die presents some important issues such as how to interconnect multiple cores. The implications of doing this when using shared buses as interconnect have been studied in [15].

The popularity of multi-cores raises the problem of how to partition programs among cores. Traditional multicore approaches are homogeneous, i.e., all the cores look the same, while some proposals advocate using heterogeneous cores [14]. Both scenarios have difficulty accommodating many sorts of workloads. A recent proposal [11] addresses this problem by joining multiple 2-wide processors into clustered processors of widths 4, 6 and 8. This proposal differs from ours in two basic ways: First, it widens the processor width instead of the instruction window and, second, reconfiguration is triggered by a system call to the operating system whereas FMC performs this transparently to the programmer.

3 The multi-scan ILP processor

This section discusses current issues in ILP processor research, describes the baseline ILP processor, and then introduces our new proposed microarchitecture.

3.1 Recent Trends in ILP Processors

Out-of-order (OoO) execution [27] helps mitigate the effects of long-latency instructions and first-level cache misses but is less effective with today's high memory latencies. Figure 1 shows the performance loss in percentage from the fetch bandwidth for a set of 4-way fetch/commit OoO processor configurations ranging from perfect caches to real memory subsystems with hundreds of cycles of latency running SPEC CPU 2000 FP. Scaling processor resources to provide a larger instruction window can theoretically overcome this limitation. However, this scaling is infeasible in practice due to technology constraints.

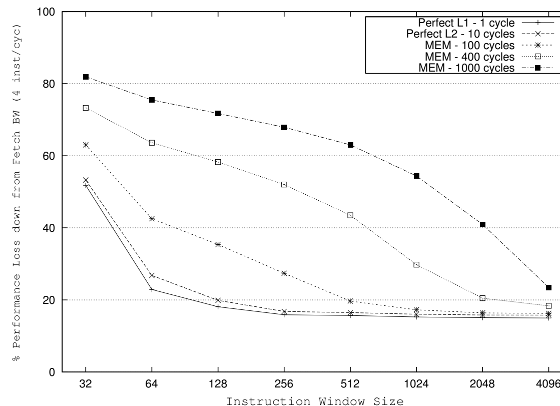


Figure 1. Performance Loss in Spec2KFP as a function of memory subsystem and instruction window size

Recent research has identified two key factors in the design of processors capable of overcoming the memory wall:

Execution Locality: This is the tendency of instruction slices to execute in bursts separated by memory accesses. Instructions linked only through register/cache accesses are said to have *high execution locality*. Otherwise they have *low execution locality*. This observation enables the construction of large-window processors requiring only moderately-sized structures by focusing only on the execution of high locality code [7, 16, 30, 23].

Non-Blocking Front-End: Running ahead and executing loads while avoiding ROB stalls increases Memory-Level Parallelism (MLP) [10], minimizing the impact of any single high-latency memory access and thus increasing overall throughput in memory-limited applications [30, 23, 20].

Execution Locality is a concept that describes a property of instruction execution. We can divide the execution of a program into dynamic instructions that are classified as either short or long latency. Short latency instructions are executed quickly e.g. within tens of cycles. They depend only on the results from other short latency instructions and loads hitting in the cache. Long latency instructions depend on loads that access main memory. Clusters of instructions with short latencies are joined to one another by long latency instructions. Within a cluster, we say instructions have *high execution locality*.

The key observation of execution locality is that, since instruction clusters suffer penalties no larger than L2 cache accesses, they can be executed using only moderately sized

structures. Processor design can be approached as a combination of a processor with idealized cache and some support structures to hide main memory access latencies.

3.2 Our Baseline: The D-KIP

The Decoupled KILO-Instruction Processor (D-KIP) [23] design will be our starting point for building a context for this research. It is typical of state of the art proposals for exploiting ILP through a large effective instruction window. In the D-KIP, two cores cooperate to execute an application. The first core, the Cache Processor (CP), is small and fast, and executes all code depending only on cache accesses (*high locality code*). The CP runs forward as fast as possible, launching all loads with known addresses. Code that depends on memory accesses (i.e., *low locality code*) is processed by a secondary core, the Memory Processor (MP), which is proposed as a small in-order processor. It executes low-locality code fetched from an in-order Low Locality Instruction Buffer (LLIB) that has previously been filled in program order by the CP. Processor recovery is ensured by using checkpoints that are created dynamically at the reorder buffer (ROB) output of the CP. Figure 2 shows a simplified overview of the D-KIP processor. The key insight is that the LLIB is capable of representing a large instruction window but is implemented as a simple RAM, not a CAM, thus avoiding the problems of latency and power inherent in a naive scaling of an out-of-order processor.

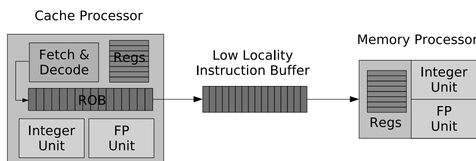


Figure 2. Microarchitecture of the D-KIP Processor

The D-KIP does not handle well the case when the LLIB contains a code mix with different localities, i.e. if there are pending cache misses yet to execute in the LLIB code. The LLIB, being in-order, prevents data-flow execution of these different locality instruction clusters, limiting performance. In addition, the implementation of a centralized checkpointing stack raises some concerns regarding design complexity and movement of register values around the chip.

3.3 Approximating Dataflow in the Memory Processor

The problems of in-order execution in the Memory Processor can be solved by introducing a relaxed form of out-of-order execution that we term Multi-Scan Execution, explained next. Full out-of-order capabilities are not necessary since small latencies are easily hidden compared to the latencies resulting from cache misses.

The Memory Processor has an interesting property that enables a different execution paradigm. Contrary to what happens in a classic processor, the memory processor does not perform fetch nor path discovery. Instead, instructions are provided by an external entity (the Cache Processor) which injects the instructions into the Memory Processor. When the long-latency load that triggered the MP finally returns from memory, the MP is in a situation where hundreds of pre-decoded instructions are possibly buffered awaiting execution. At this point, the instructions in the low-locality instruction buffer are long latency instructions that have not yet executed. The ordering of these instructions is the program ordering. All higher locality instructions in between are already executed. Thus, available inputs need to be recorded together with the pending instructions in the buffer. If we also insert executed loads and stores into the LLIB, the result is a compressed program image that does not include the higher locality instructions. The problem then is how to efficiently execute these instructions. The inclusion of all loads and stores is not necessary in principle for correction, but it will simplify the implementation of our architecture, as will be seen later.

The method that we devise for efficient coarse-grained dataflow execution is based on the concept of execution locality presented earlier. The idea is to take the LLIB and perform multiple runs through the buffer, each one focusing on a deeper execution locality level. In the first pass, instructions depending directly or indirectly on one cache miss are executed. In the second, the instructions depending on two chained cache misses are executed. In the third, those depending on three chained cache misses. This procedure continues until no more instructions need to be executed. At this point, the whole group of instructions can be committed. This includes sending all stores to the data cache. This is one of the reasons to include all stores (including executed stores) in the low-locality instruction stream.

The new Memory Processor is a considerable departure from the D-KIP. However, the concepts on which it is built are more straightforward. In addition to the instruction buffer, which we now call the *Completion Buffer*, the processor includes integer and FP functional units, a store buffer and a pair of register files associated to the head and tail of the Completion Buffer to keep the precise state. Furthermore, it keeps a small local register file to keep the partial register state necessary to process the instructions. Contrary to the D-KIP, this scheme does not require the existence of a checkpointing stack. We call this small processor a *memory engine* since it processes instructions that depend on memory accesses. *Multi-Scan* execution vaguely resem-

bles *Flea-Flicker Multipass Pipelining* [3]. However, while *flea-flicker* is a technique specifically targeted at overcoming L1 cache misses in an in-order processor, *multi-scan execution* is a way to achieve coarse grain dataflow execution in a set of high-latency instructions.

A diagram of the internal architecture of the memory engine can be seen in figure 4 (a). The output register file will initially be empty. After each scan new output registers will have been computed. The memory engine then proceeds to update these registers in the output RF. The input RF has always all registers, since it represents the register view when the first long-latency load is detected. At this point instructions start being inserted into the instruction buffer. Until now the commit register file of the Cache Processor had a precise view of the register set. This precise view is then copied into the Memory Engine.

The main problem with this scheme is that completing a scan requires the memory engine to be filled completely. The memory engine cannot start processing instructions of a lower locality earlier because newer instructions of a higher locality may still be inserted. In addition, to allow effective lookahead this scheme will require a very large completion buffer. But the larger this buffer, the worse the dataflow approximation, as memory access latencies are not uniform. An effective scheme should be able to handle lower locality instructions earlier.

3.4 Obtaining a Resizable Window with a Set of Memory Engines

The aforementioned problem is a direct consequence of having a single buffer. A simple solution to this problem is to partition the buffer into multiple sequential smaller buffers and provide each one with its own set of functional units. The buffers are then allocated round-robin to the Cache Processor as they are needed. In this scheme, instead of having a single memory engine we have a Memory Processor consisting of a set of Memory Engines. Each of these memory engines is a replica of the multi-scan engine handling a subset of the compressed instruction window. Registers are passed between the engines using the input and output register files. After each *scan* the engine checks the output register file for newly generated registers (*live-outs*). These registers are then sent over a network to the input register file of the next logical memory engine. The input register files also provide the points where the processor can perform precise recovery. If an exception occurs, e.g. a branch misprediction, only engines handling younger instructions are squashed and the contents of the input RF of the engine where the exception happened are then copied to the Cache Processor register file so that execution can resume. Uncomputed registers are marked as long-latency so that depending instructions still travel to the Memory Processor. A diagram comparing the instruction windows of several architectures can be seen in Figure 3.

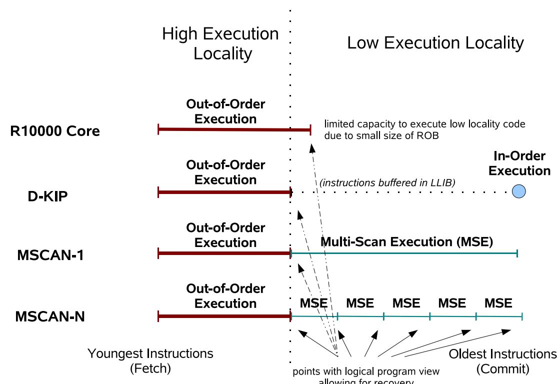


Figure 3. A comparison of Instruction Windows of the R10000, the D-KIP, and a single- and multiple-ME design with multi-scan execution

There are many parameters to tune in this microarchitecture. After a design space exploration we have settled on a memory engine design with in-order instruction queues and which scans two instructions per cycle. Each Memory Engine can handle up to 128 execution-pending instructions and up to 128 loads and stores. Reducing the issue width to one reduced IPC in 2.3% while implementing out-of-order instruction queues increased performance in about 0.1%. In-order performs well since each scan only processes high-locality instructions for which out-of-order execution is not necessary.

The interconnection of the memory engines is another critical issue. The memory engines require a path to all other engines. Communication itself happens only between conceptually adjacent engines. However, the *previous* and *next* engine could be located anywhere. To provide this all-to-all communication some sort of network needs to be provided. We will analyze the trade-offs of such a network later. A diagram of the complete architecture, showing the MEs connected with a mesh network, can be seen in Figure 4(b). The figure also highlights the different paths that are necessary for communication: memory access, register transfer and instruction insertion.

3.4.1 Memory Management

Memory management is a critical component in high-performance architectures. Much research is going on at this moment in order to build a scalable low-complexity load/store queue. A great deal of this effort can be reused for the multi-scan design. One option is to combine a hierarchical store queue [1] and a non-associative load queue

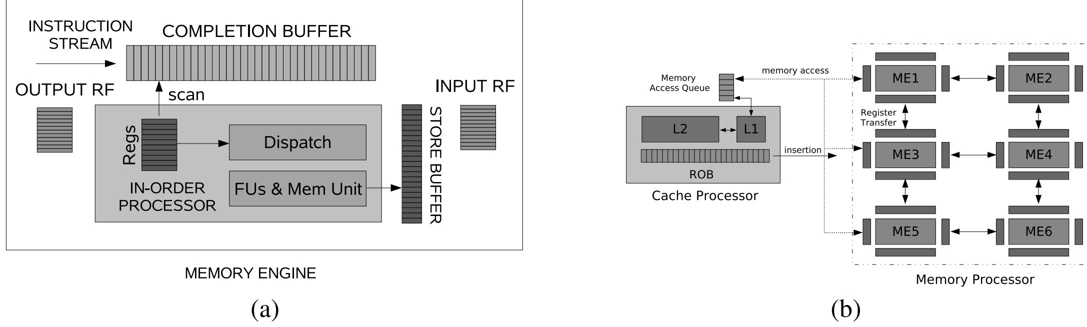


Figure 4. (a) Architecture of a single Memory Engine and (b) Generalized Processor with ME Network

using re-execution [24, 4] to handle loads and stores. As can be seen in Figure 4 (b), access the LSQ is still centralized in the Cache Processor. Our scheme is thus compatible with traditional memory consistency models for multiprocessors.

The architecture presented so far will be the basic building block for our goal of building a chip multiprocessor that can dynamically reconfigure itself to support various degrees of heterogeneity. Before proceeding with the description of the CMP architecture we evaluate the proposed architecture in single-threaded mode.

3.5 Evaluation of the multiscan processor

The microarchitecture presented so far has been evaluated using an execution-driven simulator with the Alpha ISA. We used the SPEC CPU 2000 benchmark suite with selected simulation points of 100 million instructions using a methodology based on SimPoint [22].

We will determine the equivalent window size of the FMC processor and compare its performance with other proposals. We will also analyze the important topic of memory bandwidth which will be one of the major bottlenecks in future CMP processors.

In the following we list the evaluated microarchitectures:

R10-64: A 4-way R10k-like processor with out-of-order scheduling logic and a 64-entry ROB. The integer and FP instruction queues have 40 entries. Other resources are idealized. *R10-64-PREF* is the same architecture extended with an aggressive stream prefetcher that can hold up to 256 streams of 256 bytes, totaling 64KB of prefetched data.

R10-256: Like the previous, but with a 256-entry ROB. The instruction queues can hold up to 160 instructions each. This model is much more aggressive than current microarchitectures. *R10-256-PREF* is the same processor but including the aforementioned stream prefetcher.

RA-64, RA-256: Two runahead processors [20] with 64/256-entry ROBs. *RA-64-PREF, RA-256-PREF* are the same models but including the stream prefetcher. These models include an unrestricted fully associative runahead cache which allows them to take full advantage of data forwarding during runahead.

DKIP: This is the D-KIP model as presented in [23]. The size of each LLIB is 2048 entries. When using the prefetcher we call this processor *DKIP-PREF*.

FMC: This is the proposed microarchitecture. It includes 16 memory engines. All transfers to, from and within the MP suffer an additional delay of 4 cycles representing network latency. The Cache Processor is equivalent to R10-64 in terms of structure sizes. *FMC-PREF* includes the stream prefetcher. FMC stands for *Flexible Multi-Core Architecture* and is the name we will use to refer to the proposed microarchitecture.

The Load/Store Queue has been idealized for all models. The memory model is modeled after an idealized pipelined memory that is capable of transferring 8 bytes every 4 processor cycles. Table 1 lists parameters that are equal for all configurations. Note how the FMC architecture is built completely out of small-sized structures.

Fetch/Decode Bandwidth	4
Branch Predictor	Perceptron [12]
Store/Load ports	2 shared ports
L1 Cache Size, Associativity & Access Latency	32 KB / direct mapped / 1 cycle
L2 Cache Size, Associativity & Access Latency	2 MB / 4-way set assoc / 10 cycles
Memory Latency	400 cycles
Cache Processor & R10k: IQ/FPQ/ROB/RegFile	40/40/64/96
Cache Processor & R10k: Scheduler	Out-of-Order
Memory Processor: IQ/FPQ/RegFile	20/20/32
Memory Processor: Scheduler	In-Order

Table 1. Common Parameters for all Microarchitectures

Figure 5 shows the IPC for selected microarchitectures side-by-side. The FMC performance gets close to the limit shown in Figure 1. Using 16 Memory Engines the IPC for SPEC FP reaches 2.97 using no prefetcher. There is a considerable speed-up of 12% compared to the D-KIP architecture, which reaches 2.66, and a 31% speed-up compared to RA-256. When prefetchers are in use, the speed-ups

are 5% compared to the DKIP and 18% compared to runahead. FMC outperforms runahead because it does not need to refetch instructions executed under the shadow of a cache miss and therefore obtains a stronger memory lookahead effect.

On FP codes, the FMC architecture achieves speed-ups of 53% and 90% over the R10K-256 and R10K-64 (not shown) microarchitectures, respectively. These microarchitectures are severely limited by the sizes of their ROBs and cannot overcome memory stalls.

The speed-up achieved with integer codes is not as large: up to 13% for R10K-64-PREF. The FMC sees a speed-up of 9% compared to the more aggressive R10-256 model. There are no notable differences with runahead and the D-KIP models. All these techniques hit a wall due to the frequent recoveries caused by branch mispredictions and the lack of memory level parallelism in many integer applications. In addition, the large second level cache is large enough to capture the locality of most SPEC INT benchmarks. Thus, trying to overcome the memory wall with special techniques is unlikely to give major speed-ups unless the other problems are attacked first.

Figure 5 also shows the percent increase in the off-chip memory traffic that the prefetcher is generating. Although the prefetching approaches provide good speed-ups, they do this at the cost of considerable traffic increase. In the case of FMC, the addition of the prefetcher does not improve performance for FP codes. The reason is that the window size achieved by FMC is large enough to make it insensitive to parallelizable memory accesses. The fact that FMC does not require a prefetcher at all has important benefits. In addition to the reduction of memory traffic the FMC benefits from less area, complexity and power.

3.5.1 Allocation and Efficiency of the Memory Engines

The evaluation so far has been conducted using a FMC processor with 16 memory engines allowing us to emulate a core with a window of around 1500 instructions (see figure 1). This is enough for a 400-cycle latency in most benchmarks. To analyze the effective requirements we have evaluated the average performance of the FMC processor using different numbers of memory engines, ranging from 0 to 16. The resulting IPC curve for both SPEC INT and SPEC FP can be seen in Figure 6.

The figure shows the progression of IPC starting from 0 memory engines, which is equivalent to the R10K-64 processor, up to 16 memory engines. The IPC value at this point differs in less than 1% from the value achieved with 30 MEs. Using 8 memory engines is still enough to achieve 95.7% of the maximum IPC for SPEC FP. The curve for SPEC INT saturates earlier, reaching 96.8% of the final IPC with only 4 memory engines. The architecture of the FMC is therefore well suited for power-performance trade-offs. If we want to reduce power consumption, we can deactivate memory engines and make the architecture smaller.

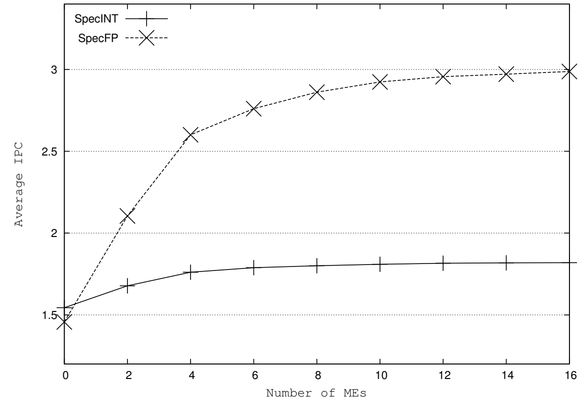


Figure 6. SPEC CPU 2000 IPC for varying number of Memory Engines. The case of zero engines is equivalent to R10k-64

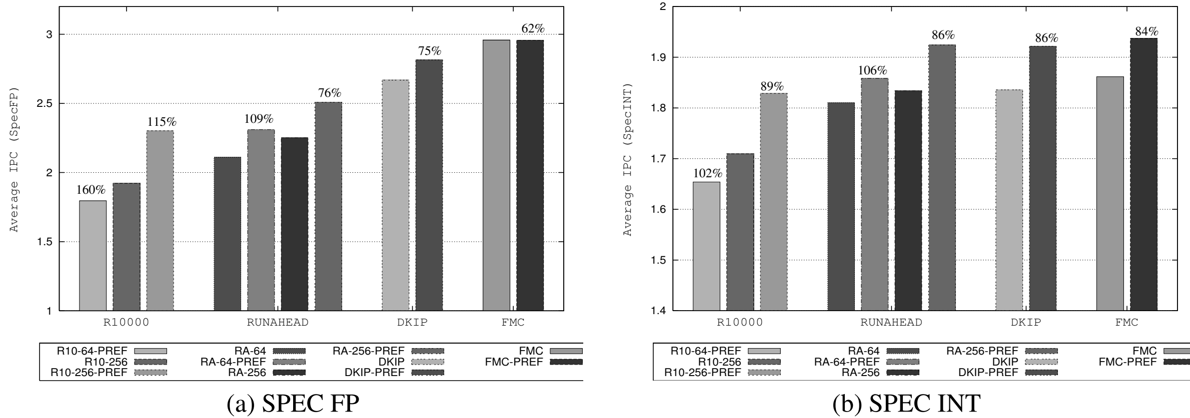
To characterize the behavior of the applications we measure two parameters:

- the average allocation of memory engines when run with 16 memory engines; and
- the minimum number of memory engines required to reach 95% of the performance of a FMC with 16 memory engines.

The results for SPEC INT and SPEC FP are listed in Table 2. These numbers allow establishing a classification of applications depending on the speed-up they experience when they can use memory engines and the average number of engines that they allocate:

High Average Allocation, High Speed-Up (Type A): This includes applications that experiment large speed-ups when additional MEs are given to them. The additional MEs allow these applications to extract more MLP and execute more distant parallelism. The benchmarks in this category are: *ammp, applu, apsi, art, equake, fma3d, lucas, mcf, sixtrack, vortex* and *wupwise*.

High Average Allocation, Low Speed-Up (Type B): This includes applications that consume many MEs but do not noticeably improve IPC in the process. The reason is that these applications have not enough MLP to exploit and instead perform sequential memory accesses. The benchmarks in this category are: *bzip2, facerec, gcc, parser* and *perlbmk*.



(a) SPEC FP (b) SPEC INT

Figure 5. IPCs for selected MicroArchitectures for (a) SPEC FP and (b) SPEC INT

Benchmark	Average Allocation	MEs for 95% IPC	Benchmark	Average Allocation	MEs for 95% IPC
bzip2	6.19	0	ammp	4.72	8
crafty	0.28	2	applu	9.83	6
eon	0.02	0	apsi	6.51	8
gap	2.08	4	art	8.77	6
gcc	4.32	2	equake	9.32	12
gzip	0.61	0	facerec	7.51	4
mcf	5.41	8	fma3d	14.16	10
parser	8.78	2	galgel	0.72	2
perlbmk	4.79	2	lucas	7.65	8
twolf	0.30	0	mesa	0.91	2
vortex	6.24	8	mgrid	3.47	4
vpr	3.0	4	sixtrack	1.94	6
			swim	3.29	4
			wupwise	3.13	6

Table 2. Behavior of SPEC INT (left) and SPEC FP (right) applications

Low Average Allocation (Type C): This includes the remaining apps that do not allocate many MEs to reach their maximum speed-ups. The reason is that the working set of these benchmarks fits nicely within a 2MB L2 cache. The benchmarks in this category are: *crafty, eon, galgel, gap, gzip, mesa, mgrid, swim, twolf* and *vpr*.

3.5.2 Latency tolerance of the ME Network

Given that we use a network to connect the memory engines we cannot assume that operations requiring the use of the network will be able to complete in a single cycle. The impact of these delays needs to be evaluated in detail. There are three cases in which data needs to be passed through the ME network: insertion of instructions, register transfer and load execution (*cache access*).

Instead of modeling a specific type of network we have opted to add a fixed penalty to transfers over the network. The mesh shown in figure 4 (b) is only one example of a network that could be used. There is nothing that precludes the use of a bus, a token ring or a different type of network. Using a fixed delay makes our evaluation independent of the network architecture and enables an easier comparison with other architectures. A fixed delay can be considered an average latency, but it is also what one would find in a butterfly/Clos network. Figure 7 shows the impact of the network delays in the performance of the system both for SPEC INT and SPEC FP.

Results show that even with an 4-cycle additional one-way latency (8-cycle round trip for cache accesses), performance is still around 1% of the maximum for both SPEC FP and and SPEC INT. The main contribution of this performance drop is the delay suffered by loads. Evaluating only register transfer and instruction insertion delays results in no appreciable performance penalty at all.

The 4-cycle extra latency implies a 8-cycle delay in both directions (from ME to LSQ and back) giving total access latencies of 9 cycles for L1 and 18 cycles for L2. For a 8-cycle access latency (16 cycles round-trip) performance

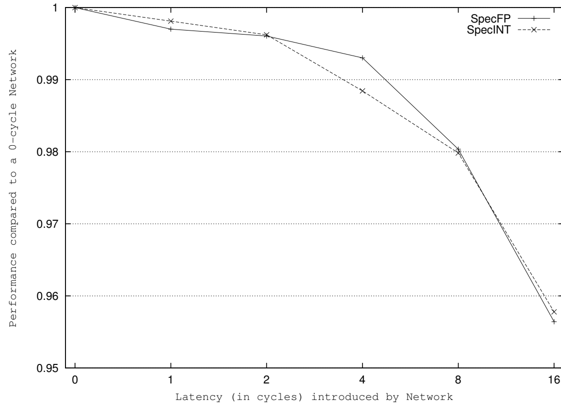


Figure 7. Effect of network delays in the MP on the final performance

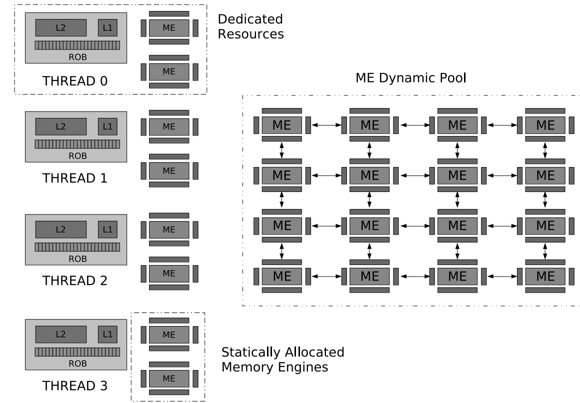


Figure 8. The microarchitecture of the flexible multi-core microarchitecture, including a set of Cache Processors, 2 statically assigned ME per thread, and a dynamic pool of memory engines

degrades about 2%. The tolerance of the Memory Processor to additional latencies is what allows these large delays to result in relatively small performance overheads. For the rest of this paper we are going to continue using an additional network latency of 4 cycles.

4 A flexible heterogeneous multi-core architecture

The fundamental property of the FMC is its ability to change the instruction window size at runtime. It can do so by dynamically adding or removing memory engines from the system. This property allows the processor to adapt to the requirements of the application and activate only those memory engines that are predicted to lead to improved performance.

We propose to use the network of Memory Engines to construct a dynamically adapting multi-core architecture that can provide high throughput to sets of threads with low requirements, mixes of applications with different resource requirements and mixes of identical high-performance programs, particularly in the case that less threads than CPs are running.

Our proposed flexible multi-core architecture consists of a set of Cache Processors, each one with a static partition of memory engines, and a pool of memory engines that can be dynamically assigned to the different threads. Figure 8 shows a general view of this microarchitecture.

The microarchitecture has good potential to adapt to application mixes as threads that do not require Memory Engines can relinquish their engines and give them to threads that require more memory engines. Moreover, when there are fewer threads than Cache Processors, those threads that are running can access the dynamic pool of memory engines without competition.

4.1 Assigning Memory Engines

For the flexible multi-core architecture we developed an algorithm for assigning Memory Engines with the goals of simplicity and reasonable performance. The algorithm works as follows: Every fixed number of cycles (we arbitrarily choose 256 cycles) a piece of logic, called the *arbiter*, collects information from the Cache Processors regarding the number of dynamic memory engines that a thread has allocated but is not using. The arbiter adds all unused engines to a common pool and reassigns the free engines to the cache processors, one at a time, using a round-robin policy starting with the thread that currently has the smallest number of MEs allocated. At all times each engine is allocated to some thread, although the engine might be in a power-saving mode. It is the responsibility of the CP to activate an engine when the application is going to use it.

4.2 Multi-core Simulation Infrastructure

The multi-core implementation that we have proposed so far is highly decoupled. There are only three elements that are shared: the dynamic pool of MEs, the system bus and the main memory. Everything else is local to the thread. This includes the two levels of cache, TLBs and functional units. This partitioning has been implemented on some commercial processors such as the Intel Montecito, Intel PentiumD or AMD AthlonX2 processors. The sharing of the memory engines has been modeled by implementing a

server process acting as the arbiter. The cache processors are clients to this process. Every 256 cycles they send a packet with the number of free engines to the server and receive a new allocation as an answer. Sharing of the system bus has been modeled by implementing a virtual memory system that statically partitions the bus bandwidth among the threads. In our model, each thread gets the same bandwidth. This assumption is pessimistic as a memory controller could perform a much better bus cycle assignation between the threads.

Evaluating multi-core/multi-threaded architectures requires the generation of workload mixes for the simulations. We evaluate a multi-core architecture with 4 Cache Processors. To generate the workload mixes we use the application classification provided at the end of Section 3.5. In constructing the workloads, we order the benchmarks alphabetically and choose them using a round robin algorithm[1]. Table 3 shows the generated workload mixes for the architecture with 4 Cache Processors. In Table 3, 'r' means *repeated*, i.e. the same application is run multiple times in parallel, a frequent scenario in scientific/engineering computations and also usual in server workloads.

Running multi-threaded simulations has special requirements as we cannot simply run 100 million instructions and stop. Different benchmarks take different amounts of time to execute and stopping in the middle of a simpoint distorts the results. To avoid this situation we use the methodology proposed in [31]. The idea behind this methodology is to re-execute the benchmarks in a workload as many times as needed until the measurements obtained (IPC in our case) are representative. In this paper we used a Maximum Allowable IPC Variance (MAIV) of 5%. The number of memory engines has been fixed to a total of 20. This number includes statically allocated engines and the dynamic pool.

4.3 Evaluation of a 4-way Multi-Core Architecture

The goal of this study is to check the effect of dynamically sharing the MEs. To this end, we compare a configuration in which each thread has 5 statically assigned MEs and no dynamic sharing (S5D0) versus a configuration where each thread has 2 statically assigned engines and there is a pool of 12 ME to share (S2D12). The S5D0 configuration models a symmetric CMP. Such a model focuses on throughput with a special emphasis on fairness. We are interested in analyzing if the dynamically reconfiguring S2D12 is capable of exceeding the homogeneous S5D0 both in throughput *and* fairness. Note that we do not evaluate an asymmetric CMP configuration. For an analysis of asymmetric architectures we refer the reader to [2].

The throughput results for the 4-way Multi-Core are shown in Figure 9. The model of the FMC architecture that was used does not include a prefetcher. The workload identifiers have been abbreviated for spatial reasons. AA workloads refer to {A,A,A,A}, AB workloads to {A,A,B,B} and AC workloads refer to {A,A,C,C}. Finally, workloads where a benchmark is run multiple times in parallel are identified as *benchmark* $\times$ *y*. In this case *y* identifies the number of parallel occurrences of the benchmark. Because each benchmark advances at the same speed, these repetition workloads have been simulated with different fast forwards. The fast forwards differ in 10 million instructions and they average to the same fast forward used in the single thread evaluations in this paper. While running applications with a difference of 10 million instructions may not allow testing the assignation algorithm for different program phases, it will allow to see how the algorithm behaves due to small variations resulting from operating system scheduler decisions.

While using more memory engines improves IPC considerably for class-A applications, there is always a point of saturation independent of the benchmark. Many applications will try to go past their saturation point and consume more memory engines than necessary. To handle this particular case of unfair behavior we have limited the maximum amount of dynamic memory engines that the arbiter will assign to a single thread to eight engines. This number represents two thirds of the size of the dynamic pool and is applied to all applications, irrespective of their type. Note that this limitation is only enforced when 4 threads are running. For the cases of two threads and one thread it is not meaningful.

The {A,A,A,A}, {A,A,B,B} and {A,A,C,C} mixes show promising results when run with a shared pool of memory engines. For this particular configuration {A,A,A,A} workloads experienced a 1.9% speed-up in throughput, AB workloads experienced a 0.4% speed-up and {A,A,C,C} saw a 3.9% improvement when running on the S2D12 configuration. We also measured the harmonic mean of workloads and found that the dynamic assignation algorithm improves its value between 2-4%. We used the *harmonic mean* defined as the mathematical harmonic mean of the relative IPCs compared to the case when the thread is running alone, i.e. when there is no competition from other threads [17]. Thus our technique not only improves throughput, but it also provides a fair execution of all threads in a workload.

When running repeated workloads the situation improves even more, particularly when fewer threads than the number of Cache Processors are running. For example, if only a single copy of *applu* is running, the throughput of this benchmark is 12.5% higher on the the S2D12 configu-

[1]That is, we have chosen the mixture of benchmarks using a fixed procedure before performing the experiments and without regard to the results achieved with a particular mixture.

Class Mix	Benchmark Combinations
{A,A,A,A}	{ammp, applu, apsi, art} {equake, facerec, fma3d, lucas}
{A,A,B,B}	{mcf, vpr, bzip2, gcc} {ammp, applu, parser, perlbmk}
{A,A,C,C}	{apsi, art, crafty, eon} {equake, facerec, galgel, gap} {fma3d, lucas, gzip, swim} {mcf, vpr, mesa, mgrid}
{A,r,r,r}	{ammp, ammp, ammp, ammp} {applu, applu, applu, applu} {apsi, apsi, apsi, apsi} {art, art, art, art}
{A,r,-,-}	{ammp, ammp, -, -} {applu, applu, -, -} {apsi, apsi, -, -} {art, art, -, -}
{A,-,-,-}	{ammp, -, -, -} {applu, -, -, -} {apsi, -, -, -} {art, -, -, -}

Table 3. Workload mixes for 4-way Multi-Core

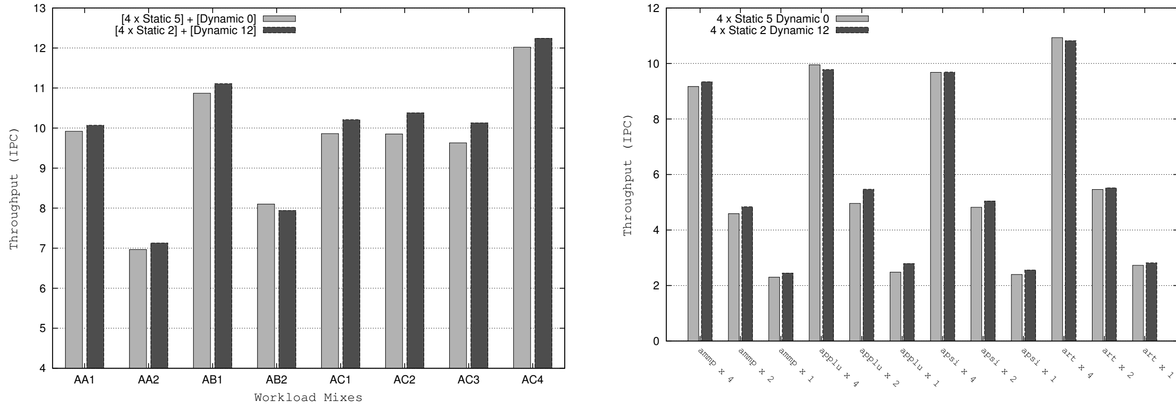


Figure 9. Throughput of the Mixed Workloads (left) and the Repetition Workloads (right) in the 4-Core Implementation

ration compared to the S5D0 configuration. This is because under these circumstances the arbiter can assign up to 14 memory engines to a single application without having to compete for resources with other threads. This is far better than what can be obtained using a homogeneous multi-core or even a heterogeneous multi-core, as none of these architectures are able to reassign all hardware resources, a limitation from which the FMC architecture does not suffer. Only the small subset of statically assigned engines is wasted in the FMC. In addition, the FMC architecture performs all these reconfigurations dynamically and can thus adapt to variations in program behavior.

5 Conclusions

We have presented a flexible multi-core microarchitecture capable of running one or many threads with high performance. Each core is very simple, thus our approach scales to a large number of cores, allowing for an efficient and simple design.

For floating point applications, we have shown that our design improves performance by 53% over a next-generation superscalar processor and 12% over recent previous work in large instruction window designs. With integer applications there is a more modest but still quite significant speed-up of 9% over the out-of-order processor with a 256-entry instruction window. Moreover, our design allows multiple threads to use as many or as few resources as they need from a pool of available cores, rather than allocating a single thread per core as in previous multi-core designs. In a 4-core environment we find that our approach improves throughput and fairness on average around 2-4%. This result is encouraging given the simplicity of the arbiter scheme that was implemented. We believe this design is the right path to provide best performance for workloads consisting of a wide variety of applications, both single and multi-threaded.

Acknowlegements

This work has been supported by the Ministerio de Educación y Ciencia of Spain under contract TIN–2004–07739–C02–01 and the HiPEAC European Network of Excellence (Framework Programme IST–004408). Daniel A. Jiménez is supported by NSF Grant CCF–0545898.

References

[1] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. 2003.

[2] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The impact of performance asymmetry in emerging multicore architectures. In *Proc. of the Intl. Symp. on Computer Architecture*, pages 506–517, June 2005.

[3] R. D. Barnes, S. Ryoo, and W. mei W. Hwu. Flea-Flicker Multipass Pipelining: An Alternative to the High-Power Out-of-Order Offense. In *Proc. of the 38th. Annual Intl. Symp. on Microarchitecture*, December 2005.

[4] H. W. Cain and M. H. Lipasti. Memory ordering: A value-based approach. In *Proc. of the 31st Intl. Symp. on Computer Architecture*, June 2004.

[5] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt. Simultaneous subordinate microthreading (SSMT). In *Proc. of the 26th. Intl. Symp. on Computer Architecture*, 1999.

[6] A. Cristal, J. Martinez, J. LLosa, and M. Valero. Ephemeral registers with multicheckpointing. Technical report, 2003. Technical Report number UPC-DAC-2003-51, Departament d'Arquitectura de Computadors, Universitat Politecnica de Catalunya.

[7] A. Cristal, D. Ortega, J. Llosa, and M. Valero. Out-of-order commit processors. In *Proc. of the 10th Intl. Symp. on High-Performance Computer Architecture*, 2004.

[8] A. Cristal, M. Valero, A. Gonzalez, and J. LLosa. Large virtual ROBs by processor checkpointing. Technical report, 2002. Technical Report number UPC-DAC-2002-39.

[9] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proc. of the 11th Intl. Conf. on Supercomputing*, pages 68–75, 1997.

[10] A. Glew. MLP yes! ILP no! In *ASPLOS Wild and Crazy Idea Session '98*, 1998.

[11] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez. Accomodating workload diversity in chip multiprocessors via adaptive core fusion. In *Workshop on Complexity-Effective Design*, pages 10–21, June 2006.

[12] D. A. Jimenez and C. Lin. Dynamic branch prediction with perceptrons. In *Proc. of the 7th Intl. Symp. on High Performance Computer Architecture*, pages 197–206, January 2001.

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

[14] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-isa heterogeneous multicore architecture for multithreaded workload performance. In *Proc. of the 31st Intl. Symp. on Computer Architecture*, June 2004.

[15] R. Kumar, V. Zyuban, and D. M. Tullsen. Interconnection in multicore architectures: Understanding mechanisms, overheads, and scaling. In *Proc. of the 32nd Intl. Symp. on Computer Architecture*, June 2005.

[16] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A large, fast instruction window for tolerating cache misses. In *Proc. of the 29th Intl. Symp. on Computer Architecture*, 2002.

[17] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. In *Proc. of the 2001 IEEE Intl Symp on Performance Analysis of Systems and Software*, November 2001.

[18] J. Martinez, J. Renau, M. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed early Resource recycling in out-of-order Microprocessors. In *Proc. of the 35th Intl. Symp. on Microarchitecture*, pages 3–14, 2002.

[19] M. Moudgill, K. Pingali, and S. Vassiliadis. Register Renaming and Dynamic Speculation: an alternative approach. In *Proc. of the 26th. Intl. Symp. on Microarchitecture*, pages 202–213, 1993.

[20] 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 *Proc. of the 9th Intl. Symp. on High Performance Computer Architecture*, pages 129–140, 2003.

[21] I. Park, C. L. Ooi, and T. N. Vijaykumar. Reducing design complexity of the load/store queue. In *Proc. of the 36th Intl. Symp. on Microarchitecture*, 2003.

[22] E. Perelman, G. Hamerly, M. V. Biesbrouck, T. Sherwood, and B. Calder. Using SimPoint for accurate and efficient simulation. In *Proc. of the Intl. Conf. on Measurement and Modeling of Computer Systems*, June 2003.

[23] M. Pericàs, A. Cristal, R. Gonzalez, D. A. Jimenez, and M. Valero. A decoupled kilo-instruction processor. In *Proc. of the 12th Intl. Symp. on High Performance Computer Architecture*, February 2006.

[24] A. Roth. Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization. In *Proc. of the 32nd Intl. Symp. on Computer Architecture*, June 2005.

[25] A. Roth and G. Sohi. Speculative data-driven multithreading. In *Proc.of the 7th Intl. Symp. on High-Performance Computer Architecture*, 2001.

[26] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler. Scalable hardware memory disambiguation for high ILP processors. In *Proc. of the 36th Intl. Symp. on Microarchitecture*, 2003.

[27] J. E. Smith and A. R. Pleszkun. Implementation of Precise Interrupts in Pipelined Proccessors. *Proc. of the 12th Intl. Symp. on Computer Architecture*, pages 34–44, 1985.

[28] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proc. of the 22nd. Intl. Symp. on Computer Architecture*, June 1995.

[29] Y. H. Song and M. Dubois. Assisted execution. Technical report, 1998. Technical Report #CENG 98-25, Department of EE-Systems, University of Southern California.

[30] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual flow pipelines. In *Proc. of the 11th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2004.

[31] J. Vera, F. J. Cazorla, A. Pajuelo, O. J. Santana, E. Fernandez, and M. Valero. FAME: FAirly MEasuring Multithreaded Architectures. In *Proc. of the 16th Intl. Conf. on Parallel Architecture and Compilation Techniques*, September 2007.

[32] W. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, March 1995.

[33] H. Zhou. Dual-core execution: Building a highly scalable single-thread instruction window. In *Proc. of the 14th Intl. Conf. on Parallel Architectures and Compilation Techniques*, September 2005.