Federation: Boosting Per-Thread Performance of Throughput-Oriented Manycore Architectures

Michael Boyer, David Tarjan, and Kevin Skadron

Manycore architectures designed for parallel workloads are likely to use simple, highly multi-threaded, in-order cores. This maximizes throughput, but only with enough threads to keep hardware utilized. For applications or phases with more limited parallelism, we describe creating an out-of-order processor on the fly, by *federating* two neighboring in-order cores. We reuse the large register file in the multi-threaded cores to implement some out-of-order structures and re-engineer other large, associative structures into simpler lookup tables. The resulting federated core provides twice the single-thread performance of the underlying in-order core, allowing the architecture to efficiently support a wider range of parallelism.

Categories and Subject Descriptors: C.1.2 [**Multiple Data Stream Architectures**]: Multiple-instruction-stream, multiple-data-stream processors (MIMD)

General Terms: Design, Performance

Additional Key Words and Phrases: Federation, CMP, multicore, out-of-order

1. INTRODUCTION

Increasing difficulties in improving frequency and instruction-level parallelism have led to the widespread adoption of multicore processors. When designing such a processor, there is a fundamental tradeoff between the complexity or capability of each individual core and the total number of cores that can fit within a given area. For applications with sufficient parallelism, Davis et al. [2005] and Carmean [2007] show that maximum aggregate throughput is achieved by using a large number of highly multi-threaded scalar cores. However, for applications with more limited parallelism, performance would be improved with a smaller number of more complex cores.

How can these two approaches be reconciled? To improve the single-thread performance of an existing throughput-oriented system, one approach would be to add dedicated out-of-order (OOO) cores to the existing scalar cores. Certainly a dedicated OOO core will give great performance on *one* thread, and provisioning a single OOO core is a sensible solution to deal with the Amdahl's Law problem

Author's address: M. Boyer, Computer Science Dept., University of Virginia, 151 Engineer's Way Box 400740, Charlottesville, VA 22904-4740, {boyer, dtarjan, skadron}@cs.virginia.edu.
D. Tarjan is now with NVIDIA Research, Santa Clara, CA.

posed by serial portions of a parallel program or a single, interactive thread.[1] Unfortunately, dedicated OOO cores come at the cost of multiple scalar cores, reducing the aggregate throughput of the system. Using an even larger core with simultaneous multi-threading (SMT) would still limit the throughput and/or increase overall power. This is because the area efficiency of OOO cores, even with SMT, is lower than that of multi-threaded in-order cores [Davis et al. 2005].

Instead, we propose *Federation*, a technique that allows a pair of in-order cores to be combined at runtime to form a higher performance OOO core, retaining a significant fraction of the performance benefit of the dedicated core with a much smaller area overhead. We achieve this by re-engineering the major hardware structures required for out-of-order execution for much lower complexity and power by replacing content addressable memories (CAMs) and broadcast networks with simple lookup tables. This makes it possible for manycore processors to offer competitive single-thread performance without incurring the major area and power overhead of a dedicated high-performance core.

1.1 Contributions

In this work, we first describe how to take two minimalist, scalar, in-order cores that have no branch prediction hardware and combine them to achieve two-wide, OOO issue. We then show that the area-efficient structures used by Federation can be used to construct a *lightweight* dedicated OOO core. Finally, we show that Federation, with some small adaptations, can be extended to dual-issue in-order cores, enabling the construction of a 4-way federated OOO core.

The main contributions of this paper are:

- We show how to build a minimalist OOO processor from two in-order cores with less than 2KB of new hardware state and only a 5.7% area increase over a pair of scalar in-order cores, using simple SRAM lookup structures and no major additional CAM structures. Whereas a traditional 2-way OOO core costs 2.65 scalar cores in die area, Federation can synthesize a 2-way OOO core in the area of 2.11 scalar cores, while still retaining the ability to use the scalar cores independently for high throughput.[2]

  We show that despite its limitations, such an OOO processor offers enough performance advantage over an in-order processor to make Federation a viable solution for effectively supporting a wide variety of applications. In fact, relative to a traditional OOO organization of the same width, the two-way federated organization provides nearly the same performance, is competitive in energy efficiency, and has better area-energy-efficiency.
- We introduce the Memory Alias Table (MAT), which provides approximately the same functionality and performance as the Store Vulnerability Window [Roth 2005] while using an order of magnitude fewer bits per entry.

[1]We see this approach embodied in the Sony Cell [Hofstee 2005] and AMD Fusion [Hester 2006].
[2]The total area comparison (2.65 vs. 2.11 scalar cores) underemphasizes the area advantage of Federation. For example, assume we wish to add a higher-performance OOO core to an existing throughput-oriented architecture composed of many scalar cores, without reducing the number of scalar cores. Adding a dedicated core would require additional area equal to 2.65 scalar cores. Adding a federated core, however, would only require additional area equal to 0.11 scalar cores.

- We show that completely eliminating the forwarding logic between loads and stores is a viable option for a small OOO core.
- We show that using these area- and power-efficient structures allows the construction of a lightweight dedicated 2-way OOO core.[3]
- We show that it is possible to extend Federation to 2-way in-order cores and achieve performance close to a dedicated 4-way OOO core.

Federated cores are best suited for workloads that usually need high throughput but sometimes exhibit limited parallelism. Federation provides faster, more energy-efficient cores for the latter case without sacrificing area that would reduce thread capacity for the former case.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 outlines our approach to Federation and explains some fundamental assumptions made in our design. Section 4 discusses how an in-order pipeline can be adapted to support OOO execution and the new structures required. Section 5 presents the details of our Memory Alias Table design. Section 6 discusses a dedicated OOO core using the same area- and energy-efficient principles as Federation. Section 7 describes the details of our simulation environment and benchmarks. Section 8 presents performance, power, and area-efficiency results. Section 9 extends Federation to more complex baseline cores. Section 10 concludes the paper.

2. RELATED WORK

This paper extends previous work by Tarjan, et al. [2008] by discussing in greater detail: the low-level operation of the OOO pipeline, the Memory Alias Table, the performance and area impact of the new structures, and the overall performance of the federated core. It also adds a discussion of a dedicated lightweight out-of-order core that employs the same area-efficient principles as Federation and an evaluation of the benefits of federating 2-way in-order cores.

The Voltron architecture from Zhong et al. [2007] allows multiple in-order VLIW cores of a chip multiprocessor (CMP) to combine into a larger VLIW core. It requires a special compiler to transform programs into a form that can be exploited by this larger core. The performance is heavily dependent on the quality of the code the compiler generates, as the hardware cannot extract fine-grained instruction parallelism from the instruction stream by itself. The work on composable cores [Kim et al. 2007] leverages the block-level dataflow EDGE ISA [Burger et al. 2004] and associated compiler [Smith et al. 2006] to allow multiple small cores to work on a single instruction stream, without requiring traditional out-of-order structures such as a rename table or issue queue. Our work does not assume an advanced compiler and is applicable to RISC, CISC, and VLIW ISAs.

Salverda and Zilles [2008] explore the performance limits of a design that contains a number of in-order lanes or pipelines that can be fused at run-time to achieve out-of-order execution. Their work assumes a "slip-oriented out-of-order execution model," in which out-of-order execution only occurs when the individual lanes slip

[3]Note that the lightweight OOO core is not a viable solution to the problem of phases with limited thread count, because a dedicated OOO core of any size comes at considerable area cost in terms of the number of high-throughput scalar cores that are replaced.

with respect to one another. In other words, within each lane, instructions always execute in-order. The performance constraints shown in their work are only valid for machines that utilize this execution model. Federation is *not* based on the slip-oriented out-of-order execution model. When in-order pipelines are federated, instructions can be issued out-of-order to any pipeline and thus instructions within the same pipeline can execute out-of-order with respect to one another. This approach raises some scaling issues of its own but frees Federation from the fundamental constraints of the slip-oriented model.

Work by Kumar et al. [2004] on heterogeneous cores showed that the overall throughput of a heterogeneous CMP can be higher than an area-equivalent homogeneous CMP, if the OS can schedule threads to different types of cores depending on their needs. However, because the mix of large and small cores has to be set at design-time, the OS or hypervisor cannot dynamically make a tradeoff at runtime between the number of cores (i.e., the number of thread contexts) and single-thread latency. Grochowski et al. [2004] follow up on this line of work and observe that the combination of performance- and throughput-oriented cores with dynamic voltage scaling can provide a better combination of single-thread latency and throughput than either technique can provide alone.

Adjoining cores that are federated have their caches merged when in federated mode, similar to work by Kumar et al. [2004] and Dolbeau and Seznec [2004]. However, we do not require two cores to be able to access the same cache simultaneously, since only one core's load and store ports are active when in federated mode.

Numerous groups have evaluated various combinations of clustered OOO processors and multi-threading. İpek et al. [2007] provide a comprehensive overview of this body of work. Another approach to improve the single-thread performance is to use runahead execution [Chou et al. 2004; Mutlu et al. 2003], which is orthogonal and even complementary to federating two simple cores. Runahead reduces time spent waiting on cache misses, which would potentially help the more powerful federated core relative to the underlying scalar core. Additionally, federating two cores would help the runahead thread run faster and thus further ahead of the main thread.

The Store Vulnerability Window (SVW) was introduced by Roth [2005] as a verification mechanism for load/store aliasing and ordering that can be used in conjunction with several load speculation techniques. The Memory Alias Table is a similar structure to the SVW, but uses much less hardware. More recent work [Sha et al. 2005; 2006; Subramaniam and Loh 2006] has tried to largely or completely eliminate the Load-Store Queue (LSQ) by using the SVW as the checking mechanism for speculative forwarding, which we avoid due to its complexity. Our work differs in that we do not try to replace a part of an OOO processor, but instead augment a simple in-order processor so that it can detect memory order violations with minimal hardware cost. We also do no speculative forwarding; indeed, we abandon forwarding completely in our design.

Comparison to Core Fusion

The work on Core Fusion [İpek et al. 2007] provides an interesting comparison point to Federation. Core Fusion and Federation employ very different approaches to the problem of how to aggregate smaller cores into a single, higher performance core.

Core Fusion aims to build an OOO core with a very deep execution window and lots of execution resources. To achieve this goal, Core Fusion combines a larger number of cores (up to four cores) than Federation (two cores). Due to the complexity of the extra structures needed for Core Fusion and the latency required to communicate between several cores at multiple locations in the pipeline, Core Fusion must increase the length of many of the critical loops of the processor pipeline. Federation employs almost exactly the opposite approach, focusing on aggregating fewer, smaller cores and placing an emphasis on NOT increasing any of the critical loops of the pipeline unless absolutely necessary. The choice of a centralized Issue Queue and centralized MAT stem directly from trying to avoid such overheads. We believe that the large body of work on clustered architectures show convincingly that distributing the critical structures of an OOO core only makes sense if the workload exhibits large amounts of ILP and few serializing conditions such as branch mispredictions and memory aliasing events; conditions that are not true for many applications that are not easily decomposed into multiple threads and thus need higher single-thread performance the most.

In both the work by İpek et al. [2007] and this paper, the performance of the aggregated core is compared to that of a dedicated 4-way OOO core. Of course, directly comparing the results is difficult, since the dedicated cores in the two comparisons are configured differently and use different simulation methodologies. Nevertheless, Core Fusion of four 2-way OOO cores achieves about 102% and 115% of the performance of a dedicated 4-way core on SPECint and SPECfp, respectively. We show in Section 9.3 that Federation of two 2-way in-order cores achieves 92% and 86%, respectively, of the performance of a dedicated 4-way OOO core, with half the execution resources and much lower power. Thus, even with much simpler baseline cores, Federation is able to achieve performance that is competitive with Core Fusion.

Comparing the areas of the aggregated cores is not necessarily useful, since one can assume that a manycore processor will have more than enough cores for any aggregation technique. Comparing the area overhead of the aggregation techniques and the area efficiencies of the baseline cores is more instructive. The area overhead of Core Fusion is estimated to be 8.64mm^2 from a 200mm^2 die with 100mm^2 devoted to core area, or about 8.6% of the core area. Using scalar in-order cores as a baseline, we estimate in Section 8.2 that the area overhead of Federation is approximately 5.7% of each pair of cores, and thus 5.7% of the total core area regardless of the number of cores. Using 2-way in-order cores with branch prediction as a baseline, the area overhead is much smaller, since the majority of the area overhead of federating scalar cores was due to the addition of a small branch predictor.

For phases of execution in which the thread count is high, a manycore processor implementing either Core Fusion or Federation will be best off without any cores fused/federated in order to provide as many hardware thread contexts as possible. In such a case, Federation's multi-threaded in-order baseline cores will provide much higher aggregate throughput than Core Fusion's 2-way OOO baseline cores because of their significantly higher area efficiency. Carmean [2007] estimates that a multi-threaded in-order core takes up only one-fifth the area of a traditional core while providing more than 20 times the throughput per unit area. Thus, Core Fusion will provide superior performance when the thread count is extremely low. For medium to high thread counts, however, the higher throughput of the underlying cores in Federation will provide significantly higher performance.

3. BASELINE ARCHITECTURE

Future microprocessor designs will likely incorporate many simple in-order cores rather than a small number of complex OOO cores [Asanovic et al. 2006]. Current examples of this trend include the Sun Niagara I and II [Johnson and Nawathe 2007; Kongetira et al. 2005], each of which contain up to eight cores per processor. At the same time, graphics processors (GPUs), which traditionally consist of a large number of simple processing elements, have become increasingly programmable and general purpose [Owens et al. 2007]. The most recent GPU designs from NVIDIA [NVIDIA 2009] and AMD [AMD 2007] incorporate 240 and 320 processing elements, respectively. This so-called *manycore* trend will provide substantial increases in throughput but may have a detrimental effect on single-thread latency. Federation is proposed to overcome this limitation.

When designing a federated processor, there are two possible approaches: design a new processor from the ground-up to support Federation or add Federation capability to an existing design. For the purposes of this paper, we will take the latter approach and add Federation support to an existing manycore, in-order architecture. Based on the current trends cited above, the baseline in-order microarchitecture that we will focus on is similar to Niagara. It is composed of multiple simple scalar in-order cores implementing the Alpha ISA, which are highly multi-threaded[4] to achieve high throughput by exploiting thread-level parallelism (TLP) and memory-level parallelism (MLP) [Glew 1998]. Specifically, each in-order core has four thread contexts, with hardware state for 32 64-bit integer registers and 32 64-bit floating point registers per thread context. Additionally, the integer and floating point register files are banked, with one bank per context and two read ports and one write port per bank. Unlike Niagara I (but like Niagara II), each core in our baseline architecture has dedicated floating point resources. To deal with multi-cycle instructions such as floating point instructions and loads, the in-order core has a small (four-entry) completion buffer. This buffer is used both to maintain precise exceptions and to prevent stalling when a multi-cycle instruction issues. The in-order cores implement only static not taken branch prediction[5] and use a branch address calculator (BAC) in the decode stage to minimize fetch bubbles and to conserve ALU bandwidth.

To simplify the discussion, in this paper we focus on a single pair of in-order cores that can federate to form a single OOO core. In practice, the techniques we describe are intended to be applied to a manycore processor with a significantly larger number of cores, with each adjacent pair of cores able to federate into a single OOO core.

[4]But as Table VIII shows, the area overhead of multi-threading is not very large and Federation is thus an attractive option even for single-threaded cores.

[5]Niagara I provides no branch prediction whatsoever while Niagara II employs static not taken. We chose the more complex of these two options in order to give the scalar cores a slight boost in single-thread performance.

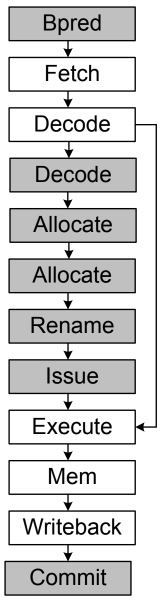


Fig. 1. The pipeline of a federated core. New pipeline stages are shaded.

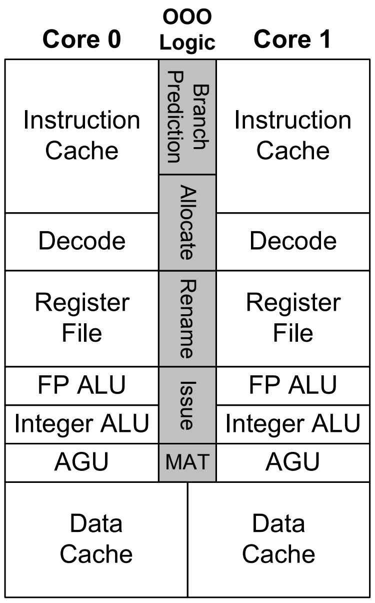


Fig. 2. A simplified floorplan showing the arrangement of two in-order cores. New structures necessary for Federation are shaded.

The process of federating and de-federating a pair of cores is controlled by the OS. Although the software implications of Federation are beyond the scope of this paper, we discuss them briefly here. After deciding to federate a pair of cores, the OS must interrupt them, wait for outstanding memory operations to complete, save the state of the cores' threads as necessary, flush one of the core's instruction and data caches to ensure that no duplicated data exists, indicate to the hardware to federate the cores, and finally set up the state of the thread that will run on the new federated core. De-federating cores is similar, although no cache flushing is required. Perhaps more interesting than determining *how* to federate and de-federate is deciding *when* to do so. One straightforward approach would be to federate cores when a high-priority thread needs a performance boost or when the number of runnable threads drops below the number of thread contexts that would be available after federating, and to de-federate when the number of runnable threads becomes larger than the number of thread contexts. More complex approaches may prove beneficial, but are left for future work.

4. OUT-OF-ORDER PIPELINE

The primary goal of Federation is to add OOO execution capability to the existing in-order cores with as little area overhead as possible. Thus, each federated OOO core is relatively simple compared to current dedicated OOO implementations. Specifically, each federated core is single-threaded[6] and two-way issue with a 32-

[6]Thus when the two in-order cores federate, the number of thread contexts provided by the pair of cores is reduced from eight to one.

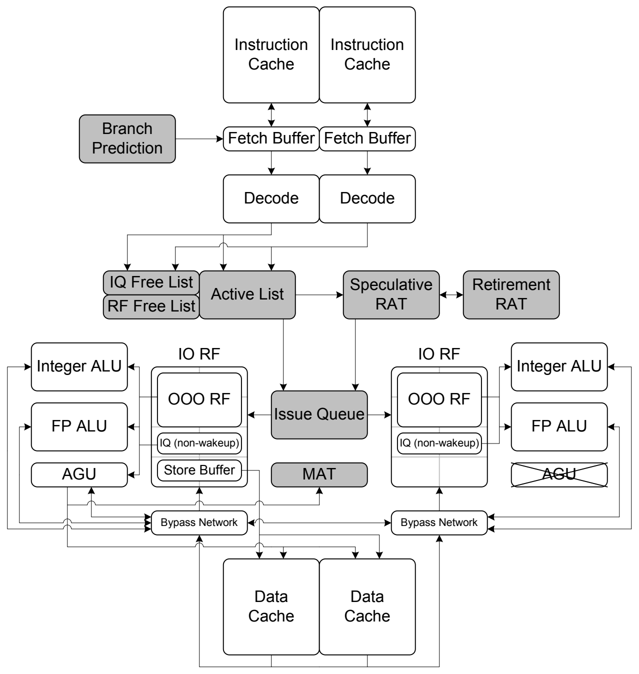


Fig. 3. Structural diagram of a federated core. New structures are shaded. The unified register file, a portion of the issue queue, and the store buffer are mapped onto multiple banks in the existing in-order register file. For simplicity, not all connections are shown, and connections not used in federated mode, such as connections to the address generation unit (AGU) in the second core, are intentionally omitted.

entry instruction window. The federated core implements the pipeline shown in Figure 1, with the additional pipeline stages not present in the baseline in-order cores shown in shaded boxes. A possible high-level floorplan for the federated core is shown in Figure 2. A structural diagram of the OOO pipeline is shown in Figure 3.

In order to limit the area overhead of Federation, we strive to avoid adding any significant CAMs or structures with a large number of read and write ports. Table I lists the sizes of the new structures required to support OOO execution, as well as whether or not each structure is implemented by re-using the existing hardware from the large, banked register file of the underlying multi-threaded core. While an extremely area-conscious approach could use the register file to implement all of the new structures, this would excessively increase the complexity and wiring overhead of the design, as well as increasing the contention for register ports. The structures that reuse the register file in our design are those that are close to the register read and writeback stages in the pipeline, require few read and write ports, and are read and written to at sizes close to those that the register file already supports.

The major new wiring required to support Federation is listed in Table II. The following subsections provide a detailed explanation of the operation of each pipeline stage, along with justification for the design tradeoffs that were made.

Structure	**Size (bits)**	**Type**	**Reuses RF**
Branch Predictor (NLS)	6,656	SRAM	No
Branch Predictor (Bimodal)	4,096	SRAM	No
Return Address Stack	256	SRAM	No
Rename Tables	1,152	Reg	No
Free Lists	384	Reg	No
Active List	4,096	Reg	No
Issue Queue (Wakeup)	176	Reg	No
Issue Queue (Non-Wakeup)	896	Reg	Yes
Unified Register File	6,144	Reg	Yes
Store Buffer	1,024	Reg	Yes
Memory Alias Table	96	Reg	No
Worst Case Total (Bits)	11,008 SRAM / 13,968 Register		
Assumed Total (Bits)	11,008 SRAM / 5,904 Register		

Table I. Estimated number of bits required for the new structures added to the baseline in-order processor. Type differentiates between: 6T SRAM cells, which are used for caches and large tables; and registers, which are used for building the smaller structures inside the pipeline, have full swing bitlines, and are potentially multiported. The last column indicates whether we assume the structure can be built using only reused register file entries if the baseline core is multi-threaded. The worst case total is calculated under the assumption that none of the structure can reuse the register file.

New Wiring	**Width (bits)**
Cross I-Cache to Decode	32
Decode to Allocate	64
Cross Core Value Copying	140
Memory Unit to Second D-Cache	128

Table II. The size of major wires that must be added to the baseline core in order to support Federation.

4.1 Branch Prediction

Branch prediction is implemented using Next Line and Set (NLS) prediction [Calder and Grunwald 1995; Kessler et al. 1998; Tremblay and O'Connor 1996] instead of a branch target buffer. NLS maintains an untagged table indexed by the branch address, with each entry pointing to a line in the instruction cache. NLS predicts the location in the cache where the next line will be fetched rather than the actual address to be fetched. This significantly reduces the overhead of supporting NLS. For example, implementing a 512 entry NLS requires only about 0.75 KB of extra state. A small return address stack (RAS) is also added, which requires only 256 bits of state. We have omitted the top 32 bits of the return addresses and assume they do not change; no negative performance impact on our workload is observed. Branches are resolved during the commit stage, as described in Section 4.10.

4.2 Fetch

The fetch stage starts by receiving a predicted cache line from the NLS predictor, a return address from the RAS, or, in the case of a misprediction, a corrected PC from the branch unit in the execute stage. It then initiates the fetch by forwarding this information to the instruction cache (IC). The ICs of the two cores are combined into a cache with double the associativity and random replacement. We assume that each core's L1 cache has a 3-cycle latency for a hit. We further assume that the clock frequency is low enough and there is enough slack in the cache access time that, when federated, the combined cache still has a 3-cycle hit latency. Each in-order core has a fetch buffer, which holds the most recently accessed cache line. Fetching two instructions thus requires reading two instructions from one of the fetch buffers, rather than increasing the width of the cache. Since each core can only decode a single instruction, the second instruction is sent to the other core for decoding. So that this extra wire does not influence cycle time, we allocate an extra pipeline stage for copying the instruction to the other core, buffering the first instruction in a pipeline register.

4.3 Decode

Once the instructions have been received from the fetch stage, the separate decode units in the two cores can operate independently. The decoded instructions are then routed to the allocate stage. If the first of the two instructions is a taken branch, a signal is sent to the allocate stage to ignore the second decoded instruction. Since the allocate unit is a new structure located between the two cores, propagating the instructions to it in the same pipeline stage as decode or allocate might influence overall cycle time. We instead allocate an extra pipeline stage (the first Allocate stage in Figure 1) to allow the signals from both decode units to propagate to the allocate unit. The performance implications of the extra front-end pipeline stages are discussed in Section 8. The BAC of one of the baseline cores is used to calculate and verify the target of any taken branch.

4.4 Allocate

During the allocate stage, each instruction checks for space in several structures required for OOO execution. All instructions check for space in both the Issue Queue (IQ) and the Register File (RF). In traditional OOO architectures, load and store instructions would also need to check for a free Load-Store Queue (LSQ) entry, but our implementation uses a Memory Alias Table, which is free from such constraints (see Section 5). If space is not available in any of the required structures, the instruction (and subsequent instructions) will stall until space becomes available.

The allocate stage maintains two free lists, one for the IQ and one for the unified register file, with both lists implemented as new structures. We decided against using existing register file entries to implement these free lists because of their early position in the pipeline, the small size of each entry, and the complexity of deciding which entries to add to or remove from the free list. This complexity means that only a fraction of a clock cycle is available for the actual read/write operation. In addition to the free lists, the allocate stage also maintains the current Active List (AL) head and tail pointers so that it can determine if there is space available in the AL and then assign an AL entry to the current instruction(s).

4.5 Rename

The federated core uses a unified register file, in which integer and floating-point instructions share a pool of physical registers, with speculative and retirement Register Alias Tables (RATs). Since the design utilizes a subscription-based instruction queue (see Section 4.6), it must keep track of the number of subscribers for each instruction. For each architected register, its status and the number of consumers currently in the IQ is stored in a second table, which is accessed in parallel with the RAT.

Each rename table for a two-way OOO processor requires four read ports and two write ports, while each existing register bank has only two read ports and one write port. Thus, implementing the rename tables using the existing register files would require the exclusive use of two entire register banks. Given the relatively small amount of state stored in the rename table, it makes sense to implement it as a separate structure.

The unified register file consists of the 64 architected registers (32 integer and 32 floating-point) and a number of rename registers, implemented using the register files of the underlying multi-threaded cores, with each register stored in both cores simultaneously. As mentioned earlier, the existing register files are heavily banked. The unified register files use part of several of these banks in order to support the required number of read and write ports. Even so, it is still possible for a particular register access pattern to require more writes to a single bank than that bank can support. Additional logic detects this case and causes one of the two instructions to stall. The performance impact of bank contention is explored in Section 8.

Logic is needed to check for read after write (RAW) dependencies between two instructions being renamed in the same cycle. Additional logic is also necessary to check for race conditions between an instruction being renamed and an instruction that generates one of its input operands being issued in the same cycle. This logic checks whether the status of one of the input operands is changing in the same cycle as its status is being read from the rename table. This classic two ships passing in the night problem is also present in many in-order processors, where instructions that check the poison bits of their input operands must be made aware of any same-cycle changes to the status of those operands. Thus, depending on the design of the baseline in-order core, it might be possible to reuse this logic for the OOO processor. We assume that this capability is not supported by our baseline in-order core and that it must be introduced from scratch.

Because branches are only resolved at commit time, there is no need to checkpoint the state of the RAT for every branch. If a branch misprediction or another kind of exception is detected, the pipeline is flushed and a bit-vector (one bit associated with each RAT entry) is reset to indicate that the most up to date version of all registers is in the non-speculative RAT. During normal execution, an instruction in the rename stage writing to a particular register sets the corresponding bit to indicate that the speculative version is the most up-to-date. During rename an instruction will either first access the bit-vector to determine the correct source for the rename lookup, or access both speculative and non-speculative RAT and do a late select based on the state of the corresponding bit. Which option to use would be dependent on both power and timing tradeoffs.

Note that the per RAT-entry bit-vector indicating how many consumers an entry in the IQ has is reset whenever a new value is written to an architected register. If the RAT-entry indicates that the instruction producing a value has already executed and/or is non-speculative, then the consumer bit-vector can be ignored.

Each entry in the active list holds the logical register mapping as well as the old and new physical register mappings for an instruction. This data is needed at commit time in order to update the register file free list and the retirement RAT.

4.6 Issue

As with the register file, the federated core uses a unified Issue Queue (IQ) that tracks both integer and floating-point instructions. The area and power constraints of our design prevent the implementation of a traditional CAM-based IQ. To avoid tag broadcast or tag match logic, we use a simple table in which consumers "subscribe" to their producers by writing their IQ position into one of the producer's IQ entry's consumer fields, similar to a number of prior designs [Brekelbaum et al. 2002; Huang et al. 2002; Raasch et al. 2002; Ramírez et al. 2004; Sato et al. 2001]. Huang et al. showed that, for a processor with a 96-entry instruction window, over 90% of all dynamic instructions have no more than one dependent instruction in the instruction window when they execute [2002]. Thus, each IQ entry in our design only has a small number of consumer fields. The exact number of consumer fields per entry is a design choice; we found that limiting the number of fields per entry to two reduced performance by only a fraction of percent compared to a traditional IQ. The performance impact of this decision is evaluated in greater detail in Section 8. Note that the designers of the TRIPS dataflow ISA made a similar decision, limiting each instruction to at most two consumers [Sankaralingam et al. 2006].

Each entry in the IQ holds the usual opcode, operand register IDs, and immediate/displacement values, but also has several consumer ID fields and two ready bits, which are set when the left and right operands become available, respectively. On issue, each instruction checks its consumer fields and sets the appropriate ready bits in the consumer's entry. If both input operands are ready, the ready signal for that entry is sent to the scheduler.[7] Because the opcode and immediate/displacement value are not required for the critical wakeup and select loop, they are stored in a table physically separate from the ready bits and the consumer IDs (referred to as the non-wakeup portion of the IQ). These fields are mapped onto the underlying register file to conserve area.

The table holding the ready bits, while logically part of one larger structure, is implemented separately since it is a combination of a state table with a larger number of write ports than the other parts of the issue queue, per-entry logic for checking that both input operands are ready, and logic to reset the ready bits when an instruction is issued. The table does not require any read ports, since each entry's values, after being gated by resource constraints as described by Sassone et al. [2007], are simply ANDed and routed to a priority encoder. When an instruction issues, it must set the ready bits of its consuming instructions. Thus, for each issue port, the table needs two write ports. When an asynchronous unit completes, it must set the ready bits of its consuming instructions. Thus, for each asynchronous unit, the table also needs two write ports. Overall, the table holding the ready bits has 8 write ports for a table with $2N$ bits (where N is the size of the issue queue). The table holding the consumer IDs needs two read ports, since two instructions

[7]Note that all loads can issue speculatively, without waiting on unresolved stores; see Section 5 for an explanation.

can issue each cycle, and two write ports, since two instructions can be allocated each cycle.

Since the number of consumer fields is small, an instruction can stall if its producer's consumer entries are all occupied. This necessitates the addition of an extra bit to each entry in the IQ that is set if the instruction is oversubscribed. If this bit is set when the instruction executes, a signal is sent to the rename stage to unstall the waiting instruction(s).

The normal scheduling logic for an out-of-order processor tries to issue older instructions first, which makes it complex and power hungry. We instead implement a simpler pseudo-random scheduler [Sassone et al. 2007] that uses a static priority encoder and does not take into account the age of different instructions. For a small out-of-order window, this simplified scheduler has a negligible impact on performance, as shown in Section 8.

In addition to favoring the oldest instructions, schedulers for clustered architectures often attempt to schedule consuming instructions on the same cluster as their producers in order to avoid the overhead of copying the result between clusters. Given that our design maintains a copy of each register value on both cores, the core on which a consuming instruction is scheduled is only relevant in the case where it is ready to be issued as soon as its producer has issued. We again choose the simplest design, scheduling all instructions on core zero when possible and only assigning an instruction to core one when a previous instruction has been assigned to core zero that cycle.

In order to prevent a producer-consumer pair from getting scheduled back-to-back on different cores, when an instruction is issued, the ID of the core on which it executes is buffered in the scheduling logic of its consuming instructions' ready bits. This information is treated as a resource constraint, as described above, so that the consuming instruction will not be scheduled on the other core. To avoid maintaining memory ordering across the two cores, loads and stores are only assigned to core zero.

4.7 Execute

Each instruction executes normally on the ALU to which it was assigned, operating on values read out from the local core's register file during the issue stage. The only change to the bypass network on each core is the addition of circuitry for copying the result to the register file of the other core. Since this is not a zero-cycle operation, the new circuits can be added without affecting the critical path. Additionally, a benefit of using the subscription-based IQ is that we know during execution whether it is necessary to broadcast the result using the bypass network, based on whether or not any consumers have subscribed to the instruction.

4.8 Memory Access

The data caches are merged in the same way as the instruction caches, by having each cache hold half the ways of a merged cache with twice the associativity. Instead of a traditional load-store queue, our design uses a simpler structure called a Memory Alias Table (MAT). We do not allow memory bypassing and flush the pipeline when we detect a load and store accessing the same address out-of-order. A detailed description of the MAT is provided in Section 5.

The only additional action required of load instructions in this stage is to index into the MAT with their target address and increment a counter. Store instructions write their effective address and data into an entry in the store buffer. Because we do not support store-to-load forwarding, loads never access the store buffer.

4.9 Write Back

Similar to the Alpha 21264 [Kessler et al. 1998], all results are written to the register files on both cores, to avoid the complication of having to generate explicit copy instructions for consumers on the other core. Depending on the layout, copying the results from one core to the other may incur one or more extra cycles of latency. We assume a one-cycle latency in the results discussed in Section 8.

4.10 Commit

Branches are resolved at commit time, obviating the need to maintain multiple snapshots of the speculative rename table or the need to walk the AL in the case of a branch misprediction. We have also explored the performance impact of using a limited number of branch checkpoints. However, since our focus is on simplicity, our base case for the federated core still uses commit time branch recovery, which reduces performance by approximately 3%. Adding just two snapshots of the rename table would almost completely eliminate this overhead, but we show results for the simplest case.

Each baseline core has a post-commit write combining buffer in order to save data cache bandwidth, which conceptually is considered part of the data cache. At commit, store instructions copy their data from the store buffer to the write combining buffer. As in the baseline cores, a load instruction accessing the cache also accesses the write combining buffer.

5. MEMORY ALIAS TABLE

Traditional Load-Store Queues (LSQs) are used for enforcing correct ordering between loads and stores that can potentially execute out of program order, and to forward values between aliasing loads and stores. They have large CAMs for address matching, circuitry for age prioritization in case of multiple matches, and a forwarding network. All these structures would add considerable power and complexity to our baseline processor. Instead, we propose the Memory Alias Table (MAT), which builds on ideas from the Store Vulnerability Window (SVW) [Roth 2005] and work by Garg et al. [2006] and Onder and Gupta [1999].

5.1 Store-to-Load Forwarding

Of the two main purposes of a traditional LSQ, only one is necessary to ensure correctness: the enforcement of proper ordering between loads and stores accessing the same memory address. The forwarding of values from stores to loads is a performance optimization and can be omitted without impacting correctness, as long as a load cannot execute until after any aliasing stores have committed.

To estimate the potential performance impact of omitting this functionality, we first measured the number of forwarding events on a processor configuration that does support forwarding. We measured the data for all of the benchmarks in the SPEC2000 suite. The simulated processor is a 2-wide out-of-order machine

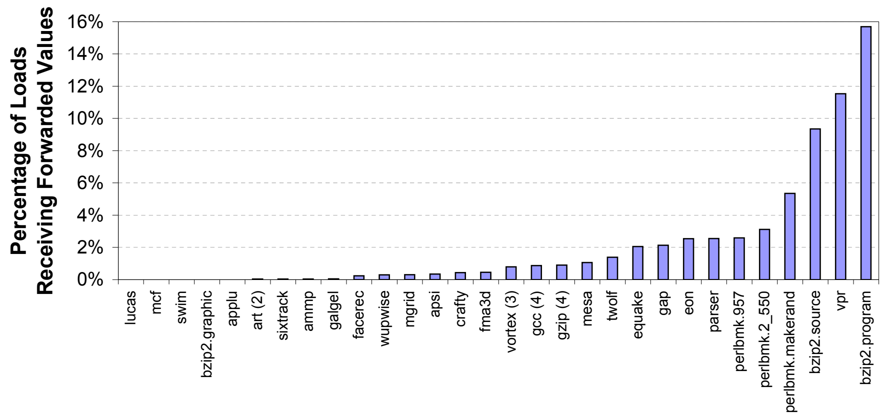


Fig. 4. Fraction of load instructions that receive their values directly from an earlier store instruction. Benchmarks with multiple reference inputs that exhibit similar behavior have been averaged; the number of inputs for those benchmarks is indicated in parentheses.

implementing the Alpha instruction set; the specific configuration is described in greater detail in Table III. The details of our simulation environment are described in Section 7.

The percentage of loads that receive a value forwarded from a store is shown in Figure 4. To decrease the number of data points and improve readability, we present the average across multiple reference inputs for those benchmarks that exhibit similar behavior for all inputs. For over half (23 out of 39) of the benchmark and reference input pairs, fewer than one percent of loads receive their values forwarded from a store. Only three benchmarks exhibit a percentage greater than five: bzip2 (two of three inputs), perlbmk (one of three inputs), and vpr. On average, only 1.54% of loads receive their values from a store.[8]

This data alone presents an incomplete picture, because some benchmarks contain many more load instructions than others. In other words, the importance of forwarding hardware is dependent on how frequently store-to-load forwarding occurs. Figure 5 shows the forwarding period, defined as the average number of clock cycles between store-to-load forwarding events. Once again, for those benchmarks that exhibit similar behavior across all reference inputs, we present the average across the inputs. Note that the period is undefined for the benchmarks lucas, mcf, and swim, because they exhibit no forwarding events. More than half (22 out of 39) of the benchmark and reference input pairs have a forwarding period greater than 500 cycles. Only three benchmarks have a period less than 100 cycles: bzip2 (two of three inputs), perlbmk (one of three inputs), and vpr. On average, a forwarding event occurs once every 399 cycles.[9]

[8]This average was computed by first averaging across all reference inputs for those benchmarks with multiple inputs, and then averaging across all benchmarks. This ensures that all benchmarks are assigned equal weight.

[9]This average was computed by averaging the period across all reference inputs for those benchmarks with multiple inputs, converting from cycles per event to events per cycle (because the period of some of the benchmarks is undefined), averaging across the entire suite, and then con-

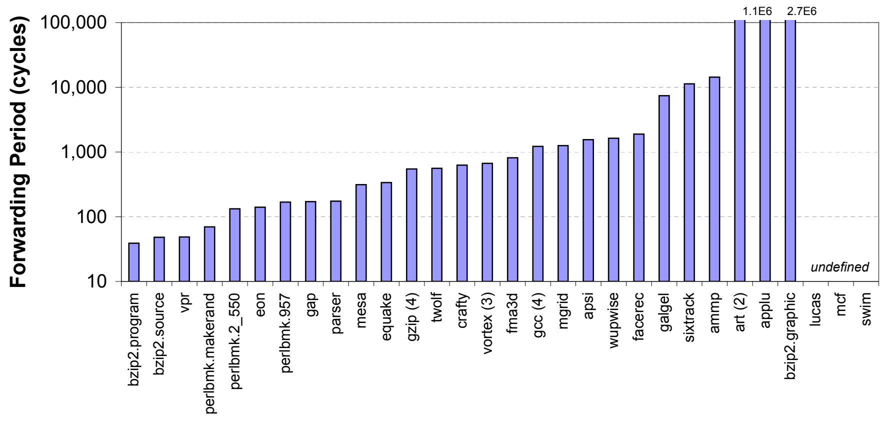


Fig. 5. Average number of cycles between store-to-load forwarding events. Benchmarks with multiple reference inputs that exhibit similar behavior have been averaged; the number of inputs for those benchmarks is indicated in parentheses. Note the log scale on the y-axis.

Motivated by the infrequency of store-to-load forwarding, the MAT does not provide a mechanism for forwarding store results to loads, eliminating the need for a forwarding network that can deal with multiple (partial) matches. Instead, it only detects memory order violations after they have occurred and causes the offending instruction and all subsequent instructions to be re-executed. This provides considerable area savings with minimal performance loss, as we will show in Section 8. Garg et al. [2006] also found store-to-load forwarding to be an infrequent occurrence and omitted forwarding hardware from their design.

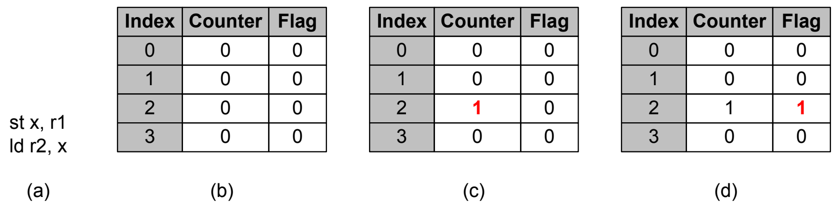
Unlike in work by Sha et al. [2006], we do not implement a load-store alias predictor, but statically predict all loads and stores to not alias. A dynamic predictor is necessary for a large, high-performance design, where store-to-load forwarding would be needed to exploit the available machine resources, but can be omitted from our small design.

5.2 Concept of the Memory Alias Table

Conceptually, the MAT operates as follows: each load places a token in an address-indexed hash table when it executes and removes the token when it commits. Each store checks the hash table at commit for a token from a load that is still in the pipeline. Any store finding a valid token when it commits knows that the token is from a potentially aliasing load that executed out-of-order and signals a memory order violation. The store does not need to cause an immediate pipeline flush but instead sets an exception flag in the table when it commits. The offending load will discover this exception flag during commit when it invalidates its token in the hash table. The load can then either replay or cause a pipeline flush.

The implementation of the MAT is relatively simple: each load increments a counter when it executes and decrements the same counter when it commits. Stores check the MAT only when they commit. Since any committed load will have re-

verting back to cycles per event.



```
st x, r1
ld r2, x
```

(a)

Index	Counter	Flag
0	0	0
1	0	0
2	0	0
3	0	0

(b)

Index	Counter	Flag
0	0	0
1	0	0
2	1	0
3	0	0

(c)

Index	Counter	Flag
0	0	0
1	0	0
2	1	1
3	0	0

(d)

Fig. 6. Example of the MAT's operation for the sequence of instructions shown in (a), with the load executing before the store. The MAT is shown (b) in its initial state, (c) after the load executes, and (d) after the store commits.

moved any sign of its presence from the MAT before a store reaches commit, the store knows that if its counter in the MAT is non-zero, there must be at least one potentially aliasing load in the pipeline that executed out-of-order with respect to the store. Previous proposals had the store check their equivalent of the MAT as soon as the address generation for the store was complete, rather than at commit. They thus had no way of telling if an aliasing load had executed out of program order; they could only determine that it had executed earlier.

Figure 6 shows an example of the MAT detecting and recovering from a memory order violation. The instructions being executed are shown in program order in Figure 6(a). Since the store writes to the same memory location from which the load reads, in order to guarantee correct execution the store must commit and write to the data cache before the load reads from the data cache. In this example, however, the load instruction executes out-of-order, before the store instruction. Initially, all of the counters and exception flags in the MAT are set to zero, as shown in Figure 6(b). During execution, the load indexes into the MAT using the low-order bits of its memory address[10] and increments the corresponding counter, as shown in Figure 6(c). When the store commits, it also indexes into the MAT using the low-order bits of its memory address and checks if the corresponding counter is zero. Since the counter is non-zero, the exception flag for that MAT entry is set, as shown in Figure 6(d). When the load attempts to commit, it indexes into the MAT once again, observes that the exception flag is set, and initiates a flush of the pipeline. The processor then resumes execution at the offending load instruction, with all counters and exception flags in the MAT set to zero.

The hash table proposed by Garg et al. [2006] utilizes the same basic concept as the MAT, while the SVW [Roth 2005] inverts the relationship between loads and stores, with stores leaving tokens in a table and loads checking the table for valid aliasing entries. A critical distinction between the MAT and these previous proposals is how instruction age is represented in hardware. Previous proposals used a store sequence number (SSN) or a load sequence number (LSN) to determine relative age. Since it is non-trivial to determine when the last load vulnerable to a store committed, a counter representing dynamic instruction age was used. This required relatively large entries and the comparison of 16-bit or larger values to determine the relative ages of a load and a store. The scheme presented by Onder

[10]Loads and stores are treated as though they access 64 bit (8 byte) values, since this is the largest operand size in the Alpha ISA.

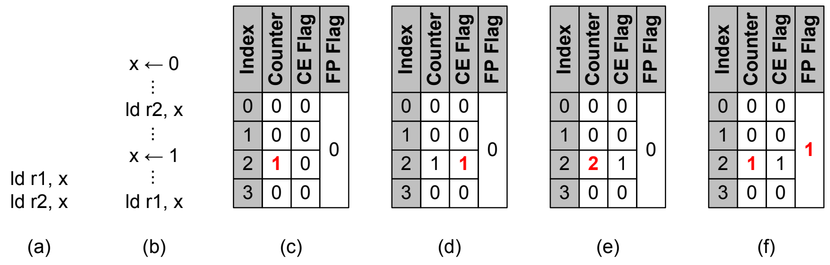


Fig. 7. Example of the MAT's operation for the sequence of instructions shown in (a), with events occurring in the sequence shown in (b). The MAT is shown (c) after the load to r2 executes, (d) after the coherence event (x ← 1) occurs, (e) after the load to r1 executes, and (f) after the load to r1 commits. Note that the coherence exception (CE) flag shown here is different than the exception flag set by store instructions; for simplicity, we omit the store exception flag here.

and Gupta [1999] is similar to the MAT in that loads access a table after executing and stores check that same table for conflicting entries at commit. However, their implementation is significantly more complicated, because each load stores in the table its full address as well as the value it reads, and each store must search the table for a matching address and compare the value read by the load with its own value. Another proposal by Sethumadhavan et al. [2003] used simple counting bloom filters, but could not determine the relative age of a load or store.

Since our proposal relies on the precision of the counters in the MAT for correctness, the number of bits in each counter must equal the logarithm of the size of the AL. Note that because our design does not have a separate LSQ structure, the whole AL can be filled with loads and/or stores. Even for much larger instruction windows than we discuss here, the size of the counters is still much smaller than the 16 bits required to store the SSN[11] in work by Sha et al. [2006]. Moreover, multiple counters can share a single set of higher-order bits (with only the LSB private to each counter), further reducing the amount of storage required per entry without impacting correctness (proof omitted). The sharing of the upper bits can be considered the inverse of sharing the LSB in certain branch predictor tables [Seznec et al. 2002]. We show in Section 8 that sharing all but the LSB between multiple counters is feasible, as it introduces very few extra false positive memory order violations.

5.3 Dealing with Coherence

To ensure correct execution in the presence of cache coherence, the MAT must ensure that no load get the wrong value, even if it initially executed out of program order. Two loads from the same location can be out of order with respect to each other as long as no change to that location occurs between the two accesses. To ensure this property, any cache coherence transaction indexes into the MAT and sets

[11]The SSN can be smaller than 16 bits, but since overflowing the SSN counter requires a pipeline flush and a reset of the hash table, a smaller SSN leads to lower performance in a traditional processor. Federation, however, can most likely use smaller SSNs without significantly impacting performance, because the SSN counter can be reset after each branch misprediction when the pipeline is guaranteed to be empty. The performance results for the SVW presented in Section 8 are in the context of a traditional, dedicated out-of-order core and thus use 16-bit SSNs.

a coherence exception flag (distinct from the exception flag set by store instructions) for its entry if the corresponding counter is non-zero. Any load that maps to the same MAT entry that has executed but not committed when this occurs will force a flush of the pipeline when it tries to commit.

Any load committing in the same cycle as the cache coherence event can ignore it, since it is assured to have received its value before the event. Any committing load that decrements the counter to zero can reset the coherence exception flag, since no loads that have already received their values and have this location as their target are in the window any longer. To ensure forward progress, the first load to see the coherence exception flag at commit can still commit, since it cannot have received the wrong value in any combination of events. This load sets a separate forward progress flag (shared across the whole table) to indicate that later loads are not the first to have seen the coherence exception flag. All flags are reset at pipeline flushes.

An example of the MAT's operation in the context of coherence events is shown in Figure 7. The instructions being executed are shown in program order in Figure 7(a). The shared variable x initially has the value zero. As shown in Figure 7(b), the second load (to register r2) executes before the first load and receives the original value of x (zero). The load also increments the counter in the MAT associated with the address of x, as shown in Figure 7(c). Before the first load executes, a remote core updates the value of x to one, which causes the value in the local cache to be updated as well. When this update occurs, the MAT is accessed and, because the counter associated with the address of x is non-zero, the coherence exception flag for that entry is set, as shown in Figure 7(d). At some later time, the first load executes and also increments the counter in the MAT, as shown in Figure 7(e). When the first load attempts to commit, it sees the non-zero value of the coherence exception flag and thus checks the table-wide forward progress flag. Since this flag is zero, the load commits successfully but also sets the flag to one, as shown in Figure 7(f). When the second load attempts to commit, it also sees the non-zero value of the coherence exception flag and checks the forward progress flag. Seeing that this flag is also non-zero, the pipeline is flushed and all counters and flags in the MAT are reset. Execution resumes with the second load instruction.

6. BUILDING A LIGHTWEIGHT OUT-OF-ORDER CORE

In the previous sections, we have discussed how to build an OOO core with low power and area overhead from relatively simple multi-threaded in-order cores. For a system that aims for a different balance between single-thread performance and aggregate throughput, the choice of a baseline core may be between larger, higher performance in-order cores, or small and low power OOO cores. Implementing a dedicated OOO core using the same area-efficient structures employed by Federation allows designers to build a *lightweight* OOO core with competitive performance and lower power relative to a traditional OOO core.

Compared to a federated core, a dedicated lightweight core has some important advantages. Since it is designed from the ground up for OOO execution, there is no need for extra pipeline stages in the frontend for copying values between cores. The number of ports in the underlying register file is no longer a limiting factor, so

Parameter	**Scalar IO**	**Federated IO**	**2-way OOO**	**4-way OOO**
Active List	-	-	32	128
Issue Queue	-	-	16	32
Load-Store Queue	-	-	16	64
Integer ALUs	1	2	2	4
FPUs	1	2	2	4
Data Cache	8KB	16KB	16KB	32KB
Instruction Cache	16KB	32KB	32KB	32KB
Unified L2 Cache	256KB	256KB	256KB	2MB
Branch Target Buffer	-	512 (NLS)	512	4K
Direction Predictor	not-taken	2K bimodal	2K bimodal	16K tournament
Memory	100 Cycles, 64-Bit			

Table III. Simulator parameters for the different core types. The federated and lightweight cores have the same sized resources as the dedicated 2-way core. Note that the federated and lightweight cores use an MAT instead of an LSQ, and thus the number of loads and stores is limited by the size of the Active List rather than the size of the LSQ.

there is no possibility of bank conflicts occurring. Also, because it does not use two separate register files from two distinct cores, there is no need for an extra cycle of latency for writing values to both register files. The biggest advantage of the lightweight core, however, is that adding multiple rename table checkpoints is more easily achieved, making it feasible for branch recovery to occur before the commit stage.

In the following sections, we will evaluate such a lightweight OOO core that can fetch, decode, issue and retire two instructions per cycle. For systems that need higher performing cores or are so area constrained that adding a traditional high-performance OOO core is not an option, adding a lightweight core that is small, low-power, and still relatively fast might be an attractive alternative.

7. SIMULATION SETUP

We evaluate our design using a simulator based on the SimpleScalar 3.0 framework [Burger et al. 1996] with Wattch extensions [Brooks et al. 2000]. For the OOO cores, our simulator models separate integer and floating point issue queues, load-store queues and active lists, and assumes that the scheduler receives the hit/miss signal from the cache one cycle before the data payload [Skadron et al. 2003]. The pipeline has been expanded from the 5-stage pipeline of the baseline simulator to faithfully model the power and performance effects of the longer frontend pipelines. When simulating the MAT, our simulator allows loads to issue in the presence of unresolved stores. In the case that a memory order violation occurs, the pipeline is flushed when the offending load attempts to commit.

Wattch has been modified to model the correct power of the separately sized issue queues, load-store queues and active lists. Additionally, we accurately model the power of misspeculation in the active lists. Static power has been adjusted to be 25% of maximum power, which is closer to recently reported data [Mesa-Martinez et al. 2007].

We use the full SPEC2000 suite with reference inputs compiled for the Alpha instruction set. The Simpoint [Sherwood et al. 2002] toolkit was used to select representative 100 million instruction traces from the overall execution of all SPEC2000

Subscriber Slots	Change in IPC
1	-0.55%
2	-0.25%
4	+0.05%
8	0.00%

Table IV. Impact on harmonic mean IPC of the number of subscriber slots in the subscription-based IQ, relative to a traditional IQ.

benchmarks. For each run the simulator was warmed up for 10 million instruction to avoid startup effects. When presenting averages across the entire benchmark suite, we weigh all benchmarks equally by first taking the average across the multiple reference inputs for those benchmarks that have them.

The federated core is compared against five other cores: the baseline scalar, in-order core from which the federated core is built; a 2-way in-order core, designated federated in-order, built from two scalar cores; the lightweight 2-way OOO core; and traditional, dedicated 2-way and 4-way OOO cores. The simulation parameters for the different cores are listed in Table III. Although the in-order cores are highly multi-threaded, the simulations run only a single thread, since this represents the best case for single-thread latency. Note that the smaller L2 for the small cores represents a single tile of a much larger L2, to simulate the fact that these cores will not be the only cores active on the chip and thus do not have exclusive use of the whole L2.[12]

8. RESULTS

We first present results from sensitivity studies of the changes to the major structures introduced earlier. To isolate the performance impact of each feature and to avoid artifacts due to clustering, we evaluate each feature separately in the traditional, dedicated 2-way OOO core.

Table IV shows that restricting the number of subscription slots in each IQ entry has very little impact on overall performance. We attribute this to the fact that the majority of dynamic instructions have only a single consumer [Butts and Sohi 2002], and that only a fraction of those consumers are in the IQ at the same time as their producers. Based on these results, each entry in the federated core has two subscription slots. Figure 8 shows the scaling behavior of the subscription-based IQ compared to a traditional IQ, as well as the impact of using pseudo-random scheduling instead of oldest-first scheduling. The impact of both changes is very small for all configurations, which is in agreement with previous work [Sassone et al. 2007]. The largest combination of IQ and AL shows only a 1.5% difference in absolute performance between the best and worst configurations.

The use of the MAT allows most loads to execute earlier than they would have with a traditional LSQ, but at the cost of additional pipeline flushes due to both true memory order violations and false positives from the limited size of the hash table. Figure 9 shows the performance of the baseline core using either a MAT,

[12]We also simulated all cores with a 32MB L2 cache and verified that while absolute performance improves by about 20%, this occurs across the board, so that the relative performance between the federated and the dedicated OOO cores changes by less than 0.9%

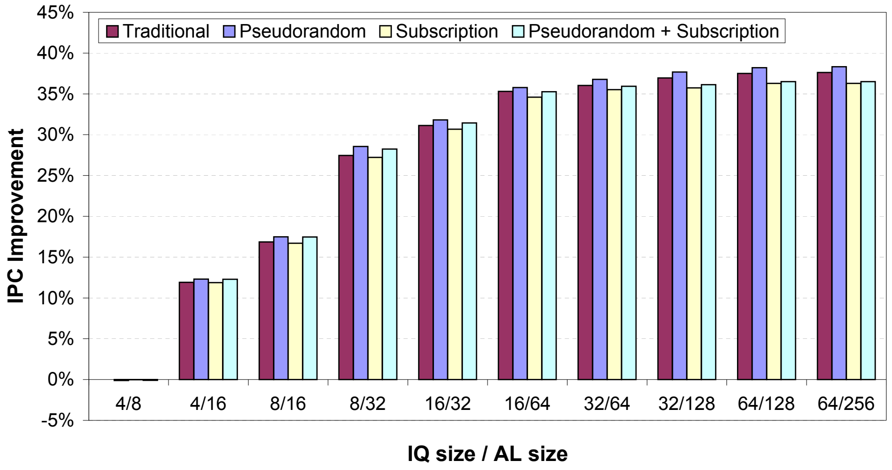


Fig. 8. Relative increase in harmonic mean IPC for different IQ designs as the sizes of the IQ and the AL are increased. By default, designs use oldest-first scheduling and a CAM-based IQ. Designs labeled "Pseudorandom" instead use pseudo-random scheduling and designs labeled "Subscription" instead use a subscription-based IQ. The percent improvement in IPC is relative to the "Traditional" configuration with a 4-entry IQ and an 8-entry AL.

Sharing Degree	Change in IPC
2	0.00%
4	-0.04%
8	-0.14%
16	-0.37%

Table V. Impact on harmonic mean IPC of sharing the higher order bits of counters in the MAT.

SVW, or LSQ. As the sizes of of the hash tables are increased, the false positives are reduced and essentially only the true memory order violations remain. Note that since both the SVW and the MAT place no restrictions on the number of loads and stores in the pipeline, even a 1-entry SVW or MAT can have as many loads simultaneously in flight as there are AL entries. The MAT and SVW have almost exactly the same performance and both use much less hardware than the LSQ. As each entry of the SVW is 16 bits and each entry in the MAT is only 6 bits, the MAT provides the best performance for a given amount of hardware. Since we would need a 16-entry LSQ to outperform even the smallest MAT, the tradeoff of hardware overhead versus performance is a very favorable one.

As discussed in Section 5.2, the MAT can save even more hardware by sharing most bits of each counter among neighboring entries in the hash table. Table V shows the impact on performance as we increase the number of counters sharing one set of upper bits. Since the performance impact is minimal, for the federated core we share one set of upper bits between eight entries. Thus, for a 32-entry instruction window, each MAT entry only uses $1 + \frac{4}{8}$ bits for the counter and an additional $\frac{1}{8}$ bit for the shared exception bit (ignoring the hardware required for coherence support).

Figure 10 shows the impact on performance of the individual design changes of the

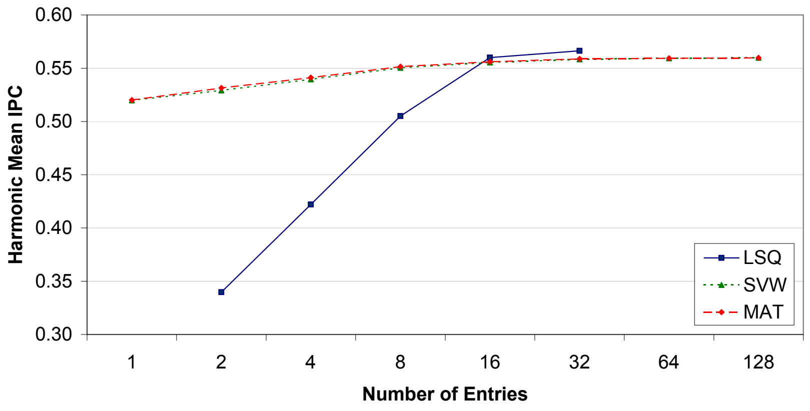


Fig. 9. Scaling of harmonic mean IPC for LSQ, SVW, and MAT as the number of entries is increased. The lines for the SVW and MAT are almost indistinguishable.

federated core. Using the dedicated two-way OOO core as a baseline, each energy- or complexity-reducing feature is enabled individually to show its (negative) impact on overall performance. Most of the individual limitations have only a very small effect on performance; commit-time branch recovery causes the largest decrease in harmonic mean IPC at 3.2%.

Figure 11 shows the cumulative performance impact of these same design changes. Moving from left to right, each feature is enabled and remains on as subsequent features are enabled. Note that the cumulative effect of these changes (6.9% reduction in harmonic mean IPC) is greater than the sum of their individual contributions (4.0% reduction).

To separate out the impact of those features that might be applied to a traditional OOO core from the extra constraints imposed by federating two scalar cores, the three constraints that are a direct consequence of combining two distinct cores (increased fetch latency, register bank contention, and clustered ALUs) are shown on the left of the figures. As described in Section 6, the lightweight OOO core differs from the federated OOO core in that it does not suffer from these three limitations, and also does not wait until the commit stage to recover from branch mispredictions.

8.1 Other Points in the Design Space

The design chosen for the federated core represents only one point in a whole spectrum of possible designs. We have aimed for a balance between minimizing extra area and maximizing performance, but would also like to discuss some alternative design choices using the techniques we have presented that either provide greater area savings or increased performance. Commit time branch prediction recovery has a large negative performance impact on our design. The design tradeoff here would be to limit the number of unresolved branches in the AL at any given time and add a small number of shadow rename maps, which are saved on each branch and restored on a branch misprediction, to allow OOO branch recovery at writeback. Our experiments (not shown) reveal that adding only two shadow rename maps

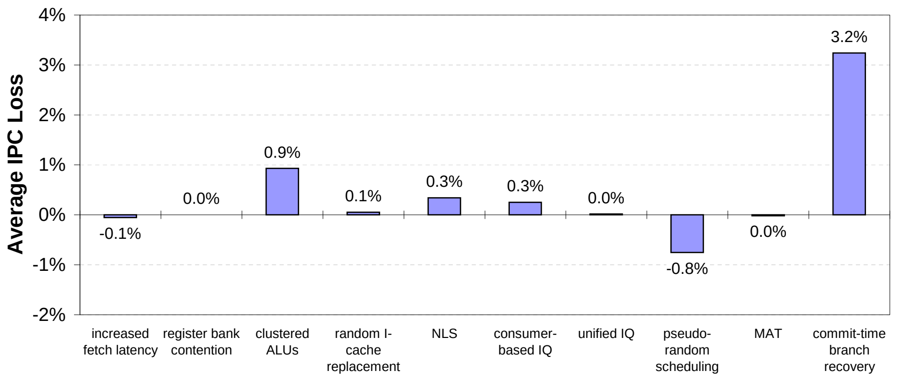


Fig. 10. Reduction in harmonic mean IPC due to each individual feature or limitation of Federation. The baseline design is the dedicated two-way OOO core.

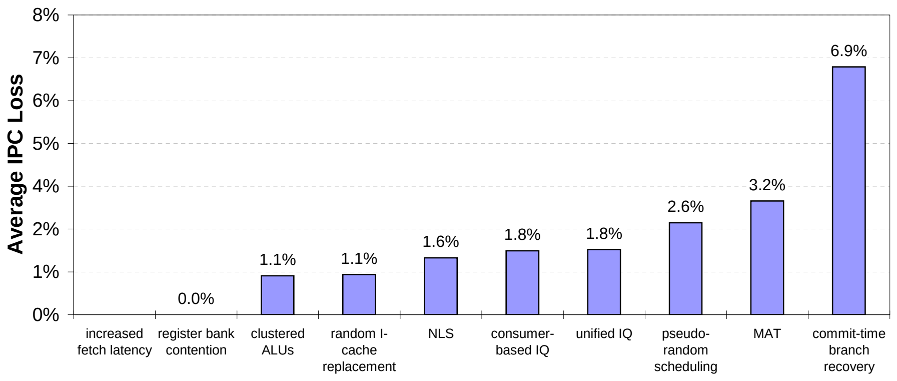


Fig. 11. Cumulative reduction in harmonic mean IPC as each feature/limitation of Federation is enabled and left on, moving from left to right. The baseline design is the dedicated two-way OOO core. The rightmost bar represents the performance of the federated core.

(768 register bits overhead) provides most of the benefit of OOO branch recovery and results in 5.1% better performance than the normal federated core.

One of the biggest additional structure of the federated core is the NLS branch predictor. To save even more space, we considered moving branch prediction from the fetch stage to the decode stage and only using a way predictor, reducing the number of bits in each NLS entry to the logarithm of the number of ways in the instruction cache. The target of direct branches would be calculated using the BAC, which is used to verify branch targets in all designs, and the NLS predictor would only predict which way of the set to read from the instruction cache. The most common indirect branches (returns) would be predicted by the RAS; however, the core would have to stall on other indirect branches. Using the way predictor would preserve the power savings due to reading out only one way during most cycles, but reduce the size of the NLS from 6,656 to 1,536 bits. While the performance impact

Structure	Size (bits)	Area (mm^2)
Branch Predictor (Direction Table)	4,096	0.017
Branch Predictor (Target Table)	6,656	0.031
Return Address Stack	256	0.001
Rename Tables	1,152	0.019
Free Lists	384	0.006
Active List	4,096	0.068
Instruction Queue (Wakeup)	176	0.023
Memory Alias Table	96	0.001
Inter Core Wires	-	0.052
Total	16,912	0.2185

Table VI. Estimated area of extra structures required for Federation.

Pipeline Stage	Area (mm^2)	
	Old	New
Branch Prediction	-	0.049
Fetch	1.307	1.307
Allocate/Rename	-	0.026
Issue	-	0.023
Execute	1.425	1.450
Writeback	1.097	1.122
Commit	-	0.069

Table VII. Area of each pipeline stage of a pair of in-order cores before and after adding Federation. The Fetch and Writeback stages include the instruction and data caches, respectively. The active list and MAT are considered part of the Commit stage.

Core Type	Area (mm^2)
1-way in-order	1.739
1-way in-order MT	1.914
Federated 2-way OOO	4.047
Lightweight 2-way OOO	3.946
2-way OOO	5.067
4-way OOO	11.189
Federated 4-way OOO	8.897
8KB L1 Cache	0.305
16KB L1 Cache	0.609
32KB L1 Cache	1.219

Table VIII. Estimated area of different core types and cache sizes in 45nm technology. Core areas include L1 instruction and data caches. The federated 4-way OOO core is described in Section 9.

of moving branch prediction to the decode stage is only 0.5% on average, stalling on non-return indirect branches affects some programs significantly.

8.2 Area Impact of Federation

Estimating the sizes of the different core types and the area overhead of Federation is a difficult task, and we can only provide approximate answers without actually implementing most of the features of the different cores in a specific design flow. To estimate realistic sizes for the different units of a core, we measured the sizes of the different functional units of an AMD Opteron processor in 130nm technology from a publicly available die photo. We could only account for about 70% of the total area, the rest being x86-specific, system level circuits, or unidentifiable. We scaled the functional unit areas to 45nm, assuming a 0.7 scaling factor per generation. The sizes of the different cores were then calculated from the areas of their constituent units, scaled with capacity and port numbers.[13]

The area of the federated core was calculated by adding the areas of all the major new functional units, shown in Table VI, to the area of two scalar in-order cores.

[13]We do not account for the area of the decode logic, because decoding x86 instructions is significantly more complicated than decoding Alpha instructions. This hurts Federation slightly, because its relative area overhead appears larger than it would be if we accounted for decode area.

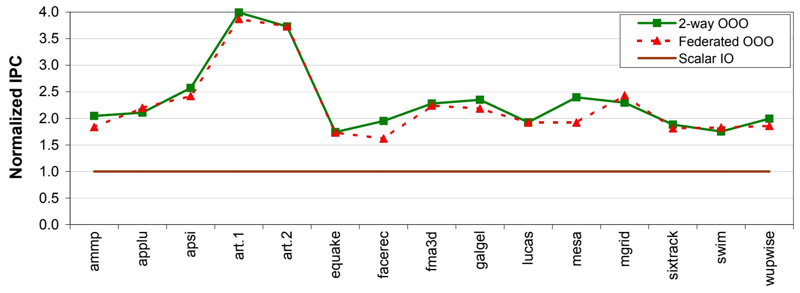


Fig. 12. IPC of the dedicated 2-way OOO core and the federated OOO core on the floating point benchmarks from the SPEC2000 suite, normalized to the IPC of the baseline in-order core.

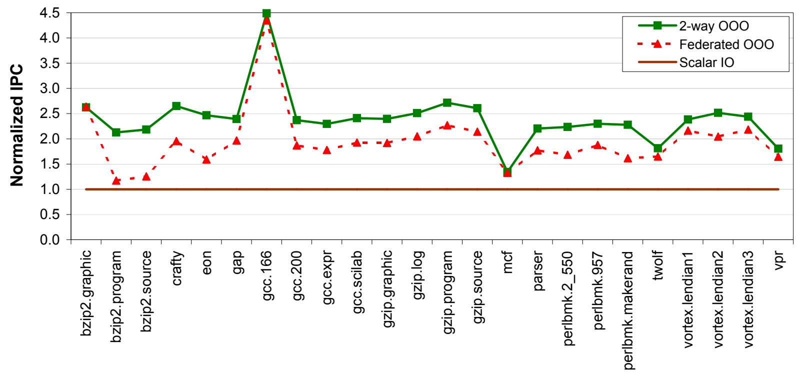


Fig. 13. IPC of the dedicated 2-way OOO core and the federated OOO core on the integer benchmarks from the SPEC2000 suite, normalized to the IPC of the baseline in-order core.

We estimated the area needed by the major inter-core wiring listed in Table II by calculating the width of the widest new unit (the integer and floating point rename tables laid out side-by-side) and using the same 280nm wire pitch as used by İpek et al. [2007]. In contrast to their work, which has a significant amount of extra area devoted to new inter-core wires, the area used by the wires for federating two cores is only 0.052mm^2, since the wires do not have to cross over multiple large cores, but only connect two immediately adjacent small cores. The estimated area of each pipeline stage before and after adding Federation capability is shown in Table VII, and the total area of each different core design is shown in Table VIII.[14] Based on these estimates, the area overhead of adding Federation to a pair of multi-threaded in-order cores is only 5.7%.

8.3 Overall Performance and Energy Efficiency Impact of Federation

The performance of the federated OOO core relative to the baseline in-order core and the dedicated 2-way OOO core is shown in Figures 12 and 13 for the floating

[14]It is interesting to note that the ratio of the area of the 4-way OOO core to the area of the in-order core is close to the 5-to-1 ratio found by Carmean [2007], even though our assumptions and baseline designs are somewhat different.

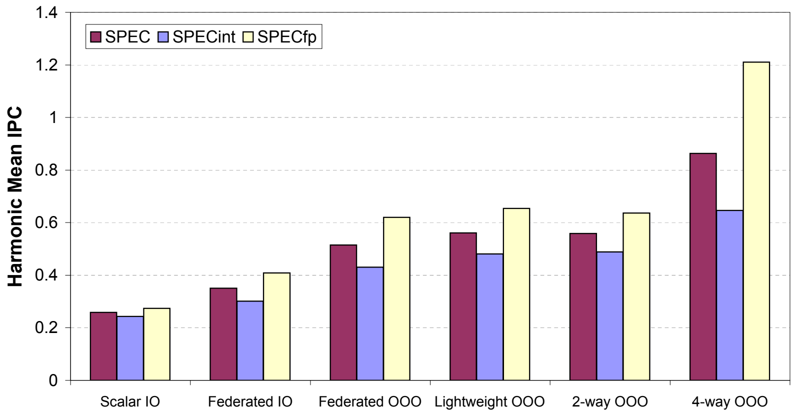


Fig. 14. Harmonic mean IPC.

point and integer benchmarks in the SPEC2000 suite, respectively. The performance of the federated core is very similar to that of the dedicated OOO core for the floating point benchmarks. The federated core actually provides slightly higher performance on three benchmarks (swim, mgrid, and applu), most likely due to the MAT allowing loads to execute earlier than in a traditional design. Based on weighted harmonic mean IPC, the federated core provides 124% higher performance than the baseline scalar core and 3.7% lower performance than the dedicated OOO core for the floating point benchmarks.

Unfortunately, the federated core does not come as close to matching the performance of the dedicated OOO core on the integer benchmarks. Much of the performance gap is most likely due to the combination of two factors: more branch mispredictions, due to the use of a simpler branch predictor; and a larger branch misprediction penalty, due to the longer pipeline and the commit-time branch recovery. The federated core performs particularly poorly on bzip for two of its three reference inputs. These benchmark and input pairs exhibit the most frequent store-to-load forwarding events on the traditional core and so are particularly impacted by the lack of forwarding hardware. The federated core provides 75% higher performance than the baseline scalar core and 12.8% lower performance than the dedicated OOO core for the integer benchmarks.

The average performance of the six different core types is shown in Figure 14, with their average power consumption shown in Figure 15. The 4-way OOO core achieves 68% higher IPC than the federated OOO core but uses about three times as much power, while the dedicated 2-way OOO core achieves 8.4% higher performance than the federated OOO core but dissipates 30% more power. The lightweight OOO core achieves 8.8% better performance than the federated OOO core with only a fraction of a percent higher power consumption. The lightweight core also performs 1.7% worse than the dedicated core on the integer benchmarks, but performs 2.7% better on the floating point benchmarks, mostly due to the MAT allowing loads to execute early. The dedicated in-order core and the federated in-order core have substantially lower performance than the federated OOO core, which is not fully offset by their lower power consumption. This can be partially attributed to the

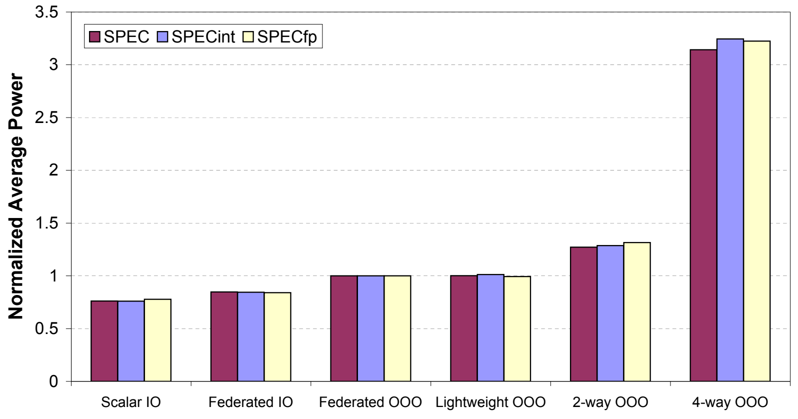


Fig. 15. Harmonic mean power dissipation, normalized to federated OOO.

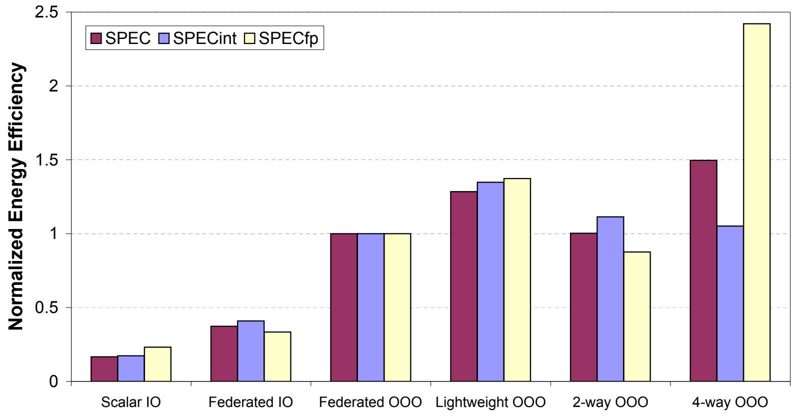


Fig. 16. Energy efficiency ($\frac{\text{BIPS}^3}{\text{Watt}}$), computed using harmonic mean performance and power values and normalized to federated OOO.

fact that all cores—except for the 4-way OOO core, which has larger caches—have similar amounts of leakage in their caches and thus the savings in active power are offset to some degree by the static leakage power.

Figure 16 shows the average energy efficiency in $\frac{\text{BIPS}^3}{\text{Watt}}$ of the different cores.[15] The high-performance 4-way OOO core has a large advantage over the smaller cores in SPECfp, because it is able to use its higher power to achieve substantially better performance. On the other hand, it provides much lower energy efficiency in SPECint, because its higher power comes with a much smaller performance gain. The dedicated 2-way OOO core has better efficiency than the federated OOO core in SPECint, but lower efficiency in SPECfp. The lightweight OOO core has higher

[15] $\frac{\text{BIPS}^3}{\text{Watt}}$ is like ED^2 in that both are voltage-independent metrics to capture the energy cost required for a particular performance level. We prefer the BIPS-based metric because (unlike ED^2) larger values imply better results.

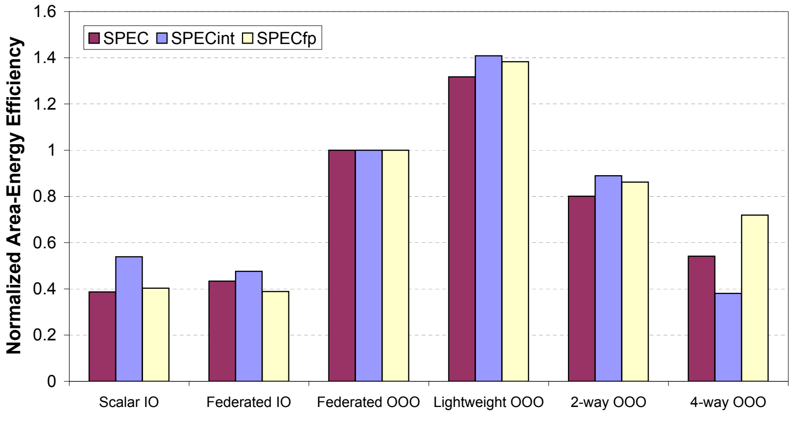


Fig. 17. Area-energy efficiency ($\frac{\text{BIPS}^3}{\text{Watt}\cdot\text{mm}^2}$), computed using harmonic mean performance and power values and normalized to federated OOO.

energy efficiency than the federated core, thanks to its higher performance and essentially equivalent power dissipation; it also provides higher efficiency than the dedicated 2-way OOO core. The two in-order cores have the lowest energy efficiency, even though they have the lowest absolute power consumption. Once again, this is mostly due to leakage power, which penalizes cores with longer execution times.

To measure both the energy- and area-efficiency of the different cores, Figure 17 shows the $\frac{\text{BIPS}^3}{\text{Watt}\cdot\text{mm}^2}$ of the different configurations.[16] The purpose of this metric is to account for the area cost of attaining a certain $\frac{\text{BIPS}^3}{\text{Watt}}$ value. In fact, this metric does not even show Federation's true benefits, since most of the area of the federated core is reused from the underlying scalar cores, whereas the area of the dedicated cores must be cannibalized from the existing cores. Nevertheless, in terms of $\frac{\text{BIPS}^3}{\text{Watt}\cdot\text{mm}^2}$, the lightweight OOO core outperforms the federated OOO core by 32%, while the federated OOO core outperforms the dedicated, traditional 2-way OOO core by 25% and the 4-way core by 85%. In terms of just area-performance efficiency (data not shown), the federated core provides approximately equivalent $\frac{\text{BIPS}^3}{\text{mm}^2}$ as the dedicated 2-way OOO core, while the lightweight and 4-way cores provide 32% and 70% better $\frac{\text{BIPS}^3}{\text{mm}^2}$ than the federated core.

9. FEDERATING 2-WAY CORES

In the previous section we explored federating two multi-threaded scalar cores into an OOO core, based on the assumption that scalar cores were the most efficient use of area for throughput. There have been several recent designs [Johnson and Nawathe 2007] that employ 2-way in-order cores, even when the power budget is very limited. Reasons for choosing 2-way cores instead of scalar cores might include an inability to include a single high-performance core along with the multiple throughput cores, or under-utilization of expensive structures like the data caches

[16]The results for the "Scalar IO" core are based on the area of the single-threaded scalar core shown in in Table VIII, not the multi-threaded core, since only a single thread is executing.

and floating-point units. Such systems need the higher single-thread performance a 2-way core can offer.

For designs that use 2-way in-order cores as their baseline, we explored federating two of these cores into a 4-way OOO core. While all the new structures we introduced for Federation can be scaled to support a 4-way core, we add improvements to some structures to enable both higher performance and energy-efficiency when scaled. The estimated area of the federated 4-way core is given in Table VIII.

9.1 Changes to Federation Structure

Many high performance OOO cores support predicting multiple branches per cycle. While NLS can implicitly jump over non-taken branches, we do not extend either the direction or NLS predictor to produce multiple predictions per cycle.

Commit time branch recovery was already the biggest single performance cost in the 2-way federated core, and would have imposed a 7.3% performance penalty on the 4-way federated core (data not shown). Changing the processor to allow OOO branch recovery requires a small number of rename map checkpoints, as well as logic in the rename stage that steers updates of the rename map to the appropriate branch checkpoint. We found that four branch checkpoints delivered performance almost equivalent to having no limit to the number of branches in the AL.

Simply scaling the subscription-based issue queue to support 4-way issue would require doubling both the number of read and write ports as well as extending the arbitration logic to support issuing four instructions to the different ALUs. To reduce the number of ports required as well as the complexity of the arbitration logic, we use ideas from Tseng and Asanovic [2006] to partition the issue queue among the issue ports in a fixed manner. For a federated core of 2-way baseline cores, the instruction queue is partitioned into four equal partitions. Each partition can only receive and issue a single instruction per cycle, but receives wakeup signals from all partitions. Instructions are assigned to issue queue partitions at rename time primarily based on which ALU type is assigned to which issue port, and secondarily on a load-balancing heuristic. As with load-balancing between cores and selecting among ready instructions, we choose the simplest mechanism possible of distributing instructions round-robin to partitions with empty slots.

Unlike issue queues in clustered architectures, which are distributed among the different clusters, assigning an instruction to a particular partition of the instruction queue does not mean a fixed assignment to a fixed ALU on a fixed core. For the case of a federated 4-way core, the partitioned instruction queue steers instructions to the two cores based on how many instructions are being issued in any given cycle. This is accomplished by taking the ready signals from the four partitions and feeding them into a four-entry priority encoder. The first two partitions with ready instructions get to execute their instructions on core zero, while the next two partitions execute their instructions on core one. For most benchmarks this steering policy means that the great majority of instructions are executed on core zero and do not incur any extra latency when sending or receiving values from the load/store unit. Because the ready information for instructions in the issue queue has to be available before select can occur, the inter-partition priority encoder can operate in parallel to instruction select and not impact the critical path.

An issue that parallels the problems of the issue queue is the increasing number

Parameter	2-way IO	4-way OOO
Active List	none	128
Issue Queue	none	32
Load-Store Queue	none	64
Data Cache	32KB	64KB
Instruction Cache	32KB	64KB
Unified L2 Cache	2MB	2MB
Branch Target Buffer	512	4K
Direction Predictor	2K bimodal	16K tournament
Memory	150 Cycles, 64-Bit	
Branch Misprediction Penalty	16 Cycles minimum	

Table IX. Simulator parameters for the 2-way in-order and 4-way out-of-order cores.

of ports on the register file. While the number of read ports required by Federation is matched by the underlying cores, the number of write ports is not. To avoid having to increase the number of write ports, we partition the unified register file between the different functional units, as proposed by Kucuk et al. [2003]. Using the banked register file of the underlying core, we assign one bank per issue port, reducing the number of write ports required to just one per bank.

Because the 4-way core can still only issue one load and one store per cycle, the MAT retains the same number of ports as the 2-way federated core. To support a larger number of memory instructions in flight without too many false positive aliasing events, we increase the number of entries in the MAT from 32 to 128.

9.2 Simulation Setup

The simulation infrastructure described in Section 7 was also used for this set of experiments. The resources of the dedicated 2-way in-order and 4-way OOO cores are shown in Table IX. To reflect the greater emphasis on single-thread performance that a design using 2-way in-order cores might have, we substantially increased the pipeline depth of all of the core types to more accurately represent designs that aim at achieving higher frequencies.

We compare the 4-way federated core against five other cores: the scalar and 2-way in-order cores used as the baseline for the 2-way and 4-way federated cores, respectively; the lightweight 2-way OOO core; and the dedicated 4-way OOO core. The resources of the lightweight core have been scaled to match those of the the dedicated 4-way OOO core.

9.3 Results

Figure 18 shows the relative performance and energy efficiency of the six core types. The 4-way federated core achieves performance only 11% worse than the dedicated 4-way OOO core. Comparing the $\frac{\text{BIPS}^3}{\text{Watt}}$ of the different cores shows that the 4-way federated core provides 7.3% better energy efficiency than the dedicated 4-way core. This result shows that even large OOO cores can benefit significantly from more power efficient structures, as long as they do not impact performance too heavily.

The changes to the Federation structures outlined in Section 9.1 impact performance as follows: the enlarged MAT boosts performance by 1.0% due to fewer false positive memory aliasing events; the partitioned instruction queue has only a negligible impact on average performance; and the introduction of branch checkpoints

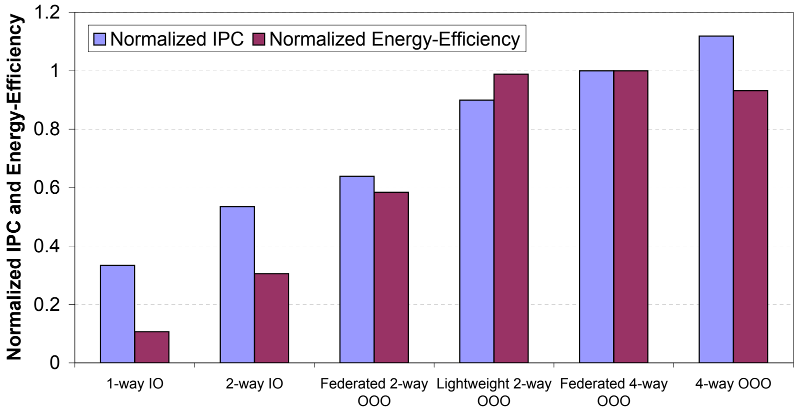


Fig. 18. Harmonic mean IPC and $\frac{\text{BIPS}^3}{\text{Watt}}$, normalized to federated 4-way OOO.

improves performance by 7.8% and is the single largest contributor to the improved performance of the 4-way federated core.

10. CONCLUSIONS AND FUTURE WORK

Manycore chips of dozens or more simple but multi-threaded cores will need the ability to cope with limited thread count by boosting the per-thread performance. This paper shows how 2-way OOO capability can be built from very simple, in-order cores, with performance 99% better than the in-order core, 21% lower average power than a dedicated 2-way OOO core, and competitive energy efficiency compared to a 2-way OOO core. Using a subscription-based issue queue and eliminating the Load-Store Queue in favor of the Memory Alias Table, we have shown that no major CAM-based structures are needed to make an OOO pipeline work. In fact, these same insights can be used to design a new, more efficient, dedicated OOO core, as the lightweight OOO results show. We have also shown that the techniques of Federation can be applied to higher performance 2-way in-order cores to achieve performance close to that of a dedicated high-performance 4-way OOO core.

As mentioned in Section 2, the structure of Federation was chosen with the lessons of clustering in mind. As such, we designed Federation without further plans for horizontally aggregating more than two cores into a single very wide core. For higher single-thread performance, the combination of Federation with techniques that can effectively shorten the critical path — such as runahead execution [Mutlu et al. 2003], sophisticated pre-fetchers [Ganusov and Burtscher 2006], or dynamic optimization [Almog et al. 2004] — seems to be the most fruitful path to pursue. An advantage of many such techniques is their toleration of infrequent or long latency communication with the main core, which makes it much easier to implement them using multiple cores of a manycore processor.

The most important advantage of Federation is that it can be added to a manycore architecture without sacrificing the ability to use the constituent in-order cores as multi-threaded, throughput-oriented cores. Federation requires several new structures, but with very low area overhead—about 1.3KB of new SRAM tables and less than 0.7KB of new register-type structures in the pipeline per *pair* of cores—only 5.7% area overhead per pair. Put another way, this means that for a set of 32 scalar cores, the area of Federation for each pair only adds an aggregate area equivalent to 0.91 cores or 0.57 MB of L2 cache. For 2-way in-order cores with branch prediction the relative area overhead is even less. As a result, Federation actually provides greater energy efficiency per unit area: 25% better $\frac{\text{BIPS}^3}{\text{Watt}\cdot\text{mm}^2}$ than a dedicated 2-way OOO core, and 85% better than a 4-way OOO core!

The option of adding Federation therefore removes the need to choose between high throughput with many small cores or high single-thread performance with aggressive OOO cores and the associated problems of selecting a fixed partitioning among some combination of these. This is particularly helpful in the presence of limited parallelism as it allows a manycore chip to trade off throughput for latency on a very fine-grained level at runtime. Federation thus allows manycore chips to give higher performance across a wider spectrum of workloads with different amounts of TLP, as well as deal with workloads that have different amounts of parallelism during different phases of execution.

11. ACKNOWLEDGMENTS

This work was supported by NSF grant nos. CCR-0306404, CNS-0509245, IIS-0612049, SRC grant no. 1607, a GRC AMD/Mahboob Khan Ph.D. fellowship, and a Ballard graduate fellowship. We would like to thank Michael Frank, Doug Burger, Mircea Stan, Jeremy Sheaffer, Liang Wang, Runjie Zhang, and the anonymous reviewers for their helpful comments.

REFERENCES

Almog, Y., Rosner, R., Schwartz, N., and Schmorak, A. 2004. Specialized dynamic optimizations for high-performance energy-efficient microarchitecture. In *Proceedings of the 2nd International Symposium on Code Generation and Optimization.*

AMD. 2007. ATI Radeon HD 2900 technology: GPU specifications.

Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., and Yelick, K. A. 2006. The landscape of parallel computing research: A view from Berkeley. Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

Brekelbaum, E., Rupley, J., I., Wilkerson, C., and Black, B. 2002. Hierarchical scheduling windows. In *Proceedings of the 35th International Symposium on Microarchitecture.*

Brooks, D., Tiwari, V., and Martonosi, M. 2000. Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture.*

Burger, D., Austin, T. M., and Bennett, S. 1996. Evaluating future microprocessors: The SimpleScalar tool set. Tech. Rep. CS-TR-1996-1308, University of Wisconsin-Madison.

Burger, D., Keckler, S. W., McKinley, K. S., Dahlin, M., John, L. K., Lin, C., Moore, C. R., Burrill, J., McDonald, R. G., Yoder, W., and the TRIPS Team. 2004. Scaling to the end of silicon with EDGE architectures. *IEEE Computer 37,* 7.

Butts, J. A. and Sohi, G. S. 2002. Characterizing and predicting value degree of use. In *Proceedings of the 35th International Symposium on Microarchitecture.*

Calder, B. and Grunwald, D. 1995. Next cache line and set prediction. In *Proceedings of the 22nd International Symposium on Computer Architecture.*

Carmean, D. 2007. Future CPU architectures: The shift from traditional models. Intel Higher Education Lecture Series.

Chou, Y., Fahs, B., and Abraham, S. 2004. Microarchitecture optimizations for exploiting memory-level parallelism. In *Proceedings of the 31st International Symposium on Computer Architecture.*

Davis, J. D., Laudon, J., and Olukotun, K. 2005. Maximizing CMP throughput with mediocre cores. In *Proceedings of the 15th Conference on Parallel Architectures and Compilation Techniques.*

Dolbeau, R. and Seznec, A. 2004. CASH: Revisiting hardware sharing in single-chip parallel processors. *Journal of Instruction-Level Parallelism 6.*

Ganusov, I. and Burtscher, M. 2006. Efficient emulation of hardware prefetchers via event-driven helper threading. In *Proceedings of the 15th Conference on Parallel Architectures and Compilation Techniques.*

Garg, A., Castro, F., Huang, M., Chaver, D., Pinuel, L., and Prieto, M. 2006. Substituting associative load queue with simple hash tables in out-of-order microprocessors. In *Proceedings of the 12th International Symposium on Low Power Electronics and Design.*

Glew, A. 1998. MLP yes! ILP no! In *ASPLOS Wild and Crazy Ideas.*

Grochowski, E., Ronen, R., Shen, J., and Wang, H. 2004. Best of both latency and throughput. In *Proceedings of the 22nd International Conference on Computer Design.*

Hester, P. 2006. 2006 AMD Analyst Day Presentation.

Hofstee, H. P. 2005. Power efficient processor architecture and the Cell processor. In *Proceedings of the 11th International Conference on High Performance Computer Architecture.*

Huang, M., Renau, J., and Torrellas, J. 2002. Energy-efficient hybrid wakeup logic. In *Proceedings of the 8th International Symposium on Low Power Electronics and Design.*

İpek, E., Kirman, M., Kirman, N., and Martínez, J. 2007. Core fusion: Accommodating software diversity in chip multiprocessors. In *Proceedings of the 34th International Symposium on Computer Architecture.*

Johnson, T. and Nawathe, U. 2007. An 8-core, 64-thread, 64-bit power efficient Sparc SOC. In *Proceedings of the 54th International Solid-State Circuits Conference.*

Kessler, R., McLellan, E., and Webb, D. 1998. The Alpha 21264 microprocessor architecture. In *Proceedings of the 16th International Conference on Computer Design.*

Kim, C., Sethumadhavan, S., Govindan, M. S., Ranganathan, N., Gulati, D., Burger, D., and Keckler, S. W. 2007. Composable lightweight processors. In *Proceedings of the 40th International Symposium on Microarchitecture.*

Kongetira, P., Aingaran, K., and Olukotun, K. 2005. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro 25,* 2.

Kucuk, G., Ergin, O., Ponomarev, D., and Ghose, K. 2003. Distributed reorder buffer schemes for low power. In *Proceedings of the 21st International Conference on Computer Design.*

Kumar, R., Jouppi, N., and Tullsen, D. 2004. Conjoined-core chip multiprocessing. In *Proceedings of the 37th International Symposium on Microarchitecture.*

Kumar, R., Tullsen, D. M., Ranganathan, P., Jouppi, N. P., and Farkas, K. I. 2004. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of the 31st International Symposium on Computer Architecture.*

Mesa-Martinez, F. J., Nayfach-Battilan, J., and Renau, J. 2007. Power model validation through thermal measurements. In *Proceedings of the 34th International Symposium on Computer Architecture.*

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 Conference on High Performance Computer Architecture.*

NVIDIA. 2009. NVIDIA CUDA programming guide. Version 2.2.1.

Onder, S. and Gupta, R. 1999. Dynamic memory disambiguation in the presence of out-of-order store issuing. In *Proceedings of the 32nd International Symposium on Microarchitecture.*

Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krger, J., Lefohn, A. E., and Purcell, T. J. 2007. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum 26,* 1.

RAASCH, S. E., BINKERT, N. L., AND REINHARDT, S. K. 2002. A scalable instruction queue design using dependence chains. In *Proceedings of the 29th International Symposium on Computer Architecture*.

RAMÍREZ, M. A., CRISTAL, A., VEIDENBAUM, A. V., VILLA, L., AND VALERO, M. 2004. Direct instruction wakeup for out-of-order processors. In *Proceedings of the International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems*.

ROTH, A. 2005. Store vulnerability window (SVW): Re-execution filtering for enhanced load optimization. In *Proceedings of the 32nd International Symposium on Computer Architecture*.

SALVERDA, P. AND ZILLES, C. 2008. Fundamental performance challenges in horizontal fusion of in-order cores. In *Proceedings of the 14th International Conference on High Performance Computer Architecture*.

SANKARALINGAM, K., NAGARAJAN, R., MCDONALD, R., DESIKAN, R., DROLIA, S., GOVINDAN, M. S., GRATZ, P., GULATI, D., HANSON, H., KIM, C., LIU, H., RANGANATHAN, N., SETHUMADHAVAN, S., SHARIF, S., SHIVAKUMAR, P., KECKLER, S. W., AND BURGER, D. 2006. Distributed Microarchitectural Protocols in the TRIPS Prototype Processor. In *Proceedings of the 39th International Symposium on Microarchitecture*. 480–491.

SASSONE, P. G., II, J. R., BREKELBAUM, E., LOH, G. H., AND BLACK, B. 2007. Matrix scheduler reloaded. In *Proceedings of the 34th International Symposium on Computer Architecture*.

SATO, T., NAKAMURA, Y., AND ARITA, I. 2001. Revisiting direct tag search algorithm on superscalar processors. In *Proceedings of the Workshop on Complexity-Effective Design*.

SETHUMADHAVAN, S., DESIKAN, R., BURGER, D., MOORE, C. R., AND KECKLER, S. W. 2003. Scalable hardware memory disambiguation for high ILP processors. In *Proceedings of the 36th International Symposium on Microarchitecture*.

SEZNEC, A., FELIX, S., KRISHNAN, V., AND SAZEIDES, Y. 2002. Design tradeoffs for the Alpha EV8 conditional branch predictor. In *Proceedings of the 29th International Symposium on Computer Architecture*.

SHA, T., MARTIN, M. M. K., AND ROTH, A. 2005. Scalable store-load forwarding via store queue index prediction. In *Proceedings of the 38th International Symposium on Microarchitecture*.

SHA, T., MARTIN, M. M. K., AND ROTH, A. 2006. NoSQ: Store-load communication without a store queue. In *Proceedings of the 39th International Symposium on Microarchitecture*.

SHERWOOD, T., PERELMAN, E., HAMERLY, G., AND CALDER, B. 2002. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*.

SKADRON, K., STAN, M. R., HUANG, W., VELUSAMY, S., SANKARANARAYANAN, K., AND TARJAN, D. 2003. Temperature-aware microarchitecture. In *Proceedings of the 30th International Symposium on Computer Architecture*.

SMITH, A., BURRILL, J., GIBSON, J., MAHER, B., NETHERCOTE, N., YODER, B., BURGER, D., AND MCKINLEY, K. 2006. Compiling for EDGE architectures. In *Proceedings of the 4th International Symposium on Code Generation and Optimization*.

SUBRAMANIAM, S. AND LOH, G. H. 2006. Fire-and-forget: Load/store scheduling with no store queue at all. In *Proceedings of the 39th International Symposium on Microarchitecture*.

TARJAN, D., BOYER, M., AND SKADRON, K. 2008. Federation: Repurposing scalar cores for out-of-order instruction issue. In *Proceedings of the 45th Design Automation Conference*.

TREMBLAY, M. AND O'CONNOR, J. M. 1996. UltraSparc I: A four-issue processor supporting multimedia. *IEEE Micro 16,* 2.

TSENG, J. H. AND ASANOVIC, K. 2006. RingScalar: A complexity-effective out-of-order superscalar microarchitecture. Tech. Rep. MIT-CSAIL-TR-2006-066, MIT CSAIL.

ZHONG, H., LIEBERMAN, S. A., AND MAHLKE, S. A. 2007. Extending multicore architectures to exploit hybrid parallelism in single-thread applications. In *Proceedings of the 13th International Conference on High Performance Computer Architecture*.