Disjoint Out-of-Order Execution Processor

MAGEDA SHARAFEDDINE, KOMAL JOTHI, and HAITHAM AKKARY,

American University of Beirut, Lebanon

High-performance superscalar architectures used to exploit instruction level parallelism in single-thread applications have become too complex and power hungry for the multicore processors era. We propose a new architecture that uses multiple small latency-tolerant out-of-order cores to improve single-thread performance. Improving single-thread performance with multiple small out-of-order cores allows designers to place more of these cores on the same die. Consequently, emerging highly parallel applications can take full advantage of the multicore parallel hardware without sacrificing performance of inherently serial and hard to parallelize applications. Our architecture combines speculative multithreading (SpMT) with checkpoint recovery and continual flow pipeline architectures. It splits single-thread program execution into disjoint control and data threads that execute concurrently on multiple cooperating small and latency-tolerant out-of-order cores. Hence we call this style of execution Disjoint Out-of-Order Execution (DOE). DOE uses latency tolerance to overcome performance issues of SpMT caused by interthread data dependences. To evaluate this architecture, we have developed a microarchitecture performance model of DOE based on PTLSim, a simulation infrastructure of the x86 instruction set architecture. We evaluate the potential performance of DOE processor architecture using a simple heuristic to fork control independent threads in hardware at the target addresses of future procedure return instructions. Using applications from SpecInt 2000, we study DOE under ideal as well as realistic architectural constraints. We discuss the performance impact of key DOE architecture and application variables such as number of cores, interthread data dependences, intercore data communication delay, buffers capacity, and branch mispredictions. Without any DOE specific compiler optimizations, our results show that DOE outperforms conventional SpMT architectures by 15%, on average. We also show that DOE with four small cores can perform on average equally well to a large superscalar core, consuming about the same power. Most importantly, DOE improves throughput performance by a significant amount over a large superscalar core, up to 2.5 times, when running multitasking applications.

Categories and Subject Descriptors: C.1.4 [Processor Architectures]: Parallel Architectures

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Speculative multithreading, checkpoint processors, continual flow pipelines, latency tolerant processors

ACM Reference Format:

Sharafeddine, M., Jothi, K., and Akkary, H. 2012. Disjoint out-of-order execution processor. *ACM Trans. Architec. Code Optim.* 9, 3, Article 19 (September 2012), 32 pages.

DOI = 10.1145/2355585.2355592 <http://doi.acm.org/10.1145/2355585.2355592>

1. INTRODUCTION

Conventional Out-of-Order Microarchitectures have practically reached the end of the road when it comes to performance [Agarwal et al. 2000]. Increased performance by

This work is supported by a research gift from Intel Corporation.

Author's addresses: M. Sharafeddine, K. Jothi, and H. Akkary, Department of Electrical and Computer Engineering, American University of Beirut, P. O. Box 11-0236, Riad El-Solh, Beirut, Lebanon 1107 2020; email: {mas117, kmj04, ha95}@aub.edu.lb.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2012 ACM 1544-3566/2012/09-ART19 \$15.00

DOI 10.1145/2355585.2355592 <http://doi.acm.org/10.1145/2355585.2355592>

making cores bigger or wider comes at significant cost in power, area, complexity, and cycle time. This and the advent of single-chip multicore processors pose a new challenge to microprocessor designers: how to provide high single-thread performance for applications that are difficult to parallelize, while allowing more cores to be placed on the same die for high throughput.

One solution to this challenge is to rewrite or automatically parallelize applications using a compiler to take advantage of the parallel multicore hardware. Unfortunately, many applications exhibit little coarse grain parallelism and are too difficult to parallelize due to complex control flow patterns and data communication behavior. Even if it becomes possible with future innovations to automatically parallelize using software the most difficult applications, the engineering and validation effort to parallelize the very large number of existing single-thread applications would be tremendous.

Another solution, if feasible, would be to have multiple cores cooperate on executing a single-thread program. If the program execution can be split properly across multiple cores, it may be possible to extract large quantities of instruction level parallelism (ILP) without having to rely on complex power-hungry out-of-order superscalar cores.

Speculative Multithreading (SpMT), pioneered in the Multiscalar work [Franklin and Sohi 1992; Franklin 1993; Sohi et al. 1995] is one approach for using multiple cores to improve single-thread performance. Although SpMT has been the subject of extensive research for many years, such as [Dubey et al. 1995; Akkary and Driscoll 1998; Cintra et al. 2000; Codrescu and Wills 1999; Codrescu et al. 2001; Chen and Olukotun 1998; Marcuello et al. 1998; Marcuello and Gonzalez 2000; Marcuello 2003; Olukotun et al. 1999; Prabhu and Olukotun 2005; Rotenberg et al. 1997; Steffan and Mowry 1998; Steffan et al. 2000; Tsai et al. 1998; Tsai et al. 1999] and many others, there is still no practical implementation of speculative multithreading in any main stream microprocessor from industry. The main reason is that current high-performance superscalar cores consume a lot of power to extract instruction level parallelism from difficult to parallelize programs. Using multiple of these mainstream superscalar cores for SpMT requires too much power and cannot be justified by the modest sublinear performance improvement reported in current SpMT proposals.

SpMT uses control flow prediction to split a single program stream into multiple tasks or threads [Akkary and Driscoll 1998; Codrescu and Wills 1999; Jacobson et al. 1997; Marcuello and Gonzalez 2002; Rotenberg et al. 1997] and executes these tasks on multiple communicating single-threaded cores. A key design challenge is selecting the proper tasks to minimize thread startup and thread commit overhead, control and data mispredictions, load imbalance, and intertask data dependences and data communication delays [Vijaykumar and Sohi 1998]. Although various studies have shown large amount of instruction level parallelism (ILP) on common non-numeric applications [Lam and Wilson 1992; Nicolau and Fisher 1984; Riseman and Foster 1972], the complex control flow and data communication patterns of these applications exacerbate the load imbalance and intertask data communication problems, making task selection and optimization very difficult and severely limiting the actual amount of ILP achieved in a practical implementation. Hence, conventional SpMT architectures that use multiple large superscalar cores achieve modest performance but consume a lot of power.

In this work, we propose a novel SpMT architecture that we call Disjoint Out-of-Order Execution (DOE). DOE combines SpMT with checkpoint recovery and continual flow pipeline architecture (CFP). Unlike conventional SpMT, DOE overcomes the impact of intertask data dependences and communication delays on performance by applying latency-tolerant CFP to intertask data dependences, as we explain next with an example. By using checkpoints and CFP, DOE achieves latency tolerance of intertask data dependences without using large multiported instruction buffers common in current

Fig. 1. DOE versus conventional SpMT execution.

out-of-order cores, thus presenting the potential for a practical, good performance, and lower power SpMT implementation.

1.1. Comparing DOE to Conventional SpMT Execution

DOE suffers lesser stalls due to intertask data dependences than conventional SpMT. As an example, consider the program instruction stream depicted in Figure 1(a), and the corresponding conventional SpMT execution sequence depicted in Figure 1(b). In the figure, execution time flows from left to right.

The instruction stream is split into approximately equal tasks (T1, T2 and T3) that execute concurrently on multiple cores. Intertask dependences are shown as dotted arrows between producer and consumer instructions. Even though there is a high degree of fine grain parallelism between the tasks, as the majority of instructions do not depend on data from earlier tasks, the actual performance gain of conventional SpMT in this example is limited. This is due to the position of the instructions involved in the intertask dependences and the resulting long execution stalls due to data communication delays. For example, T2 stalls shortly after it starts when it reaches the consumer instruction at the head of the intertask dependence arrow from T1 to T2.

In contrast, DOE handles intertask data communication more effectively than conventional SpMT models as shown in Figure 1(c). DOE treats instructions dependent on data from preceding active tasks differently from instructions that are independent of other tasks. For example, when task T2 encounters its intertask dependence on T1, it defers the execution of the dependent instruction and all its dataflow descendent instructions. These dependent instructions drain out of the core pipeline that is executing T2, with their ready input operands, and wait in a special backup buffer outside the pipeline. Because dependent data thread instructions do not tie up for long core resources, such as reservation stations and reorder buffer entries, the intertask dependences do not stall execution, allowing T2 core to look far ahead for useful independent instructions to execute instead of waiting for input data to arrive from T1.

When T1 execution completes, it forwards its results to T2. T2 then saves the independent instruction results and switches execution to the dependent data thread instructions in the backup buffer. After the dependent data thread executes, its results are merged with the saved independent instruction results. After merging results, T2 resumes normal execution without having to go back to execute independent instructions again.

Notice that even though the dependent instructions in T2 are not contiguous (see Figure 1(b)), the dependent data thread executes continuously until completion (represented by dark rectangle in Figure 1(c)). In this example, DOE completely hides the data communication latency.

Depending on the size of tasks and the amount of parallelism between tasks, DOE in some situations may run out of independent instructions to execute and may reach the end of a task before the previous task completes. In such situations, DOE will suffer a partial data stall.

1.2. Article Contributions

This article makes the following contributions to microprocessor architecture as a first step towards what we believe could be a fundamentally new way in the future for achieving single-thread performance in multicore processors:

- Novel latency tolerant SpMT architecture.* The architecture splits a single-thread program into disjoint control and data threads that execute concurrently on multiple latency-tolerant cores, thus we call the architecture Disjoint Out-of-Order Execution (DOE).
- DOE uses small out-of-order latency tolerant cores to reduce power.* Although in-order latency tolerant cores have been proposed for multicore processors to improve single-thread performance in the presence of frequent data cache misses to DRAM [Nekkalapu et al. 2008; Chaudhry et al. 2009], this is the first use of small out-of-order core architecture that tolerates stalls from intertask data dependences and communication delays between cores in a speculative multithreading architecture. We believe this small core architecture could help overcome excessive power as a major barrier to the adoption of Speculative Multithreading in main stream commercial processors.
- Decoupling performance and task selection from intertask data communication timing.* A conventional SpMT architecture requires careful consideration of the position of producer and consumer instructions that are responsible for intertask data communication. This complicates the task selection process, reduces threading options and often limits performance. DOE automatically and transparently moves all data consuming instructions and their dataflow descendent instructions to the end of the task execution. Consequently, performance in DOE depends mainly on task size distribution and amount of parallelism between tasks, but not on hard to predict intertask communication timing delays.
- DOE performance study as proof of concept.* Using a DOE microarchitecture performance model of the x86 instruction set architecture and simple heuristic to fork control independent threads by hardware at the target addresses of future procedure return instructions, we see 4-core DOE performance comparable to a large superscalar core on Spec2000 integer benchmarks, under realistic DOE processor model constraints. We also see that DOE outperforms conventional SpMT architectures of similar configuration by 15% on single-thread applications, and outperforms by more than 2x a large superscalar processor on throughput oriented applications.

Fig. 2. DOE processor architecture block diagram.

1.3. Article Outline

Section 2 gives an overview of the DOE processor architecture. Section 3 discusses in detail the DOE core microarchitecture. Section 4 outlines the simulation methodology and machine configuration. Section 5 presents performance, power and area simulation results and data analysis of DOE. Section 6 discusses related work and we conclude the article in Section 7.

2. DOE PROCESSOR ARCHITECTURE

We first present a high level overview of the DOE processor architecture.

2.1. DOE Overview

Figure 2 shows a block diagram of DOE processor architecture. The architecture consists of a collection of cores connected in a ring network. Task dispatch logic monitors the execution sequence in each core and assigns tasks to the available cores. A task is a set of contiguous instructions in the dynamic instruction stream. A core fetches and executes a task until the end of task. The end of a task is the start of the next dispatched task. Since tasks are spawned in program order, only the youngest task can spawn another task. Although dispatching tasks out of order is possible [Akkary and Driscoll 1998] and helps load balance in situations where there are deeply nested functions in the code, we choose dispatching tasks in program order, thus ensuring that two consecutive tasks run on two adjacent cores in the ring and simplifying data communication between tasks.

The tasks are selected to start at future control independent points in the program or at future points in the program that have high probability of being reached in a short amount of time by the program execution flow. This ensures that regardless of branch execution within a task, the task will most probably join the next dispatched task. The execution region within the dynamic instruction trace at a given time is bound by the

first fetched instruction in the oldest running task and by the last fetched instruction in the youngest running task. However, instructions within the execution region are fetched, executed, and retired out of order. The program order of tasks matches the cores physical order in the ring. A head pointer and a tail pointer rotate around the ring, pointing to the oldest running task and the youngest running task in program order, respectively. A new task is allocated a core at the tail of the ring, if that core is free. A task at the head commits after it executes all its instructions and reaches (i.e., joins) the first instruction of the next thread. When a task commits, its assigned core becomes free and can be allocated to another task.

Each core has unique characteristics that make it suitable for DOE execution:

- *Control independent execution* [Aho et al. 2006]. Each task is control independent of previous branches. DOE performs recovery from mispredicted branches locally within a core without squashing other tasks. This allows the architecture to exploit distant ILP far beyond mispredicted branches, unlike conventional processors which are limited to the amount of ILP available between mispredicted branches.
- *Disjoint data threads execution*. Each core executes two disjoint data threads out-of-order. One data thread consists of all instructions that are data dependent on previous active tasks, and the other data thread consists of all instructions that are data independent of previous active tasks. The independent data thread starts execution immediately when a task is dispatched, while the dependent data thread is buffered outside the execution pipeline and executes when the previous task completes and commits. By this time all previous mispredicted branches have been corrected and input data propagated from the previous task. The dependent data thread therefore does not block the execution of the independent thread. This achieves two goals: 1) it supports control independent execution of tasks, and 2) it provides tolerance to intertask data dependences and the delays encountered by input data produced by other cores and communicated through the ring and memory.
- *Resource efficiency*. The nonblocking latency tolerant cores are built without large multiported cycle critical buffers. Data dependent threads free all core resources for the independent threads, and wait in a dedicated buffer outside the execution pipeline until their inputs become available. A key feature is that this buffer can be built with dense SRAM memory since its access latency is not critical.
- *Checkpoint recovery and results integration*. The core architecture utilizes checkpoints for resource efficient recovery [Akkary et al. 2003] and for instantaneous integration of results from multiple data threads [Nekkalapu et al. 2008]. By using checkpoints for recovery, completed independent instructions can pseudo-retire, freeing their allocated resources. In case of an exception or misprediction in the dependent data thread, DOE restores precise state from a checkpoint and restarts execution after flushing the pipeline and squashing the task. If the data dependent thread completes without exceptions or branch mispredictions, DOE integrates the results from the dependent and independent threads by merging state from two different checkpoints using flash copy.

2.2. Task Spawning and Prediction

In SpMT, dividing a single program stream into tasks to be executed in parallel can be done by the compiler [Sohi et al. 1995], or completely in hardware [Akkary and Driscoll 1998]. Usually tasks are selected at control independent points or at “pseudo” control independent points, which are entry points to blocks of code that have a very high probability of being reached by the execution stream in the near future. Often, the task

selection mechanism, whether done by the compiler or in hardware, is supported by a hardware predictor that predicts the best next task to spawn [Jacobson et al. 1997].

Several hardware methods of various levels of complexity have been used by researchers for identifying and spawning tasks at run time. Dynamic Multithreading (DMT) [Akkary and Driscoll 1998] exploits well behaved loop and procedure structures and predicts that the static instructions following a loop backward branch or a procedure call to be reconvergence points in the control flow. DMT forks speculative tasks at these points. The DMT prediction method is simple and almost always correct. However, the method is limited by the type of code structure it exploits, and for many applications, provides little or no freedom in choosing where the speculative tasks are forked, resulting in serious load balance issues.

Other simple, heuristic-based methods for identifying and exploiting control independence include Skipper [Cher and Vijaykumar 2001] and Selective Branch Recovery [Gandhi et al. 2004]. Both focus on simple if-then and if-then-else structures, but like DMT's loop and procedure heuristics, fail to identify many control independent or pseudo-independent potential tasks.

The method of Control Quasi-Independent Points (CQIP) is a more aggressive approach for exploiting dynamic control independence [Marcuello and Gonzalez 2002]. Control Quasi Independent points are defined to be future instructions that have 95% or more probability of being on a future control path. This approach leads to higher misprediction rates than the simpler methods in DMT, Skipper and Selective Branch Recovery, and is less suitable for SpMT architectures that have high performance penalty on task mispredictions.

Collins et al. [2004] propose a general technique that does not depend on the compiler or on heuristics to predict various forms of control independence. Unlike heuristic methods, Collins et al. [2004] show that the technique is “robust in the face of aggressive compiler optimizations,” and reports prediction accuracy higher than 99% for SpecInt benchmarks, with different compilers and instruction set architectures.

2.2.1. DOE Task Prediction. In this study, we use a modified DMT fork-on-call heuristic to spawn tasks in hardware because of the simplicity of this method and the unavailability of a DOE compiler at this time. Implementing fork-on-call heuristic dynamically in hardware has allowed us to quickly study and gain early insight into the viability, potential, and key issues of Disjoint Out-of-Order Execution. However, we emphasize that all prior researched task spawning methods are applicable to DOE architecture and we expect further improvements in DOE performance as we study, optimize and vary the task spawning methods in future work.

Unlike the heuristic introduced in DMT, which forks tasks from procedure call instructions at the immediate procedure continuation sites, our heuristic forks tasks at a future procedure continuation, skipping a fixed number of procedures of distance equal to four. This distance was manually selected for this study based on observed performance of our benchmarks, but could be predicted and modified dynamically as well in hardware. Forking tasks after skipping multiple procedures provides better load balance and performance, as we will show in Section 5.

Identifying future procedure continuation tasks requires little hardware. Each DOE core tracks in a queue the PC value of the last 3 procedure call sites in the program stream. When the queue is full and a new procedure call is encountered, the PC of the oldest procedure call entry in the queue and the PC of the newest encountered procedure call, incremented to point to the next static instruction (i.e., newest procedure return target address), are stored in a 4K-entry task predictor table indexed by the call PC. Each entry in the table also contains a 2-bit saturating counter, initialized to the value 2 when a new entry is allocated. When a call in the program stream is

Table I. Simulated Machine Configuration

Baseline Core	15 stage, 4-wide, 160-entry ROB
DOE Core	15 stage, 2-wide, 40-entry ROB
L1 Data Cache	Superscalar: 64KB, 8-way, 3 cycles, 64-byte line DOE core: 16KB, 4-way, 2 cycles, 32-byte line
L1 Instruction Cache	Superscalar: 64 KB, 8-way, 3 cycles, 64-byte line DOE core: 16KB, 4-way, 2 cycles, 32-byte line
L2 Cache	256 KB, 8-way, 12 cycles, 64-byte line
L3 Cache	1MB, 16-way, 40 cycles, 64-byte line
L3 to Memory Latency	150 cycles (assuming integrated DRAM controller)
Branch Predictor	Combined bimodal-gshare 64K, Meta, 64K bimodal, 64K gshare 4K BTB, 4K indirect branch predictor

decoded, a lookup into the task prediction table is executed using the PC of the call. If the lookup results in a hit and the value of the saturating counter in the table is larger or equal 1, a task is spawned using the PC of the future procedure return target address stored in the table. In order to avoid spawning excessively large tasks, which we observed to harm performance due to load imbalance and hardware instruction buffers limitation, the two bit counter is incremented when a forked task is joined after executing a minimum of 200 instructions. It is decremented if the parent task executes more than 5000 instructions before joining the spawned task. Moreover, the spawned task is squashed if the parent task executes more than 10000 instructions before joining with the spawned task. All these task size numbers used in our training algorithm are static numbers that were observed empirically to provide reasonable performance. We hope to improve performance by optimizing the task prediction training mechanism using dynamic size selection in future work.

2.3. Cache Hierarchy

The DOE architecture studied in this article uses a private L1 instruction cache, private L1 data cache and private Miss Status Holding Registers (MSHR) [Kroft 1981] for every core. A MESI (Modified, Exclusive, Shared, Invalid) cache coherence protocol, modified to account for speculative execution state, is used in this study. L2 and L3 caches are unified caches. Cache associativity and sizes are listed in Table I.

2.4. Branch Predictor

All cores use a combined bimodal-gshare branch predictor [McFarling 1993]. Each core has its own bimodal array, branch history register, and gshare and Meta predictor arrays. The bimodal predictor is updated locally from its own core. Only one core, the one corresponding to the oldest nonspeculative task, updates the global and Meta predictor state in all cores. Notice that unlike reads, these updates are not in a critical path.

SpMT requires careful implementation of the branch predictor due to short threads and out-of-order fetch of instructions by multiple cores. Issues associated with branch prediction in SpMT processors were studied by Bumyong et al. [2008]. We use the method proposed in their work of initializing the history register for the gshare predictor using the PC of the first task instruction. Lookups during the execution of the task and updates during task commit of the gshare branch prediction counters are consistent, both using the PC value of the task as initial history register state. Each core has its own return address stack [Kaeli and Emma 1991] to predict targets of return instructions in the task.

Fig. 3. DOE core architecture block diagram.

3. DOE CORE MICROARCHITECTURE

The DOE core microarchitecture we use in this study is 2-wide out-of-order similar to the Pentium Pro processor architecture [Papworth 1996] and enhanced with Continual Flow Pipeline capability [Srinivasan et al. 2004; Nikkalapu et al. 2008]. Continual Flow Pipeline processors have many similarities to other latency tolerant architectures, such as, Out-of-Order Commit processors [Cristal et al. 2004] and Sun’s Rock many-core processor [Chaudhry et al. 2008]. Figure 3 shows a block diagram of the DOE core microarchitecture.

The DOE core handles dependent thread instructions differently from independent thread instructions. When a task is spawned, a bit mask that identifies the set of influence registers is loaded from a hardware predictor into a set of logical registers poison bits in the rename map table of the core that is assigned to execute the spawned task. Influence registers are live-in input registers of a spawned task that are modified by the parent task after the spawn point. The dependents of all influence registers and their descendants are extracted from the task instruction stream using the poison bits in the rename map table and are stored in a Dependent Thread Buffer (DTB) outside the pipeline. The dependent thread instructions therefore do not consume or occupy pipeline resources such as reservation stations or reorder buffer entries. This completely frees the pipeline resources for independent instructions to execute. Since dependent instructions do not tie pipeline resources, the core achieves a continual flow of execution and can look ahead far into the task for independent instructions to process until the task reaches the end.

It is not sufficient to account only for intertask data dependences that occur through registers. Intertask data dependences can be through memory as well. Since intertask dependences may occur through memory stores and loads instructions, DOE uses memory dependence prediction [Moshovos et al. 1997] to identify and poison intertask dependent loads and their intra-task dependent instructions.

In order to minimize bubbles in the pipeline due to dependent instructions poisoning and extraction, and to allow the front end to provide an instruction supply sufficient to saturate the issue stage into the reservation stations, we place small decoupling queues between the rename stage and the scheduling logic (reservation stations), as well as between the rename stage and the DTB. We add one read port into the reorder buffer to support the additional read bandwidth required by dependent instructions to read any nonpoisoned input registers. This read happens when an instruction moves from the queue to the DTB. If the nonpoisoned input register of the dependent instruction at the head of queue is not marked valid in the reorder buffer (i.e., has not been computed and written back), the queue stalls and then waken-up when that register is written back.

When the previous task completes, a checkpoint is taken of the architectural register file, containing at this time the results of the independent instructions, and execution

switches to the dependent instructions from the DTB. When all the DTB instructions execute, their results are merged with the independent instruction results in the saved checkpoint. The checkpoint is then discarded and execution resumes normally without having to go back to execute independent instructions again. We next describe various details of this microarchitecture.

3.1. Independent Thread Execution and Dependent Thread Construction

We call execution in this phase independent execution. This phase starts when the task dispatcher spawns a task.

At the beginning of this phase, a bit mask corresponding to all influence registers is loaded into the poison bits entries in the rename map table. This bit mask is predicted dynamically and supplied using hardware prediction. Noninfluence registers are also transferred from the previous task to the architectural register file in the core assigned to the spawned task. These registers are identified by the complement of the influence register bit mask provided by the hardware predictor. The live-in subset of these registers contains all input registers required by the independent thread to execute.

Dependent instructions in this execution phase are identified by propagating poison bits from producer to consumer instructions. Instructions that read poisoned register entries in the rename map table are dependent instructions. They proceed from the rename stage to a dependent instructions queue from which they are then written into the DTB. Dependent instructions read nonpoisoned source operands from the reorder buffer or from the architectural register file if the operands have been retired. These nonpoisoned operands are stored with the dependent instructions in the DTB. Instructions that do not read poisoned registers are independent and proceed from the rename stage to an independent instructions queue. They are then moved from the independent queue into their reservation stations from which they are scheduled for execution when their input operands become ready.

Intertask memory dependences require a different poisoning mechanism. A load that depends on a store from another active task is poisoned even if its source registers are not. Its destination register poison bit in the rename map table is set to propagate the poisoning to its dependent instructions.

The independent execution phase ends when the previous task completes.

3.2. Dependent Thread Execution

This phase starts when the previous task completes. At the beginning of this phase, DOE creates a first checkpoint (Ind Chkpt latch, Figure 4) of the poison bits from the rename table and architectural register file (RF latch, Figure 4), which contains at this point in time the execution results of the independent instructions. The influence registers are then copied from the previous task to the architectural register file. These registers are inputs required by the dependent thread. The rename map table is reset to map all registers to the architectural register file.

Instruction renaming and execution then switches to the dependent instructions in the DTB. Some Instructions in the DTB can have input operands stored with them from the time these values were produced during the independent execution phase. DTB instructions carry these operands with them as immediate operands. The DTB instructions also have input operands that were poisoned. The poisoned input operands are outputs of other instructions in the DTB and are computed during the dependent execution phase and propagated normally using the register renaming mechanism. The dependent execution phase ends when all instructions in the DTB are redispached and executed.

Fig. 4. Register file cell with two checkpoints.

3.3. Normal Execution

A task is in normal execution phase after the dependent execution phase ends and until the task completes. A task that is executing in normal mode is the oldest task in the ring.

3.4. Partial Data Stalls

A task encounters what we call a partial data stall if it reaches the next task during independent execution. The task waits in this state until the previous task completes.

3.5. Checkpoints

DOE uses a flash copy of the architectural register file for creating checkpoints. A checkpoint can be created in one clock cycle using a logical shift operation. Figure 4 shows the DOE customized register file cell with its checkpoint support logic. When `CHKPT_CLK` is asserted, the register file bit (RF latch) is shifted to a first backup latch called `Ind Chkpt` and the `Ind Chkpt` bit is shifted into a second backup latch called `Commit Chkpt`. The register file values can be restored from the either checkpoint using a 2-to1 multiplexer by asserting `RSTR_CLK`.

3.6. Independent and Dependent Results Integration

Result integration is a special checkpoint copy-and-restore sequence. At the end of independent execution phase, the independent register file state (RF latch, Figure 4) is saved in the independent checkpoint (`Ind Chkpt` latch, Figure 4). The poison bits vector in the rename map table is also saved. At the end of the dependent execution, in the absence of dependent thread exceptions or branch mispredictions, a restore cycle is performed from the `Ind Chkpt` latch to the RF latch. However, not all registers are restored. As shown in Figure 4, only the nonpoisoned registers are copied by using the poison bits to gate the clock of the restore operation. This effectively merges the results of the dependent instructions in the RF latch with the independent results that have been saved in `Ind Chkpt` latch, seamlessly creating the precise architecture state needed to proceed with normal execution of the task.

3.7. Commit Checkpoint

When a task completes execution, DOE commits the task register state results by performing two shift operations in the register file cell shown in Figure 4, thus saving the register file state (RF latch) into the commit checkpoint (`Commit Chkpt` latch).

DOE uses the commit checkpoint to recover when a task encounters an exception or dependent branch misprediction. In this situation, DOE squashes the task and restarts it in normal execution using the commit checkpoint state from the previous task.

3.8. Speculative Cache and Load and Store Execution

Like other SpMT proposals, DOE allows multiple speculative cache block versions to reside simultaneously in the distributed L1 data caches of the various cores. A task creates, reads and writes its speculative data blocks in its own cache during its speculative execution. The speculative data created by a task is flushed when the task is squashed as a result of data misprediction or exception. The speculative data is committed if the task completes without any data mispredictions or exceptions.

We emphasize a key property of DOE speculative cache that differentiates it from previous SpMT speculative caches. A DOE task accesses its own speculative data blocks, but not those of other tasks. This is a consequence of DOE execution algorithm, which executes independent instructions immediately and defers the execution of intertask dependent instructions until all previous tasks complete and commit their speculative data. Hence, DOE does not need to maintain a version order list or require version control logic to select the properly ordered speculative version to access [Gopal et al. 1998].

The rest of this section describes in more detail the DOE speculative cache states and load and store execution.

3.8.1. Speculative Cache State. In order to support speculative execution of dependent and independent loads and stores and maintain cache coherence, DOE adds two new cache block states: Speculative Independent (Spec_Ind) and Speculative Dependent (Spec_Dep). A block that is not in one of these two states is committed and would be in one of the states defined by the cache coherence protocol, for instance, Shared, Exclusive, or Modified in a MESI coherence protocol.

3.8.2. Independent Store Execution. Independent stores from the independent thread allocate blocks in the first level cache and write them speculatively after the stores pseudo-retire. Independent stores set the Spec_Ind bit in the written cache block. If an independent store hits a Modified block in the L1 data cache, the block is first written back to the L2 cache before the store writes the block and changes it into Spec_Ind state. Independent stores are also written in a special nontagged, SRAM buffer we call the Store Redo Log (SRL), at the same time they are written into the L1 data cache.

3.8.3. Independent Load Execution. Independent loads read Spec_Ind or Committed blocks in the L1 cache, whichever they happen to hit. Independent stores therefore forward data to their descendent independent loads through the data cache, long before they are actually committed. This allows DOE to deallocate stores from the store queue immediately at pseudo-retirement, keeping the store queue as small as possible.

3.8.4. Dependent Store Execution and Store Commit. Since independent stores are written into the SRL during the independent execution phase, by the time the dependent execution phase starts the independent subset of the task stores have already been written into the SRL. To create and commit the complete, program ordered memory state of the task, dependent stores from the dependent thread execution phase are written first into the SRL, interleaved in program order with the independent stores. As the dependent stores retire, the dependent thread commits all stores, dependent and independent, from the SRL into the data cache, in program order, this time setting the Spec_Dep bits in the cache block. If an SRL dispatch store hits a dirty block in the L1 data cache, the block is first written back to the L2 cache before the store writes the block and changes it into Spec_Dep state. Notice that Independent stores are written

twice into the data cache: 1) speculatively by the independent thread to forward data to independent loads, and 2) by the dependent thread interleaved with dependent stores in program order to enforce a final, correct order of memory writes. After the dependent thread executes, and the DTB and SRL become empty, all Spec.Dep blocks of the task are bulk committed, thus leaving in the cache ordered stores data.

3.8.5. Dependent Load Execution. Dependent loads are re-issued from the DTB and execute from the data cache after all stores ahead of them in the SRL are committed to the cache. Synchronizing the dependent load execution with the stores ahead of them in the SRL is performed using SRL IDs assigned to all loads in the front end of the pipeline when they are fetched and decoded. The load SRL ID identifies the entry in the SRL that was assigned to the last prior store. Notice that since dependent loads execution is synchronized with older SRL stores, memory dependence mispredictions of dependent loads do not require a task recovery squash, due to the fact that a dependent load always reads ordered store data, either from a committed block or from a Spec.Dep block in the cache.

3.8.6. Speculative Cache Coherence. DOE modifies the MESI cache coherence protocol to support speculative cache state and speculative execution as follows:

1. Independent stores that hit Modified blocks write back these blocks to L2 before updating and changing the blocks to Spec_Ind state.
2. Independent stores that miss the cache result in a Read-to-Shared request to the other cores and L2 cache.
3. All requests from other cores to Spec_Ind blocks are treated as misses, and no action is taken. This is because in DOE, Spec_Ind state is used for the purpose of propagating data from stores to loads within the same task and not for forwarding store data to other tasks.
4. SRL stores (dependent or independent) that hit Modified blocks write back these blocks to L2 before updating and changing the blocks to Spec_Dep state.
5. SRL stores (dependent or independent) that miss the cache result in Read-Invalidate requests to the other cores and L2 cache.
6. Read requests from other cores' independent loads to Spec_Dep blocks are treated as misses. This case could happen due to memory dependence mispredictions.
7. Read requests to Spec_Dep blocks from other cores' dependent loads or other cores' SRL stores can never happen in DOE, since only the oldest core, at any point in time during execution can be in the dependent execution phase.

3.9. Task, Influence Registers and Stack Pointer Value Predictor

The task predictor described in Section 2.2 is supplemented with 4K-entry value prediction table used to predict influence registers mask and stack pointer register value. This table is indexed by the XOR of the forking call instruction PC, task PC, and 8 bits of global branch history register at the forking call instruction.

The influence registers mask in the value prediction table has a bit set if the corresponding register changes between the forking call instruction and the first task instruction. The influence mask bits are sticky. Once a bit is set, it stays set as long as the task entry stays in the task prediction table. Notice that the set of influence registers may vary between one task execution instance and another, depending on how branches between the spawning point and the spawned task execute, which consequently determines the execution path and the registers that are modified in the path. By making the influence mask bits sticky in the prediction table, we ensure that the predictor, after long enough training, will capture all possible influence registers, regardless of the dynamic path taken. This conservative approach reduces mispredictions

that fail to identify influence registers, which are quite costly as they require squashing the task to recover. On the other hand, this approach sometimes mispredicts by falsely identifying registers to be influence registers, when they are not. However, these mispredictions are less costly to performance since they result in over-poisoning instructions and deferring their execution to the dependent phase, but do not require squashing the task.

The value prediction table also contains the difference in the stack pointer register value between the forking call instruction and the first instruction of the forked task. When a new task is spawned, the value stored in the table is added to the current SP value and used as the predicted SP value during the spawned task execution. Our empirical data, presented in Section 5, shows this SP value prediction method to be highly accurate.

3.10. Memory Ordering and Memory Dependence Prediction

3.10.1. Memory Ordering and Store Redo Log. As described in the previous section, DOE uses a Store Redo Log (SRL) to maintain proper memory ordering of loads and stores from the independent and dependent execution phases [Gandhi et al. 2005]. During independent execution, all stores (independent as well as dependent) are assigned entries in the SRL in program order. Every load (independent as well as dependent) is assigned the SRL ID of the prior store in the program, to provide a mechanism for identifying the relative program order of all loads and stores.

When independent stores pseudo-retire, they are written speculatively, in program order, from the store queue into the first level cache, so that independent loads can read data from these stores by accessing the cache. These stores are also written in the SRL buffer in the entries that were assigned to them in program order (see Figure 3). When the core switches to dependent execution, the speculative blocks (Spec.Ind) in the cache are cleared. The independent stores are then redone, by writing them again into the cache from the SRL. This time however, they are properly interleaved in program order with the dependent stores and written into the data cache, as the dependent stores issue from the DTB and write their entries in the SRL buffer.

3.10.2. Intratask Memory Dependence Prediction. During independent execution, loads may be poisoned due to memory dependences, and deferred until the dependent execution phase for two reasons: 1) if they depend on dependent stores from the same task (intratask memory dependences) or 2) if they depend on stores from previous active tasks (intertask memory dependences). We have used a conventional store-sets predictor [Chrysos et al. 1998] to predict intra-task memory dependences and poison dependent loads.

3.10.3. Intertask Memory Dependence Prediction. For the second category (i.e., intertask memory dependences), we could not use a conventional store-sets predictor, since loads and the stores from different tasks are often fetched out of program order, making the store addresses unknown when their dependent loads are fetched. Instead, we used a new mechanism that we call task-sets memory dependence predictor. As in store-sets memory dependence prediction, our mechanism tracks in a prediction table sets of loads that are dependent on previous tasks using unique task-set IDs. The task-ID comes from the index into the task predictor table, which is the spawning call instruction PC as described in Section 2.2.1. The task-sets prediction table has one bit per entry and is indexed by the task-ID concatenated with a load instruction PC. When a bit value of 1 is read from the task-set-ID table, the indexing load is predicted to depend on a store from a previous task and is poisoned. When a bit value of 0 is read, the indexing load is predicted to be independent. The bit is set when a load to store intertask dependence is detected.

3.10.4. Detecting Memory Dependence Mispredictions. Memory dependence mispredictions are of two types: false positives, or false negatives. A false positive occurs when the memory dependence predictor predicts that an actually independent load is dependent. The performance impact of false positives in DOE is less severe than in conventional SpMT, since in DOE, poisoned loads are deferred without stalling task execution. A false negative occurs when the memory dependence predictor predicts that an actually dependent load is independent. Again, the performance impact of false negatives in DOE is less severe than in conventional SpMT. This is because in case of false negative misprediction, the task to which the mispredicted load belongs has to be squashed and execution has to be rolled back to a checkpoint; however, unlike previous SpMT implementations, subsequent active tasks do not need to be squashed.

As in Gandhi et al. [2005], we propose using a set associative load address buffer (LAB). All loads from a task receive entries in the LAB in which they write their addresses. The LAB snoops committed SRL stores' Read-Invalidate requests from other tasks as well as from the same task. A snoop hit indicates that a load has executed before an earlier store, thus memory dependence violation has occurred. If the dependent execution phase completes without detecting a memory dependence violation, the speculative stores in the cache are bulk committed and the load address buffer is reset before the execution switches to normal mode. However, if a memory dependence violation is detected before the dependent execution phase completes, the task is squashed and its speculative stores in its cache (Spec_Dep) are bulk cleared.

The load address buffer also snoops stores' Read-Invalidate requests from other threads and processors for memory consistency.

3.11. Task Commit

A task commits when it completes execution and becomes the oldest task. The register file state is saved in a retirement checkpoint (Commit Chkpt latch, Figure 4), freeing the core resources to execute another task. There is one and only one retirement checkpoint in the system at any time and it is in the last core that has committed a task.

3.12. Handling Exceptions and Branch Mispredictions

Exceptions and branch mispredictions encountered during independent or normal execution are handled normally by flushing the execution pipeline behind the mispredicted branch and clearing DTB entries younger than the branch. Assigning to branches the last allocated DTB entry is one method to identify which DTB entries are younger than the mispredicted branch, which is a method similar to one mechanism typically used for clearing load and store queue entries in superscalar processors.

On the other hand, exceptions and mispredicted branches that occur during dependent execution have to be handled in a special way. A task incurring a dependent exception is squashed with all subsequent tasks and execution restarts from the retirement checkpoint after all previous tasks commit. A task incurring a dependent branch misprediction is squashed (but the subsequent tasks are not) and waits until it becomes the oldest task on the ring. At that time, it is restarted from the retirement checkpoint of the previous task.

4. EVALUATION METHODOLOGY

To measure the potential performance of DOE, we have developed a constraint-based pipeline timing model of a DOE multicore processor using PTLSim simulation infrastructure [Yourst 2007] supported with an ORACLE execution model. PTLSIM simulates x86 code after converting complex instructions into RISC-like micro-ops (uops), a technique used in Intel processors [Papworth 1996]. The ORACLE execution model allows us, in some of the experiments that we have performed, to predict DOE

performance under ideal conditions and to measure the performance impact of interesting variables in isolation, such as the impact of memory intertask dependences, register intertask dependences, and dependent mispredicted branches. We measure DOE performance relative to a large 4-wide out-of-order superscalar configured to approximate state-of-the-art current processors core architectures.

Our research goal is to study whether DOE architecture could achieve with about the same energy consumption of a large superscalar core: 1) single-thread performance comparable to the large core, 2) better throughput on multithreaded applications, and 3) better performance than conventional SpMT architecture of similar configuration/area and spawning method. We use a small out-of-order DOE core, with half the width (2-wide) of our large core configuration. We size all DOE core buffers (re-order buffer, reservation stations, load-store queue, etc. . .) to be one fourth the size of the corresponding large core instruction buffers. We also size the total SRAM to be the same in the large superscalar, multicore SpMT and multicore DOE processor models, while accounting for all levels of cache as well as for the extra SRAM memory required in the DOE cores for the DTB, SRL, and LAB. Assuming 16-way 1M byte of L3 cache per large superscalar core and multicore SpMT processor, we have run all DOE simulations with 14-way L3 cache, which leaves 128K of SRAM storage for DOE core buffers. We assumed that the hardware added for DOE support does not cause any significant degradation in clock cycle time. Similarly, we assumed no clock cycle benefit from the reduced SpMT and DOE pipeline width and core buffers.

With our combined ORACLE-PTLSIM timing model, we have collected data for 9 spec2000 integer benchmarks. The spec2000 integer benchmarks are known to be good examples of complex, difficult to parallelize nonnumeric programs. In our experiments, we collect data for a sample of 100 million instructions after skipping the initialization phase of each benchmark and 20 million more instructions for cache and predictors warm-up. Table I shows the pipeline, cache, and branch prediction configuration.

5. RESULTS AND ANALYSIS

We performed a series of experiments starting with a limit study of an ideal machine with perfect value prediction, and increasingly adding architecture restrictions until we reached a realistic machine model of the DOE architecture described in Sections 2 and 3. This series of experiments has given us key insights into how we may proceed in the future towards designing a well optimized DOE implementation.

Section 5.1 discusses task prediction training using an ideal machine with perfect value prediction. Section 5.2 presents a limit study of DOE performance with perfect value prediction for 3, 4, 6, and 8 cores. In Section 5.3, we evaluate the performance impact of intertask data dependences. Section 5.4 presents the benefit of predicting the stack pointer register value. We report on branch misprediction impact on performance in Section 5.5, while Section 5.6 shows DOE tolerance to intercore data communication delay. Section 5.7 discusses how buffer sizes influence performance, and Section 5.8 presents performance for a realistic 3, 4, 6, and 8-core DOE configurations with all optimizations from previous sections included. We discuss DOE prediction hardware results in Section 5.9. Section 5.10 compares DOE performance to conventional SpMT architecture of equal configuration and area. Section 5.11 reports on throughput performance of DOE when running multiple programs compared to large superscalar core. Section 5.12 presents and discusses results from our area and power models of DOE and compares DOE to the large baseline superscalar. Finally, Section 5.13 summarizes the results and key insights from all the experiments.

Fig. 5. Ideal speedup over baseline of 4-core DOE with two fork methods.

5.1. Training the Task Predictor

Spawning fork-on-call tasks [Akkary and Driscoll 1998] requires simple hardware and no code changes or recompilation. When execution encounters a call, a new task is forked at the next static instruction after the call, which is the target of the called procedure return instruction. The core that forks the new task continues executing the called procedure, while the next core on the ring executes starting from the procedure return target instruction. This task dispatch approach is simple and attractive for applications that have frequent procedure calls and for which it is necessary to maintain binary compatibility. However, there is little freedom in controlling the size or placement of fork-on-call tasks.

Figure 5 shows DOE performance relative to the baseline out-of-order core for two forking methods that exploit procedure control flow structure in programs. Both methods fork tasks at future procedure continuation (i.e., return target) points. The fork-on-call task starts at the return target of the immediate procedure, while with multiple-proc-fork method, a task starts at a procedure continuation point of a future procedure, after skipping the immediate next 4 procedures. Section 2.2.1 describes the multiple-proc-fork predictor in detail.

In this experiment, we assume a conservative 32 clock cycles to copy live-in registers between two adjacent cores on the ring as overhead for each of task dispatch and task commit. We measure for the two fork methods the potential performance of 4-core DOE configuration relative to our large core baseline performance. We have run this experiment with perfect value prediction for all register and memory intertask data dependences. This implies that a task never waits for input values from previous tasks, since the input values are known from the perfect value prediction. Using this ideal model, we study the performance of the two fork methods and the impact of task size and load balance on performance without interference from intertask data communication.

We observe two factors limiting fork-on-call performance: task size variation and task execution overlap. Since tasks are dispatched at immediate procedure continuations, task size is determined by the number of instructions between the procedure return target instruction at which a task is forked and the next procedure return target where the next task starts. On the other hand, task execution overlap between a parent task and its child is determined by the distance from the point where the child is dispatched (procedure call instruction) to the procedure return. This distance is the size of the procedure. We notice that fork-on-call suffers from significant variation in task size, ranging from less than hundred instructions per task and up to a few million instructions for some benchmarks, such as perl. This wide variation in task size causes severe load imbalance since cores that execute small tasks finish quickly and then stay idle for a long time waiting for preceding large tasks to finish. We have also observed that

Fig. 6. Limit study: DOE performance with 3, 4, 6, and 8 cores and perfect value prediction.

many forked procedures are very small (less than 30 instructions). In such cases, the dispatch and commit overhead relative to the concurrent execution time of the parent and child is too high to benefit overall performance. As a result of load imbalance, the average performance of 4-core DOE configuration with fork-on-call is about the same as a large core, with bzip2, perl and twolf significantly underperforming a large core. This is a very poor performance for a limit study which does not account for intertask data dependences, task squashes from dependent branch mispredictions, or buffer sizes.

In order to improve load balance and subsequently performance, we developed the forking and training mechanism described in Section 2.2.1, in which we still fork at procedure continuation points, but skip up to four procedures when necessary to try and spawn tasks with sizes that are neither too large (<5000 instructions target), nor too small (>200 instructions target). Figure 5 shows that this forking method gives significant improvement in performance for all benchmarks. The average performance of multiple-proc-fork method jumps to 1.5x the large core baseline performance with all benchmarks outperforming the baseline. For all the experiments in the remainder of this section, we use this multiple-proc-fork training method.

5.2. Limit Study of DOE Performance with 3, 4, 6, and 8 Cores

Figure 6 shows DOE performance relative to the large baseline out-of-order core for 3, 4, 6, and 8 multicore DOE configurations, using ideal limit-study parameters such as perfect value prediction, perfect branch prediction and unlimited DTB and SRL buffer size for holding dependent instructions and stores. Performance improves up to 8-core DOE configuration, most notably for bzip2, twolf, and vpr. The ideal 4-core DOE configuration shows average speedup of about 1.5 times over the baseline.

Since we estimate later in Section 5.12 that the 4-core DOE configuration is about equal in power consumption to the large out-of-order baseline, we next explore the potential performance of realistic 4-core DOE. We hope that a realistic 4-core DOE would provide comparable performance to a large out-of-order superscalar, thus achieving our goal of using multiple small cores architecture for high performance on parallel workloads, without sacrificing single-thread performance or energy efficiency of inherently serial applications. We will repeat again in Section 5.8 this study of DOE performance versus number of cores with a realistic machine configuration. However, we first present the individual impact of each of the various realistic parameters we will use in our final realistic DOE machine model.

Fig. 7. Limits of memory and register communication on 4-core DOE performance.

Table II. Instruction Poison Statistics

	bzip2	Eon	Gap	gzip	mcf	parser	perl	Twolf	vpr
% Instr. Poisoned Through Memory	35%	14%	17%	39%	30%	24%	11%	63%	45%
% Instr. Poisoned Through Registers	72%	47%	51%	51%	32%	56%	47%	66%	72%
% Total Instr. Poisoned	76%	51%	59%	57%	33%	64%	47%	74%	74%
% Total Instr. Poisoned, with RSP Value Prediction	51%	27%	32%	39%	30%	32%	16%	64%	51%

5.3. The Impact of Memory and Register Intertask Dependences on Performance

Figure 7 shows the impact of memory and register intertask data dependences on 4-core DOE performance with perfect branch prediction and unlimited DTB and SRL size. The graphs show 4-core DOE performance relative to the baseline out-of-order core with: a) perfect register and memory value prediction, b) perfect memory value prediction, c) perfect register value prediction and d) without register or memory value prediction. Perfect register and memory value prediction eliminates delays from all data communication between tasks while perfect register or memory value prediction eliminates delays from register and memory value communication, respectively. Without value prediction, the model accounts for all register and memory data communication, thus reflecting realistic delays of a practical DOE machine implementation. Comparing performance with and without perfect value prediction gives us an insight into the impact of memory and registers dependences on DOE performance and provides us upper bounds on the potential benefit of register and memory value prediction, techniques common in Speculative Multithreading architectures.

Figure 7 indicates that on average, performance is less impacted by memory data communication between tasks than register communication. This can be explained by the amount of dependent instructions due to memory dependences and register dependences. The execution of dependent instructions in DOE is deferred until the previous task commits. If the number of deferred instructions in a task is large relative to independent instructions, the task encounters partial data stalls since it cannot completely hide the latency of the input data from the previous task.

Table II, rows 2 and 3, and 4, show the percentage of instructions poisoned due to intertask memory data dependences, intertask register data dependences, and all intertask data dependences, respectively. We make two observations. First, the percentage of task instructions dependent on register input values from other tasks is a lot higher than the percentage of task instructions dependent on memory input values. Second, the percentage of instructions poisoned from all intertask dependences (row 4) is very close to the percentage of instructions poisoned by register dependences

Fig. 8. Impact of perfect register and RSP value prediction on 4-core DOE performance.

(row 3). This indicates that the percentage of instructions poisoned due to only memory dependences (difference between rows 3 and 4) is relatively small.

These observations are surprising given that x86 compilers rely a lot more on static and stack memory storage for nontemporary variables, due to the small number of general purpose x86 registers compared to RISC. Looking further into this issue has led us to a key optimization that we discuss next.

5.4. Predicting the Stack Pointer Register Value

We investigated the high percentage of task instructions dependent on register values to understand more about the nature of these dependences. We found that a large percentage of register dependences come from the stack pointer register, (RSP), which is natural for an architecture like x86 that has a small number of registers.

The value of the stack pointer register after returning from a procedure is highly predictable. Procedure execution simply adjusts by some offset the value of the stack pointer in order to save and restore temporary variables for the procedure on the stack. Even though the RSP value may be rewritten between the procedure call and return multiple times, the value in the end is independent of the stack adjustment in between. In other words, RSP register is a false influence register and the apparent dependence on RSP is a false dependence and can be eliminated with value prediction. Postiff et al. [1999] made a similar observation and reported significant increase in the potential amount of measured instruction level parallelism in Spec95 integer benchmarks, after eliminating all stack dependences caused by the compiler from using the memory stack for procedure linkage.

After applying value prediction to the RSP register, the percentage of dependent instructions drops significantly as seen in Table II, row 5. Performance with RSP value prediction also improves by a significant amount and is very close for most of the benchmarks to perfect register value prediction performance, as shown in Figure 8. For all experiments that we discuss in the remainder of this section, we use RSP value prediction.

5.5. The Impact of Branch Misprediction on Performance

The latency tolerant execution of DOE changes the branch prediction problem and tradeoffs in a fundamental and interesting way. The performance cost of independent mispredicted branches is the usual penalty resulting from the pipeline flush that occurs when a mispredicted branch executes. An optimized DOE core may have an advantage in independent mispredicted branch penalty over the large baseline core, since the smaller DOE core could be implemented with shorter pipeline without impacting cycle time. Dependent branch mispredictions in DOE however are very costly. The execution of a dependent mispredicted branch is deferred, potentially for hundreds

Fig. 9. Impact of dependent branch mispredictions on 4-core DOE performance.

Fig. 10. 4-Core DOE Performance sensitivity to task dispatch and commit overhead.

of cycles, until the previous task commits and the input values from the previous task become available. By the time the dependent mispredicted branch executes, the only way to recover is to squash the task and restart from the retirement checkpoint of the previous task. The difference in misprediction cost between dependent and independent branches creates an opportunity to optimize the branch predictor specifically for dependent branches. This was also observed in memory latency tolerant processors [Srinivasan et al. 2004; Chaudhry et al. 2008] but neither work presented an actual optimization.

We have added to the DOE core 4K-entry indirect branch predictor [Chang et al. 1997], specifically targeting dependent mispredicted branches. Even though the benefit of the indirect branch predictor on the large baseline out-of-order core is minimal, we have seen significant benefit in DOE, up to 10%, on some of the benchmarks that have a large number of mispredicted dependent indirect branches, such as gap and perl.

Figure 9 shows the performance of 4-core DOE relative to the baseline with a) perfect dependent branch prediction and b) with the baseline predictor configuration enhanced with indirect branch predictor. Although the percentage of dependent mispredicted branches is 1% of the total mispredicted branches on average, dependent mispredicted branches cost 11% performance in DOE due to their high misprediction penalty.

5.6. DOE Tolerance to Intercore Communication Latency

With a training algorithm optimized for large task execution overlap and task size, and with a core architecture that defers the execution of dependent instructions until input data becomes available, we expect performance to have little sensitivity to data communication delay overhead incurred when a task is forked and when a task commits. Figure 10 shows performance relative to baseline out-of-order core to be almost unchanged for task dispatch and task commit overhead of 32, 64 and 128 cycles.

5.7. Buffer Size Study

We have observed, due to the position of calls and returns in the dynamic instruction stream, task sizes that vary widely from hundreds to thousands instructions. For

Fig. 11. Performance of 4-core DOE with various buffer sizes.

Table III. Task Statistics for Realistic 8-Core DOE

Benchmark	Average Task Size	Maximum Task Size	Poisoned Instructions %
bzip2	863	5473	51%
Eon	2364	31209	24%
Gap	1712	27749	38%
Gzip	1881	1209602	39%
Mcf	32965	1164800	8%
Parser	1150	740997	35%
Perl	1496	7215641	15%
Twolf	7069	45331	39%
Vpr	5542	154330	37%

achieving good performance, large tasks require large buffers to defer the dependent data thread (DTB buffer) and for redoing stores to memory (SRL buffer). Figure 11 shows DOE performance relative to baseline for various DTB and SRL buffer sizes, up to 512-entry SRL and 2K-entry DTB. A limit study with no restriction on buffer size is also shown in Figure 11.

A majority of the benchmarks reach performance close to the limit study performance with the 512-SRL, 2K-DTB configuration. For these benchmarks the majority of tasks fit within the DTB and SRL buffers. Other benchmarks, such as eon, mcf, twolf and vpr, are too large on average to fit in the DTB buffer (Table III) and do not reach close to the ideal limit study performance.

Using large buffer sizes to obtain good performance may be undesirable for die area, but is implementable in hardware with little complexity. Unlike large buffers in out-of-order superscalar processors, the DTB and SRL buffers are single ported, and easier to design since they do not create critical timing loops in the pipeline. Both buffers can be implemented using a high density SRAM cell design like SRAM cells used in first level instruction and data cache arrays.

One option to reduce the area required for the DTB buffer is to store in the DTB pointers to instructions in the L1 instruction cache (ICache), instead of the dependent instructions themselves. This requires adding bits in the ICache to mark the dependent instruction blocks and to avoid evicting them from the ICache until the dependent thread executes and finishes. For the x86 instruction set architecture, such cache mechanism is required to handle self-modifying code.

5.8. Performance of Realistic DOE configuration with 3, 4, 6, and 8 Cores

Figure 12 shows the performance of a realistic DOE configuration with 3, 4, 6, and 8 cores, relative to the large out-of-order baseline. The architecture performance model in this experiment includes a task prediction training mechanism, combined bimodal-share predictor enhanced with indirect branch prediction mechanism, stack pointer

Fig. 12. Performance of realistically configured 3-core, 4-core, 6-core and 8-core DOE.

Table IV. DOE Predictors Misprediction Rates

	Task Predictor Miss Rate	RSP Predictor Miss Rate	Influence Registers Predictor Miss Rate	Memory Predictor Task Squashes %
Bzip2	1%	1%	1%	0.2%
Eon	6%	8%	1%	3%
Gap	20%	17%	2%	3%
Gzip	1%	1%	1%	1%
Mcf	1%	2%	1%	1%
Parser	5%	5%	2%	3%
Perl	13%	1%	1%	0.2%
Twolf	27%	1%	3%	0.5%
Vpr	8%	1%	1%	0.1%

register value predictor, 32-cycle task dispatch and commit overhead and 2K dependent thread buffer (DTB) with 512 SRL buffer.

Even though 4-core DOE achieves one of our goals and slightly outperforms the large baseline core on average, the performance improvement as the number of cores increases from 3 to 8 is not as high as we have hoped, and thus a good area for future research. Only parser and perl benchmarks show significant improvement in performance when the number of cores increases beyond 4.

In order to explain the limited performance scaling as the number of cores increase to 6 and 8 cores, we present in Table III task size statistics and percent of poisoned instructions for a realistic 8-core DOE configuration.

From data in Table III, we observe that the two benchmarks parser and perl, which show the best improvement in performance with 8 cores, have three characteristics that explain their performance: 1) small average task size <1500 instructions, 2) relatively low percentage of poisoned instructions, and 3) large number of forked tasks with high SP value prediction hit rate. The only other benchmark with average task size less than 1500 instructions is bzip2. However, bzip2 does not show good performance improvement with more than 4 cores because of the high percentage of poisoned instructions. Two other benchmarks, gzip and gap have less than 2000 instruction average task size. Neither benchmark performs well with more than 4 cores for different reasons. High SP value misprediction rate causes too many task squashes in gap (Table IV), and high rate of partial stalls and SRL buffer stalls reduce concurrent execution time and consequently overall performance in gzip.

The impact of large tasks and load imbalance on performance is mostly evident in MCF, which has large tasks on average in addition to large variation in task size. MCF

Fig. 13. Four core DOE and conventional SpMT performance.

underperforms the baseline by large amount, as it does not benefit much from DOE execution and suffers significant reduction in performance due to the core reduced buffer size, more so than the other simulated benchmarks due to its very high cache miss rates.

5.9. Task Predictors Results

In this study, we use 4K entry task predictor, 4K entry influence registers mask and RSP value predictor, and 128K 1-bit entry (16 Kbytes) task-sets memory dependence prediction table.

We recall from Section 3.8 that the RSP predictor stores the predicted difference in the values of the RSP register between the forking call instruction and the predicted task. We recall from Section 3.9.3 that the task-sets prediction table is indexed by the task-ID concatenated with load instruction PC. When a load reads a bit value of 1 from the task-set-ID table, the load is predicted to depend on a store from a previous task and is poisoned. When the bit value is 0, the indexing load is predicted to be independent. A load to store intertask dependence sets the bit to 1.

Table IV, columns 2, 3, and 4 show the misprediction rates of the task, SP value, and influence registers predictors respectively. All these mispredictions lead to task squashes. Most of the benchmarks show very low misprediction rates, but there is still potential for improvement, for instance, by using global control flow information, particularly in the case of task prediction.

Table IV, column 5 shows the percentage of tasks squashed due to memory dependence mispredictions. This is a more meaningful metric than memory dependence misprediction rate, since only false negative memory dependence mispredictions require tasks to be squashed, and multiple false negatives could occur in the same task. As can be seen from the table, misprediction rates are quite low, except for task prediction which suffers from task size variability and could be further improved, for instance, by incorporating path information in the prediction mechanism.

5.10. Performance Comparison of DOE and Conventional SpMT

Figure 13 shows the performance of 4-core DOE and 4-core conventional SpMT relative to the large out-of-order baseline core. The DOE and SpMT core configurations are exactly the same as the realistic 4-core DOE configuration discussed above. The key difference between DOE and SpMT is in the way they handle data communication. DOE defers intertask dependent instructions until their input data becomes available and execute forward as long as there are independent instructions to process. On the other hand, the SpMT processor model executes forward until it encounters long intertask dependences that fill its instruction buffers, thus stalling execution.

Table V. Area and Power Estimates for Superscalar and DOE Cores

Machine Configuration	Fetch	Dec	Ren/ DTB	RS	EU	ROB RRF	LSQ SRL LAB	DCache	Total
Superscalar Area	0.17	0.15	0.09	0.07	0.08	0.13	0.15	0.16	1
DOE Core Area	0.045	0.09	0.05	0.02	0.04	0.03	0.05	0.04	0.36
4-Core DOE Area	0.16	0.36	0.20	0.08	0.16	0.12	0.20	0.16	1.44
DOE Core Power	0.11	0.06	0.06	0.026	0.07	0.06	0.07	0.12	0.57
4-Core DOE Power	0.18	0.14	0.11	0.06	0.12	0.11	0.09	0.17	0.98

The graph in Figure 13 shows that DOE, due to its nonblocking execution, outperforms conventional SpMT significantly, by about 15% on average. DOE therefore achieves an equivalent performance to the large out-of-order baseline, while SpMT underperforms the large out-of-order baseline by 15%.

5.11. Comparing Superscalar Throughput Performance to DOE

We reaffirm results reported for single-chip multiprocessors and superscalars by other researchers, [Olukotun et al. 1996]. We have compared throughput performance of 4-core DOE processor, used as a single-chip multiprocessor, to our superscalar baseline processor. We have used for this study a multiprogramming workload consisting of all simulation samples from our spec benchmarks with which we have evaluated DOE single-thread performance. We observe that DOE used as a single-chip multiprocessor outperforms the superscalar baseline on our multiprogramming load by 2x to 2.5x, depending on how the program traces are combined in our multiprogramming workload.

5.12. Power and Area Analysis

In this section, we compare the area and power of our DOE proposal to the large superscalar core. We use for the comparison 4-core DOE configuration.

Our area and power models were derived for a 90 nm CMOS process technology using Cadence tools and Spice circuit simulations. We have used a 90 nm process technology since it is what we have available in our labs. We used a classical 6-transistor SRAM cell design with high-gain differential sense amplifiers for the SRL, DTB and LAB circuits and a single-sided register file cell design proposed by Hsu et al. [2003], augmented with checkpoint and results integration circuit. Our total power estimate for the large superscalar core is 72 watts at 2.5 GHz, which seems reasonable based on power specifications of previous generations of industry superscalar processors, built using 90 nm processes. Even though the process we have used does not represent current state of art, we believe that the relative comparison of DOE to Superscalar, to a large degree, would still hold on current scaled down industry processes.

Our power model accounts for both dynamic and leakage power, and uses execution activity factors derived from our PTLSim architectural performance simulator. We have used Spice simulation to estimate the increase in energy consumption in the circuitry added for DOE, such as, the SRL, the DTB, and the checkpointed register file. In our model, leakage contributes 30% of the total core power.

Table V shows a rollup of relative area and power consumption of various functional blocks in the superscalar baseline, the DOE core, and the 4-core DOE configuration. We report power as an average over all our SpecInt 2000 benchmarks simulation traces. All values in Table V are normalized relative to the total area and power of the baseline core including L1 data and instruction caches.

The area model predicts the single-core DOE area to be 36% of the large superscalar baseline, including the private L1 instruction and data caches. Therefore, the total

4-core DOE area is 1.44 times larger than the superscalar baseline. On the other hand, the power model gives single-core and 4-core DOE power estimate of 57% and 98% of the superscalar baseline power, respectively. Interestingly, the single-thread power consumption of DOE and superscalar is about the same, and thus meets our research goal, even though the 4-core DOE area overhead is large. This is because a program is divided into tasks that are distributed over the 4 DOE cores. Therefore, each individual instruction in a task executes in the significantly smaller and more energy efficient DOE core that is assigned to the task, thus requiring less energy per instruction.

The DOE power model is quite complex due to multiple variables that interact in difficult to predict manner. The following are some of the key factors that impact DOE power consumption which we observed in our analysis:

- The dynamic energy required to execute an instruction is significantly less on the 2-wide, small DOE core having instruction buffers (ROB, RS, LSQ) reduced to one quarter of the buffer sizes of the 4-wide large baseline superscalar. This is because of the large reduction in active capacitance area from smaller number of ports, smaller buffers, shorter signal routes, shorter array bit and word lines, and smaller signal and clock buffers.
- The dynamic energy consumed in the L1 instruction and data caches per instruction is quite smaller on the DOE core, due to the smaller private cache size and reduced associativity.
- We have measured energy waste due to mispredicted branches to be smaller on the narrow and small DOE core than on the large superscalar baseline, by an average of 12%, since the wider and bigger baseline can fetch and execute a larger number of bogus instructions by the time a mispredicted branch is resolved.
- Leakage power is higher on the 4-core DOE due to additional structures (DTB, SRL, LAB, etc. . .) and larger overall area.
- Dependent execution increases dynamic power on DOE due to additional execution mechanisms and structures.
- DOE suffers additional amount of mispredictions due to speculative task execution, hence increasing dynamic power due to bogus execution.
- DOE SRL, DTB and LAB structures contribute relatively low energy overhead, about 1% each, due to their single-ported SRAM design and low activity, as only loads and stores use the LAB and SRL, and only dependent instructions use the DTB. Compare this to the L1 data cache on the large superscalar which contributes a significant 20% of the baseline core total energy, since it simultaneously reads on every access 8 tags and 8 data blocks from 8 ways.
- Our power model estimates the DOE core and the 4-core DOE configuration to be 57% and 98% of the large superscalar baseline. Total DOE power is not proportional to the number of cores, in spite of the linear increase in leakage power as a function of number of cores and energy waste from speculative bogus DOE execution. This is due to the significant reduction in dynamic energy consumed per instruction, since each instruction in the program executes on one and only one of small DOE core.

5.13. Results Summary and Key Insights

We summarize in this section the key insights related to the DOE architecture based on the results of the experiments we have presented above.

1. *4-core DOE better overall energy-efficient performance than a large superscalar.* Latency tolerant DOE execution overcomes intertask data dependences to achieve comparable performance to the large superscalar core for about the same power on single-thread applications that are difficult to parallelize. Moreover, when DOE is

used as a single-chip multiprocessor, it outperforms the superscalar processor by up to 2.5 times on multiprogramming workloads. DOE also outperforms a conventional SpMT by about 15% due to its nonblocking pipeline organization, which minimizes stalls resulting from intertask dependences.

2. *Value prediction is necessary.* DOE can tolerate intertask dependences as long as enough independent instructions are available in the tasks for the continual flow execution of DOE to proceed. Value prediction is necessary to limit the amount of dependences between tasks. As a minimum, stack pointer value prediction is needed to remove dependences created from the use of the stack for subroutine linkage and for allocating temporary procedure variables.
3. *Branch mipredictions affect performance in a fundamentally different way than on conventional superscalar architectures.* Mispredicted branches that are dependent on data from previous tasks are a small subset of the overall set of mispredicted branches. Nevertheless, the branch prediction design optimization process needs to focus on this small subset, since mispredicted dependent branches cause costly task squashes. If necessary, new complex branch prediction methods that may require higher prediction latency could be used in DOE.
4. *DOE is tolerant of long wire delays between cores.* Task dispatch and commit overhead resulting from copying data over global wires between cores can be hidden, as long as a task has enough independent instructions to execute while data is being transferred between cores.

6. RELATED WORK

There has been a large amount of research on Speculative Multithreading. We have cited in previous sections several articles by various researchers. DOE differs from all previous SpMT work in using small latency-tolerant cores. We believe that latency tolerant execution in DOE makes the parallelization process easier, and change design tradeoffs in a significant way that necessitate a new look into SpMT as a high performance option for hard to parallelize conventional single-thread applications.

Methods for spawning tasks include: 1) hardware heuristics methods, such as fork on call, fork on backward branch [Akkary and Driscoll 1998], and fork on loop iterations [Marcuello et al. 1998], 2) hardware prediction methods, such as control independence prediction [Collins et al. 2004], 3) profiling methods, like control quasi-independent points [Marcuello and Gonzalez 2002], 4) rule-based program slicing, such as at trace cache block boundaries [Rotenberg et al. 1997], or at memory load instructions (mem-slicing) [Codrescu and Wills 1999], 5) compiler methods [Chen and Olukotun 1998; Olukotun et al. 1999; Prabhu and Olukotun 2005; Steffan and Mowry 1998; Steffan et al. 2000; Tsai et al. 1998; Tsai et al. 1999], and 6) compiler hints supported with hardware prediction [Franklin and Sohi 1992; Sohi et al. 1995]. All these methods are potential threading extensions for DOE in future work.

A framework of hardware mechanisms for eliminating data dependence squashes through learning, delayed disambiguation, value prediction and synchronization has been proposed by Cintra and Torrellas [2002]. DOE overcomes data dependences by deferring dependent execution to the end of the task, but can benefit from mechanisms such as in Cintra and Torrellas [2002] for highly dependent tasks that encounter partial data stalls.

Value prediction has been used in previous SpMT work to eliminate serialization due to intertask data dependences [Marcuello et al. 1999; Zilles and Sohi 2002; Quinones et al. 2005]. Unlike previous SpMT architectures, DOE defers dependent execution instead of serializing or stalling, but it still can benefit from value prediction to increase the amount of parallelism between tasks. In fact, we use stack pointer value prediction

in this work to increase amount of parallelism between tasks by eliminating artificial dependences created because of the use of the stack for procedure linkage.

Various compiler methods have been proposed for reducing task stalls due to inter-task data dependences. Vijaykumar and Sohi [1998] propose compiler heuristics to aid in a favorable task selection process that reduces data dependences, task overhead and load imbalance. DOE task selection process is simpler since although load balance is critical, task overhead and data dependences are tolerated by deferring dependent execution. Quinones et al. [2005] suggest pre-computing data slices that lead to task input values in order to avoid dependence stalls. Such technique could benefit DOE execution as well, but DOE also has the additional advantage with its ability to overlap the input data slice computation with the execution of the independent task instructions.

Steffan et al. [2002] apply a combination of value prediction, dynamic synchronization and hardware instruction prioritization with automatic compiler transformation to improve thread-level speculation performance. Zhai et al. [2002] use aggressive compiler instruction scheduling techniques to reduce the critical data dependence path and consequently synchronization stalls. DOE overcomes data dependences by deferring dependent execution to the end of the task, but can benefit from such techniques in applications that encounter frequent partial data stalls resulting from highly dependent tasks.

A different approach of using multiple small cores to improve single-thread performance involves fusing/combining multiple cores' hardware resources into the equivalent of a larger superscalar [Ipek et al. 2007; Watanabi et al. 2010; Gibson and Wood 2010]. Unlike this approach, which fundamentally maintains a centralized single-thread execution model, DOE and SPMT use a parallel distributed execution model of single-thread programs, featuring control independent execution and distributed simultaneous instruction fetch and data streams. DOE execution model has been shown in limit studies [Lam and Wilson 1992] to be capable of providing significantly higher level of instruction level parallelism as compared to the superscalar execution model.

Proposals for latency tolerant microarchitectures include Runahead execution [Mutlu et al. 2003], the Waiting Instruction Buffer [Lebeck et al. 2002], Virtual ROB [Cristal et al. 2002], Cherry [Martinez et al. 2002], Checkpoint Processing and Recovery [Akkary et al. 2003], Kilo Instruction Window Processors [Cristal et al. 2005], Continual Flow Pipelines [Srinivasan et al. 2004; Nekkhalapu et al. 2008], Out of Order Commit Processors [Cristal et al. 2004] and Sun's Rock many-core processor [Chaudhry et al. 2008]. All these proposals have been applied to conventional architectures and target single-thread performance on applications that frequently miss on-chip cache thus encountering high off-chip latency. DOE applies latency tolerant execution to SPMT and targets intertask data communication latencies.

7. CONCLUSIONS

We have shown in this work that it may be possible to achieve single-thread performance comparable to large superscalar cores using multiple small DOE cores. Power-hungry large and wide multiported instruction buffers, such as reorder buffers and reservation stations that consume large die area and power on superscalar processor cores are replaced in DOE with high-density single-ported SRAM structures, in which dependent instructions can be deferred, thus avoiding data synchronization stalls of conventional SpMT architectures. We have also shown that DOE outperforms by 15% a similarly configured SpMT processor, and significantly performs better than a large superscalar on multiprogramming workloads, up to 250% better.

We find the preliminary results from this study very promising given that we use a simple heuristic in hardware to dispatch tasks. We believe significant performance gains will be possible with DOE as we implement other threading mechanisms to

help improve the load balance of DOE execution. We have found in our study that benchmarks with balanced task sizes yield significant performance gains even when dependent execution comprises 20% or more of the task execution. This shows the DOE architecture to be quite tolerant to significant amount of intertask dependences.

There is a lot of interesting work to do on DOE. We are currently focusing on building more effective task dispatch mechanisms that yield good load balance and small variations in task size. We are also investigating value prediction opportunities to increase task parallelism and keep dependent execution at low level. Most significantly, we are working on exposing the DOE capability to the compiler and to parallel programming environments to study the full potential of DOE when running newly written applications or existing applications that can be recompiled.

REFERENCES

- AGARWAL, V., HRISHIKASH, S., KECKLER, S. W., AND BURGER, D. 2000. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*. ACM, 248–259.
- AHO, A. V., SETHI, R., ULLMAN, J. D., AND LAM, M. S. 2006. *Compilers: Principles, Techniques and Tools*, 2nd Edition. Pearson Education Inc.
- AKKARY, H. AND DRISCOLL, M. 1998. A dynamic multithreading processor. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*. ACM, 226–236.
- AKKARY, H., RAJWAR, R., AND SRINIVASAN, S. T. 2003. Checkpoint processing and recovery: towards scalable large instruction window processors. In *Proceedings of the 36th Annual ACM/IEEE International Symposium on Microarchitecture*. ACM, 423–434.
- AL-ZAWAWI, A. S., REDDY, V. K., ROTENBERG, E., AND AKKARY, H. 2007. Transparent control independence (TCI). In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA34)*. ACM, 448–459.
- BUMYONG, C., LEO, P., AND TULLSEN, D. M. 2008. Accurate branch prediction for short thread. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 125–134.
- CHANG, P.-Y., HAO, H., PATT, Y. N. 1997. Target prediction for indirect jumps. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA24)*. ACM, 274–283.
- CHAUDHRY, S., CYPHER, R., EKMAN, M., KARLSSON, M., LANDIN, A., YIP, S., ZEPPER, H., AND TREMBLAY, M. 2009. Simultaneous speculative threading: A novel pipeline architecture implemented in Sun's ROCK processor. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*. ACM, 484–495.
- CHEN, M. K. AND OLUKOTUN, K. 1998. Exploiting method-level parallelism in single-threaded Java programs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT '98)*. IEEE, 176–184.
- CHER, C. AND VIJAYKUMAR, T. N. 2001. Skipper: A microarchitecture for exploiting control flow independence. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*. ACM, 4–15.
- CINTRA, M., MARTINEZ, J. F., AND TORRELLAS, J. 2000. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*. ACM, 13–24.
- CINTRA, M. AND TORRELLAS, J. 2002. Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*. IEEE, 43–54.
- CHRYSOS, G. Z. AND EMER, J. S. 1998. Memory dependence prediction using store sets. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*. ACM, 142–153.
- CODRESCU, L. AND WILLS, D. S. 1999. On dynamic speculative thread partitioning and the mem-slicing algorithm. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT '99)*. IEEE, 40–47.
- CODRESCU, L., WILLS D. S., AND MEINDL, J. D. 2001. Architecture of the Atlas chip-multiprocessor: Dynamically parallelizing irregular applications. *IEEE Trans. Comput.* 50, 1, 67–82.
- COLLINS, J. D., TULLSEN, D. M., AND WANG, H. 2004. Control flow optimizations via dynamic reconvergence prediction. In *Proceedings of the 37th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO '37)*. ACM, 129–140.

- CRISTAL, A., VALERO, M., LLOSA, J., AND GONZALEZ, A. 2002. Large Virtual ROB by Processor Checkpointing. Tech. Rep. UPC-DAC-2002-39, Department of Computer Science, Barcelona, Spain.
- CRISTAL, A., SANTANA, O. J., GAZORLA, F., GALLUZZI, M., RAMIREZ, T., PERICAS, M., AND VALERO, M. 2005. Kilo-Instruction Processors: Overcoming the Memory Wall. *IEEE MICRO* 25, 3.
- CRISTAL, A., ORTEGA, D., LLOSA, J., AND VALERO, M. 2004. Out-of-order commit processors. In *Proceedings of the 10th International Symposium on High-Performance Computer Architecture*. IEEE, 48–59.
- CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M., AND ZADECK, F. 1989. An efficient method of computing static single assignment form. In *Proceedings of the 16th Annual Symposium on Principles of Programming Languages*. ACM, 25–35.
- DUBEY, P. K., O'BRIEN, K., O'BRIEN, K. M., AND BARTON, C. 1995. Single-program speculative multithreading (smp) architecture: compiler-assisted fine-grained multithreading. In *Proceedings of the 1995 International Conference on Parallel Architectures and Compilation Techniques*.
- FRANKLIN, M. AND SOHI, G. 1992. The expandable split window paradigm for exploiting fine-grain parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*. ACM, 58–67.
- FRANKLIN, M. 1993. The multiscalar architecture. Ph.D. Dissertation. University of Wisconsin-Madison.
- GANDHI, A., AKKARY, H., RAJWAR, R., SRINIVASAN, S. T., AND LAI, K. 2005. Scalable load and store processing in latency tolerant processors. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*. ACM, 446–457.
- GANDHI, A., AKKARY, H., AND SRINIVASAN, S. T. 2004. Reducing branch misprediction penalty via selective branch recovery. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*. IEEE 254–264.
- GIBSON, D. AND WOOD, D. A. 2010. Forwardflow: A scalable core for power-constrained CMPs. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*. ACM.
- GOPAL, S., VIJAYKUMAR, T. N., SMITH, J. E., AND SOHI, G. 1998. Speculative versioning cache. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*. IEEE.
- HSU, S., CHATTERJEE, B., SACHDEV, M., ALVANDPOUR, A., KRISHNAMURTHY, R., AND BORKAR, S. 2003. A 90nm 6.5 GHz 256x64b dual supply register file with split decoder scheme. In *Proceedings of the Symposium of VLSI Circuits*.
- IPEK, F., KIRMAN, M., KIRMAN, N., AND MARTINES, J. F. 2007. Core fusion: Accommodating software diversity in chip multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. ACM.
- JACOBSON, Q., BENNETT, S., SHARMA, N., AND SMITH, J. E. 1997. Control flow speculation in multiscalar processors. In *Proceedings of the 3th International Symposium on High-Performance Computer Architecture*. IEEE, 218–229.
- KAEHL, D. R. AND EMMA, P. G. 1991. Branch history table prediction of moving target branches due to subroutine returns. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*. ACM, 34–42.
- KROFT, D. 1981. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*. ACM, 81–87.
- LAM, M. S. AND WILSON, R. P. 1992. Limits of control flow on parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*. ACM, 46–57.
- LEBECK, A. R., KOPPANALIL, J., LI, T., PATWARDHAN, J., AND ROTENBERG, E. 2002. A large, fast instruction window for tolerating cache misses. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*. ACM, 59–70.
- MARCUELLO, P. 2003. Speculative multithreaded processors, Ph.D. Dissertation, Universitat Politècnica de Catalunya.
- MARCUELLO, P. AND GONZALEZ, A. 2002. Thread-spawning schemes for speculative multithreading. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*. IEEE, 55–64.
- MARCUELLO, P. AND GONZALEZ, A. 2000. A quantitative assessment of thread-level speculation techniques. In *Proceedings of the 14th International Symposium on Parallel and Distributed Processing*. IEEE, 595–601.
- MARCUELLO, P., GONZALEZ, A., AND TUBELLA, J. 1998. Speculative multithreaded processors. In *Proceedings of the 12th International Conference on Supercomputing*. ACM, 77–84.
- MARCUELLO, P., TUBELLA, J., AND GONZALEZ, A. 1999. Value prediction for speculative multithreaded architectures. In *Proceedings of the 32st Annual ACM/IEEE International Symposium on Microarchitecture*. ACM, 230–236.

- MARTINEZ, J. F., RENAULT, J., HUANG, M. C., PRVULOVIC, M., AND TORRELLAS, J. 2002. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*. ACM, 3–14.
- McFARLING, S. 1993. Combining branch predictors. Tech. Rep. TN-36m. Digital Western Research Laboratory, June.
- MOSHOVOS, A., BREACH, S., VIJAYKUMAR, T. N., AND SOHI, G. 1997. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*. ACM, 181–193.
- MUTLU, O., STARK, J., WILKERSON, C., AND PATT, Y. N. 2003. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*. IEEE.
- NEKKALAPU, S., AKKARY, H., JOTHI, K., RETNAMMA, R., AND SONG, X. 2008. A simple latency tolerant processor. In *Proceedings of the IEEE 26th International Conference on Computer Design*. IEEE, 384–389.
- NICOLAU, A. AND FISHER, J. A. 1984. Measuring the parallelism available for very long instruction word architectures. *IEEE Trans. Comput.* 33, 11, 968–976, November 1984.
- OLUKOTUN, K., HAMMOND, L., AND WILLEY, M. 1999. Improving the performance of speculatively parallel applications on the hydra CMP. In *Proceedings of the 13th International Conference on Supercomputing*. ACM, 21–30.
- OLUKOTUN, K., NAYFEH, B. A., HAMMOND, L., WILSON, K., AND KUNYUNG, C. 1996. The case for a single-chip multi-processor. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM 2–11.
- PAPWORTH, D. B. 1996. Tuning the Pentium Pro microarchitecture. In *IEEE Micro*. 16, 2.
- POSTIFF, M. A., GREENE, D. A., TYSON, G. S., AND MUDGE, T. N. 1999. The limits of instruction level parallelism in Spec95 applications. *ACM Comput. Archit. News* 27, 1.
- PRABHU M. K. AND OLUKOTUN, K. 2005. Exposing speculative thread parallelism in SPEC2000. In *Proceedings of the 10th Symposium on Principles and Practice of Parallel Programming*. ACM, 142–152.
- QUINONES, C. G., MADRILES, C., S'ANCHEZ, J., MARCUELLO, P., GONZALEZ, A., AND TULLSEN, D. M. 2005. Mitosis compiler: An Infrastructure for speculative multithreading based on pre-computation slices. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, 269–279.
- RISEMAN, E. M. AND FOSTER, C. C. 1972. The inhibition of potential parallelism by conditional jumps. *IEEE Trans. Comput.* 21, 12, 1405–1411.
- ROTENBERG, E., JACOBSON, Q., SAZEIDES, Y., AND SMITH, J. E. 1997. Trace processors. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*. ACM, 138–148.
- SOHI, G., BREACH, S., AND VIJAYKUMAR, T. N. 1995. Multiscalar processors. In *Proceedings of the 22th Annual International Symposium on Computer Architecture*. ACM, 414–425.
- SRINIVASAN, S. T., AKKARY, H., HOLMAN, T., AND LAI, K. 2004. A minimal dual-core speculative multithreading architecture. In *Proceedings of the IEEE 22nd International Conference on Computer Design*. IEEE, 360–367.
- SRINIVASAN, S. T., RAJWAR, R., AKKARY, H., GANDHI, A., AND UPTON, M. 2004. Continual flow pipelines. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 107–119.
- STEFFAN, J., COLOHAN, C. B., ZHAI, A., AND MOWRY, T. C. 2000. A scalable approach to thread-level speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*. ACM, 1–12.
- STEFFAN, J., COLOHAN, C. B., ZHAI, A., AND MOWRY, T. C. 2002. Improving value communication for thread-level speculation. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*. IEEE, 36–44.
- STEFFAN J. AND MOWRY, T. C. 1998. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*. IEEE, 2–13.
- TSAI, J.-Y., HUANG, J., AMLO, C., LILJA, D. J., AND YEW, P.-C. 1999. The superthreaded processor architecture. *IEEE Trans. Comput.* 48, 9, 881–902.
- TSAI, J.-Y., JIANG, Z., NESS, E., AND YEW, P.-C. 1998. Performance Study of a concurrent multithreaded processor. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*. IEEE, 24–35.
- VIJAYKUMAR, T. N., AND SOHI, G. 1998. Task selection for a multiscalar processor. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*. ACM, 81–92.
- WATANABI, Y., DAVIS, J. D., AND WOOD, D. A. 2010. WIDGET: Wisconsin decoupled grid execution tile. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*. ACM.

- YOURST, M. T. 2007. PTLSim: A cycle accurate full system x86-64 microarchitectural simulator. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS '07)*, IEEE, 23–34.
- ZHAI, A., COLOHAN, C. B., STEFFAN, J. G., AND MOWRY, T.C. 2002. Compiler optimization of scalar value communication between speculative threads. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 171–183.
- ZILLES, C. AND SOHI, G. 2002. Master/slave speculative parallelization. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*.

Received July 2010; revised August 2011; accepted May 2012