Federation: Out-of-Order Execution using Simple In-Order Cores

UNIV. OF VIRGINIA DEPT. OF COMPUTER SCIENCE TECH. REPORT CS-2007-11
AUG. 2007

David Tarjan, Michael Boyer and Kevin Skadron
{dtarjan,mwb7w,skadron}@cs.virginia.edu
Dept. of Computer Science
University of Virginia
Charlottesville, VA 22904

Abstract

Manycore architectures with dozens, hundreds, or thousands of threads are likely to use single-issue, in-order execution cores with simple pipelines but multiple thread contexts per core. This approach is beneficial for throughput but only with thread counts high enough to keep most thread contexts occupied. If these manycore architectures do not want to be limited to niches with embarrassing levels of parallelism, they must cope with the case when thread count is limited: too many threads for dedicated, high-performance cores (which come at high area cost), but too few to exploit the huge number of thread contexts. The only solution is to augment the simple, scalar cores.

This paper describes how to create an out-of-order processor on the fly by *"federating"* each pair of neighboring, scalar cores.This adds a few new structures between each pair but otherwise repurposes the existing cores. It can be accomplished with less than 2KB of extra hardware per pair, nearly doubling the performance of a single, scalar core and approaching that of a traditional, dedicated 2-way out-of-order core. The key insights that make this possible are the use of the large number of registers in multi-threaded scalar cores to support out-of-order execution and the removal of large, associative structures. Federation provides a scalable, energy-efficient, and area-efficient solution for limited thread counts, with the ability to boost performance across a wide range of thread counts, until thread count returns to a level at which the baseline, multithreaded, "throughput mode" can resume.

1 Introduction

Increasing difficulties in improving frequency and instruction-level parallelism have led to the advent of multicore processors. Researchers are now debating whether we can scale to hundreds or thousands of cores. When designing such a *manycore* 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.* [10] and Carmean [8] show that the maximum aggregate throughput for a processor is achieved by using a large number of smaller, less capable cores. In fact, Carmean estimates [8] that a high performance, multi-threaded in-order core takes up only one-fifth the area of a traditional out-of-order (OOO) core while providing more than 20 times the throughput per unit area and more than 20 times the throughput per watt. Clearly, the benefits are large enough and the market for throughput computing is large enough that some vendors have targeted products at this space. Sun's Niagara/T1 chip is the most prominent example [1].

Based on that argument, this paper considers manycore chips of dozens or hundreds of simple, scalar, but multithreaded cores like those in Niagara—cores with only a simple pipeline, no branch prediction, and no hardware whatsoever to support superscalar, let alone out-of-order, execution. The problem with a manycore chip like this is that some workloads will not have enough threads to fill so many thread contexts (even 32 contexts, like Niagara, can be difficult to fill). Such a scenario may arise for a variety of reasons. It is well known that work imbalance leading up to a barrier or in problems with irregular parallelism can be significant. Dynamic load balancing can address this, but only if the problem exhibits a structure compatible with shifting load. It is also well known that in many workloads, the parallelism simply does not scale to such large numbers of threads, as communication or other overheads swamp the actual computation. And in workloads with many independent or quasi-independent tasks (including multiprogramming, map-reduce, e-commerce workloads, etc.) the number of tasks varies. When utilization is too low to fill up the manycore chip's thread contexts, it may be desirable to speed up the remaining threads.

This paper describes how to improve a workload's performance when the active threadcount is too low to take advantage of the throughput benefits provided by simple, multi-threaded cores. This is accomplished by *federating* two of these ultra-simple cores into an out-of-order core. Surprisingly, it turns out that this can be accomplished with only a few small, auxiliary structures (at an area overhead of just 3.7%, according to our estimates). The performance will be a bit less than a traditional, 2-way issue OOO core, but still a dramatic improvment over standalone, scalar cores. As a simple rule of thumb, federation can be performed when a core has only one thread and its neighbor is idle. Specific policies for engaging federation, including more sophisticated policies to make greater use of federation, are left for future work. The focus of this paper is to describe the microarchitecture and show, for a single thread running on one pair of cores, how effective it is.

1.1 Why Federation?

It may be objected that a better solution would be to have a small number of dedicated OOO cores to handle limited thread count. Even with SMT, an approach based on dedicated OOO cores cannot solve the problem for more than a few threads—it is not scalable. Certainly an OOO core will give great performance on *one* thread, and provisioning a single OOO core makes a lot of sense to deal with the "Amdahl's Law problem" posed by serial portions of execution or a single, interactive thread.[1]. Beyond that, we start to run into problems. A 2-way SMT OOO core will slightly outperform *one* federated pair for two threads (at much higher power). But dedicated OOO cores, SMT or not, do not scale beyond a few threads. To support dozens of threads (in a manycore chip of dozens or hundreds of cores) would require many OO cores. Even if it were possible to decide on the magic number of OOO cores, the area cost would come at the expense of too many simple cores. *For a manycore chip, when threadcount is more than a few, but too limited to benefit from a large number of cores, the only viable way to boost performance is to harness the many simple cores that are already present.* We will show that federation boosts performance with minimal area overhead and provides an area-efficient solution as well. In fact, federated two-way issue turns out to often give nearly competitive performance and *superior* energy-efficiency than full-fledged, two-way, OOO execution and the best energy-efficiency per unit area of all the options we studied!

Clearly, federation can also be helpful for single-thread portions of execution if the designer chooses not to include a dedicated OOO core. This might occur if single-thread workloads or serial phases are not considered sufficiently important, or if design time or intellectual property issues preclude the use of a true OOO core.

1.2 Contributions

In this work, we describe how to take two minimalist, scalar, in-order cores that have only no branch prediction hardware and combine them to achieve two-wide, OOO issue. A key observation is that a throughput-oriented core generally supports multiple thread-contexts and therefore requires many registers, while a performance-oriented core generally supports only one or two threads in the interest of maximizing single-thread latency. Building a performance-oriented core from a throughput-oriented core leaves a large pool of registers free to store the state necessary for supporting OOO execution. This work focuses specifically on how to synthesize *dual-issue* OOO execution, examining what functionality must be added, whether it can be implemented using existing hardware, and the cost of the additional hardware and interconnect required. We choose dual-issue because it represents a "sweet spot" when using such simple cores as building blocks and obtains the best balance of performance, energy-efficiency and area efficiency of all the options we explored. Synthesizing a wider OOO core out of scalar cores, especially scalar cores with no branch prediction, has the traditional problem of diminishing returns, because the hardware resources required to exploit more ILP grow super-linearly and inter-cluster control complexity and delays become more serious. We focus on the simplest cores as building blocks because they make the most challenging case for what we propose and because such simple cores are indeed a realistic case, as Sun's Niagara/T1 [1] and the architecture outlined by Carmean [8] show.

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 3.7% area increase over a pair of scalar in-order cores, using simple SRAM lookup structures and no major additional content addressable memory (CAM) structures. By comparison, a traditional 2-way OOO core costs 2.65 scalar cores in die area!

[1] We see this approach embodied in the Sony Cell [17] and AMD Fusion [16]

- 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, the two-way federated organization often approaches the performance of a traditional OOO organization of the same width, is competitive in energy efficiency with a traditional OOO core of the same width, and has better area-efficiency than all other options we studied.
- We show that a subscription-based issue queue design with no tag-match or broadcast logic can achieve competitive performance for a small issue window size.
- We introduce the Memory Alias Table (MAT), which provides approximately the same functionality and performance as the Store Vulnerability Window [28] while using an order of magnitude fewer bits per entry.
- We show that completely eliminating the forwarding logic between loads and stores is a viable option for a small OOO core.
- We show that federating a pair of cores is competitive with a "lightweight", dedicated OOO core using the same principles.[2]

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

The rest of this paper is organized as follows. Section 2 discusses relates federation to previous work. Section 3 sketches 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 OOOE and the new structures required, while Section 5 explains how the constituent cores' caches are federated. Section 6 presents the details of our memory alias table design. Section 7 describes the details of our simulation environment and benchmarks. Section 8 presents performance, power, and area-efficiency results. Section 9 concludes the paper.

2 Related Work

Previous work on combining several smaller cores into a single larger and more capable core was performed by İpek *et al.* [19]. They show how to combine several small OOO processors into a single wider OOO processor, whose performance can rival or even exceed the performance of existing high-performance designs. Because their underlying processors already contain all of the structures needed for OOO execution, they focus on showing how to make these structures work in a distributed fashion. The goal of this "core fusion" technique is to maximize single-thread performance, and might be a good alternative to a dedicated, aggressive OOO core (assuming, of course, that the requisite OOO structures already exist on the chip). Federation would not scale like core fusion, but that is not the intent. Federation solves an entirely different problem, where there are no pre-existing structures that could support the method in [19], and the goal is to maximize throughput when thread count is too modest to use the scalar cores' multi-threading capabilities. Adding dedicated OOO structures amenable to core fusion would come at a high area cost. Federation—with an area overhead of just 3.7%–is able to achieve 2-way out-of-order execution and nearly double throughput in limited-thread-count situations.

The Voltron architecture from Zhong et al. [37] allows multiple in-order VLIW cores of a chip multiprocessor (CMP) to combine into a larger VLIW core. They require a special compiler to transform programs into a form which can be exploited by this larger core. Our work does not assume an advanced compiler and is applicable to RISC, CISC and VLIW ISAs.

Work by Kumar *et al.* [23] 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.* [15] 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.

Sharing resources between multiple cores in a CMP has been evaluated by Dolbeau [11] and Kumar [22].

[2]Note that the lightweight OOO core is not a viable solution for our problem, 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 displaced.

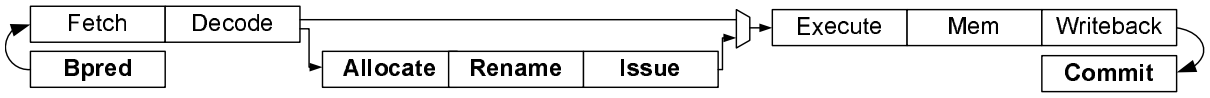

Figure 1: The pipeline of a federated core, with the new pipeline stages in bold.

Numerous groups have evaluated various combinations of clustered OOO processors and multithreading. İpek *et al.* [19] provide a comprehensive overview of this body of work.

Another approach to improve the single-thread performance is to use runahead execution [9, 25]. It should be noted that runahead is orthogonal and even complementary to federating two simple cores. Runahead reduces time spent waiting on cache misses, which may actually help the more powerful federated core relative to the underlying scalar core; and federating two cores helps the runahead thread run faster and thus further ahead of the main thread.

The Store Vulnerability Window (SVW) was introduced by Roth [28] as a verification mechanism for load/store aliasing and ordering which could be used in conjunction with several load speculation techniques. More recent work [33, 35, 32] has tried to largely or completely eliminate the load/store queue (LSQ) by using the SVW as the checking mechanism for speculative forwarding. 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.

3 Background

Future microprocessor designs will likely incorporate many simple in-order cores rather than a small number of complex OOO cores [3]. Current examples of this trend include the Sun Niagara I and II [1, 20], 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 [27]. The most recent GPU designs from NVIDIA [26] and AMD [2] incorporate 128 and 320 processing elements, respectively. As discussed above, 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 multicore, in-order architecture. Based on the current trends cited above, the baseline in-order microarchitecture which 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 to achieve high throughput by exploiting thread-level parallelism (TLP) and memory-level parallelism (MLP) [14]. 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 (4-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 and use a branch address calculator (BAC) in the decode stage to minimize fetch bubbles and to conserve ALU bandwidth.

Since one of the main goals of federation is to add OOOE capability to the existing in-order cores with as little area overhead as possible, each federated OOO core is relatively simple compared to current dedicated OOO implementations. Specifically, each federated core is single-threaded[3] and two-way issue with a 32-entry instruction window. The federated cores implement the pipeline shown in Figure 1, with the additional pipeline stages not present in the baseline in-order cores shown in bold. A possible floorplan for the federated core is shown in Figure 2. To simplify the discussion, in this paper we focus on a single pair of in-order cores which can federate to form a single OOO core.

[3]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. This clearly has implications for thread scheduling, which will be explored in future work.

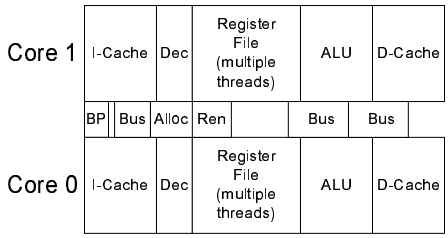

Figure 2: A simplified floorplan showing the arrangement of two in-order cores with the new structures necessary for federation in the area between the cores.

Structure	Entries	Entry Size (Bits)	Total Size (Bits)	Type	Reuses RF
Branch Predictor (NLS)	512	12	6,144	SRAM	No
Branch Predictor (Bimodal)	2,048	2	4,096	SRAM	No
Return Address Stack	8	32	256	SRAM	No
Speculative Rename Table	64	10	640	Register	No
Retirement Rename Table	64	6	384	Register	No
Free Lists	64	6	384	Register	No
Issue Queue (Wakeup)	16	11	176	Register	No
Issue Queue (Data)	16	56	896	Register	Yes
Unified Register File	64	64	4,096	Register	Yes
MAT	32	1.625	<64	Register	No
Bpred Recovery State	4	64	256	Register	No
Worst Case Total	10,496 SRAM Bits and 6,844 Register Bits				
Assumed Base Case Total	10,496 SRAM Bits and 1,852 Register Bits				

Table 1: Area estimates for the new structures added to the baseline in-order processor. Listed are the number of entries, the size of each entry, the total size, and the type of memory used to implement each structure. The last column indicates whether we assume the structure can be built using only reused register file entries from the baseline multi-threaded core. The worst case total is calculated under the assumption that none of the structure can reuse the register file.

In practice, the techniques we describe are intended to be applied to a multicore processor with a significantly larger number of cores, with each adjacent pair of cores able to federate into a single OOO core.

4 Out-of-Order Pipeline

Our principal design goal, which drives all of the design decisions mentioned in the following sections, is to add as little area overhead as possible to the existing design, and especially to avoid adding any significant CAMs or structures with a large number of read and write ports. Table 1 lists the sizes of the new structures required to support OOOE, 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. The structures which reuse the register file in our design are those which are close to the register read and write back stages in the pipeline, require few read and write ports, and are read and written to at sizes close to those which the register file already supports.

The new logic and wiring required to support federation are listed in Tables 2 and 3, respectively. The following subsections provide a detailed explanation of the operation of each pipeline stage in the federated core, along with justification for the design tradeoffs that were made.

New Logic	Explanation
Instruction Scheduler	Section 4.1
Branch Recovery	Section 4.5
Merged I-Cache Controller	Section 5
Merged D-Cache Controller	Section 5
Rename Override and Stall Logic	Section 4.5
Branch Prediction FSM	Section 4.1
BP Update Logic	Section 4.10

Table 2: New logic required for federation. See the referenced sections for further details.

New Wiring	Width
Cross Core Value Copying	2 * (64 + 6) bits
Mem Unit to 2nd D-Cache	2 * 64 bits
Cross I-Cache to Decode	32 bits
Decode to Allocate	approx. 64 bits

Table 3: The size of 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 prediction (NLS) [7, 21, 36] 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 predicted as the next cache line to be fetched. 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 visible from this simplification. A new finite state machine (FSM) keeps track of which request to send to the instruction cache, deciding among misprediction recovery requests from the commit stage, the return address stack, and the NLS.

4.2 Fetch

The fetch stage starts by receiving a predicted cache line from NLS, 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 (refer to section 5.1 for implementation details of the shared I-cache). Even though the baseline cores are only one-wide, we assume that they fetch an entire cache line per request. Since the cache line size does not change when federated, the bandwidth provided by a single fetch engine is sufficient to support the federated core. For simplicity, the decision about which fetch unit to use in federated mode is made statically at design time. The active fetch unit buffers the current cache line, rotates it if needed and routes one instruction to the decode units in each of the two cores. Since each core can only decode a single instruction, the second instruction (if valid) is sent to the second 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 second core, buffering the first instruction in a pipeline register at the same time.

4.3 Decode

Once an instruction has 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 to allow the signals from both decode units to propagate to the allocate unit. The performance implications of this routing overhead are discussed in Section 8. The BAC of the baseline core 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 OOOE. All instructions check for space in both the Issue Queue (IQ) and the Active List (AL). In traditional OOO architectures, load and store instructions would also need to check for a free LSQ entry, but our implementation uses a memory alias table, which

is free from such constraints (see Section 6). 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 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

Since none of the functionality of the rename stage is present in an in-order processor, it must be implemented from scratch. The federated core uses a unified register file with speculative and retirement register alias tables (RAT). 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 issue queue 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 size of the rename table, it makes sense to implement it as a separate structure.

There are two separate OOO register files, one each for the integer and floating point registers. Each register file consists of the 32 architected registers and a number of rename registers, implemented using the register file of the underlying multi-threaded core, and each register is 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 reads from or 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 which check the poison bits of their input operands have to 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 associated with each RAT entry is set to indicate that the most up to date version of the register is in the non-speculative RAT. As soon as an instruction in the rename stage writes to a particular register, this bit is reset to indicate that the speculative version is the most up to date.

4.6 Issue

The area and power constraints of our design prevent the implementation of a traditional CAM-based issue queue. Instead we use an issue queue without any 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 producers consumer fields, similar to the scheme evaluated by Huang *et al.* [18]. They 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. Thus, each issue queue entry in our design only has a small number of consumer fields; the impact of varying this number is evaluated in Section 8. In this scheme, an instruction can stall if its producer is oversubscribed. This necessitates the addition of an extra bit to each producer entry in the issue queue

which is set if the producer is oversubscribed. If this bit is set when the instruction executes, a signal is sent to the rename stage to unstall the dependent instruction(s).

Each entry in the issue queue contains two ready bits, which are set when the left and right operands become available, respectively. If both input operands are ready, the ready signal for that entry is sent to the scheduler. Note that all loads can issue speculatively, without waiting on unresolved stores; see section 6 for an explanation. Each entry in the issue queue also requires two fields for the active list IDs of the producers of its input operands, a field to store its opcode, and a field to store its immediate/displacement value. These extra fields are not required for the critical wakeup and select loop and can thus be stored in a table physically separate from the ready bits and the consumer IDs.

Fully exploiting the parallelism exposed by the instruction queues requires the use of an effective hardware scheduler. Traditionally, the best metric for choosing between multiple simultaneously ready instructions has been to give preference to the oldest instruction [12], and our baseline model uses instruction age for scheduling decisions. However, recent work by Sassone *et al.* [29] has shown that much simpler scheduling mechanisms can be used for small issue queues. They show that using a simple, static priority encoder which only uses the queue position as input (what they called pseudo-random scheduling) only incurs a 1% performance loss compared to age-based scheduling for a 16-entry issue queue. Since the goal of our design is minimal extra complexity, we choose to give up the slight performance advantage of an age-based scheduler in favor of the smaller and simpler implementation of a pseudo-random scheduler. Note that forward progress is assured, because even if the oldest instruction is not picked several times, it will become the only ready instruction when the AL and the IQ fill up. The exact performance impact of pseudo-random scheduling on our design is 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 mirror 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 possibility, scheduling all operations on core 0 by default and only using core 1 if core 0 has already been allocated that cycle. Also, only the load and store ports of core 0 are used. As shown in Section 8, the impact of clustering is small even with a naive scheduler.

4.7 Execute

Each instruction executes normally on the ALU to which it was assigned 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 dependence-based issue queue 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

Rather than a traditional load-store queue, our design uses a memory alias table (MAT). A detailed explanation and evaluation of the MAT is provided in Section 6. 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.

4.9 Write Back

Similar to the Alpha 21264 [21], all results are written to the register files on both cores, to avoid the complication of having to generate copy instruction 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 one extra cycle of latency and evaluate the performance impact of this delay 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 limiting the number of branch checkpoints. However, since our focus is on simplicity, our base case for the federated

core still uses commit time branch recovery. The performance penalty of commit-time branch resolution and limiting the number of branch checkpoints is shown in Section 8.

During commit, stores check the MAT for memory order violations. If a violation is detected, the offending load and all younger instructions are flushed from the pipeline as soon as that load reaches commit. Since commit is dependent on the single AL, and commit only uses load/store ports on core 0, never core 1, neither core can slip in relation to the other and global commit order is always maintained.

5 Caches

Each in-order core has its own private L1 data and instruction caches. When the two cores are federated, two physically distinct caches must act as one logical cache. There are several possible approaches to achieving this:

- Only use one of the two caches, essentially wasting half of the available memory.
- Double the line size of the merged cache by fusing corresponding cache lines in the two caches.
- Double the number of sets in the merged cache by interleaving the sets of the two caches.
- Double the associativity of the merged cache by merging the corresponding sets of the two caches.

In addition to impacting the performance of the federated processor, the decision about how to combine the caches can also impact the amount of work required to transition between in-order and OOO mode. We focus on the former impact rather than the latter, based on the assumption that the processor will spend much more time in either in-order or OOO mode rather than in a transitional state.

For both data and instruction caches, we choose to merge the caches of the individual cores into a new data (instruction) cache with double the associativity. Using only one of the two caches would impose too much of a negative performance impact, and changing either the line size or the number of sets would require changing the address decoding logic of the cache.

In order to support the merged cache, we add a cache controller that forwards memory requests from the federated core to the two individual caches. If one of the two caches hits, the data is sent to the controller, which then forwards it to the core. If both of the caches miss, the controller sends a request for the missing data to the L2 cache. When the L2 cache returns the data, the controller forwards it to one of the two L1 caches. The controller therefore is responsible for a portion of the replacement policy by making the global decision of which cache receives the new data; each individual cache is responsible for making the local decision of which block to evict to make room for the new data. One of the advantages of this approach is that each individual cache in the baseline architecture does not need to know whether it is being used in federated mode or not.

Misses to the merged L1 cache access the L2 cache as they would in the baseline in-order core. We assume that the whole chip includes many processor tiles, each containing two cores, with enough L2 cache banks and a wide enough connection to main memory to satisfy the bandwidth requirements of the many thread contexts. Each tile can reconfigure independently and gets assigned a single L2 cache bank but shares the memory bus with many other threads.

Note that when federating, the two individual caches can both contain data from the same memory location. Merging the two caches in this case would result in the same data residing in two different locations within the same set, which would violate a fundamental assumption about the cache. In order to prevent this violation, at least one of the two individual caches must be flushed when the in-order cores federate.

5.1 Instruction Caches

The merged instruction cache has double the associativity of the individual caches and uses random replacement. The only tag read out is the tag of the cache line predicted as the target by the Next Line and Set predictor. This means that the energy overhead of having to compare more tags in a higher associativity cache is not an issue in our design. Also, there is no need for extra logic to choose between the two side of each merged set.

5.2 Data Caches

We assume the data cache of each in-order core has a certain number of miss status handling registers (MSHR) per thread context, with the merged cache having twice the number of each individual data cache. Each baseline core also

has a post-commit store buffer, with the number of entries scaling with the number of thread contexts. These also are merged in federated mode. For simplicity, we assume that each MSHR and store buffer only covers its half of the merged cache. On a cache miss the MSHRs of both sides have to be checked for a possible cache miss in progress. [4]. Since the underlying cores are designed to exploit large amounts of MLP, we assume that these resources will be larger than necessary for the federated core and we treat them as ideal in our simulations.

5.3 Translation Lookaside Buffers

Since the underlying cores of the federated architecture are already highly multi-threaded, we assume that the individual instruction and data TLBs are large enough to effectively support a single-threaded OOO core. Thus, when federated, the OOO core only uses the TLBs from one of the two cores. For simplicity, this assignment is made statically at design time.

6 Memory Alias Table

6.1 Motivation

Traditional Load-Store Queues are used for enforcing correct ordering between loads and stores which 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), proposed by Roth in [28], and the work by Garg et al. [13]. Contrary to this previous work, the MAT only detects memory order violations and does not provide a mechanism for forwarding store results to younger loads, eliminating the need for a forwarding network which can deal with multiple (partial) matches. Since previous work has shown that store-to-load forwarding is rare even in large OOO cores [13] [28], omitting the forwarding network provides considerable area savings with minimal performance loss.

Memory order violations must be treated as branch mispredictions and re-executed. Unlike in [33], 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 accurate store-to-load forwarding is needed to exploit the available machine resources, but can be omitted from our small design.

6.2 Concept of the Memory Alias Table

Before explaining the operation of the MAT, we should clarify our usage of certain terms. When discussing program order, we refer to instructions as earlier or later; when discussing actual execution order, we refer to instructions as younger or older.

The MAT operates as follows: each load places a token in a address-indexed hash table, which is removed (becomes invalid) when the load commits. Each store checks the hash table at commit for a token from a younger load which is still in the pipeline. Any store finding a valid token when it is committing knows that the token is from a later load and signals a memory order violation. The store does not need to cause an immediate pipeline flush but instead leaves an exception token in the table when it commits. The offending load will discover this exception token during commit when it invalidates its token in the hash table. The load can then either replay or cause a pipeline flush.

The hash table proposed by Garg et al. utilizes the same basic concept as the MAT, while the SVW 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. Other proposals [30] used simple counting bloom filters, but could not determine the relative age of a load or store.

[4] Correct store ordering is guaranteed, since each store can only appear in one of the two store buffers

Parameter	scalar	2-way	4-way	all
Active List	none	32	128	
IQ	none	16	32	
LSQ	none	16	64	
Data Cache	8KB	16KB	32KB	3 cycles
Instruction Cache	16KB	32KB	32KB	3 cycles
Unified L2	256KB	256KB	2MB	14 cycles
BTB	none	512	4K	4-way
Dir Pred	not-taken	2K bimodal	16K tournament	
Memory	100 Cycles, 64-Bit			

Table 4: Simulator parameters for the simulated traditional cores. Note that the federated core has different structures as explained in Section 4.

The MAT uses a simpler approach: each load increments a simple counter when it executes and decrements the same counter when it commits. Stores check the MAT only when they commit. Since any earlier load will have removed 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 later load in the pipeline with which it potentially aliases. (Previous proposals had the store check their equivalent of the MAT as soon as the address generation for the store was complete. They thus had no way of telling if the aliasing load(s) were later than the store or not; they could only determine that they were older.)

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 [5] in [33]. 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. The sharing of the upper bits can be considered the inverse of sharing the LSB in certain branch predictor tables [31]. We show in Section 8 that sharing all but the LSB between multiple counters is a feasible approach, as it introduces very few false positive memory order violations.

6.3 Dealing with Coherence and Consistency

To enforce a memory consistency model 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 the exception bit for its entry or entries. Any load to this location which is in the window when this occurs will force a flush of the pipeline. [6]

7 Simulation Setup

We evaluate our design using a simulator based on the SimpleScalar 3.0 framework [5] with Wattch extensions [4]. For the OOO cores our simulator models separate integer and floating point issue queues, load-store queues and active lists. 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. Our simulator allows loads to issue in the presence of unresolved stores. In the case that a memory order violation is later detected, the pipeline is flushed when the offending load attempts to commit.

[5] The SSN can be smaller than 16 bits, but since overflowing the SSN requires a pipeline flush and a reset of the hash table, a smaller SSN leads to lower performance.

[6] Any load committing in the same cycle as the cache coherency event can ignore it, since it is assured to have received its value before the event. Any committing load which decrements the counter to zero can reset the exception bit, since no loads which 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 exception bit at commit can still commit, since it cannot have received the wrong value in any combination of events. This load can set a second bit (shared across the whole table), to indicate that later stores are not the first to have seen the exception bit. This bit is reset at all pipeline flushes.

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 max power, which is closer to recently reported data [24].

We use the full SPEC2000 suite with reference inputs compiled for the Alpha instruction set. The Simpoint [34] toolkit was used to select representative 100 million instruction traces from the overall execution of all SPEC2000 benchmarks. For each run the simulator was warmed up for 10 million instruction before statistics were kept 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 simulation parameters for the different cores we compare a federated core against are listed in Table 4. 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 don't have exclusive use of the whole L2[7].

8 Results

8.1 Evaluating Design Tradeoffs

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 a dedicated, traditional 2-way OOO core with the same execution resources and caches as the federated organization. The dedicated core has a frontend pipeline with two fewer stages due to the lack of extra wiring overhead, a 512 entry, 4-way set associative BTB, and a 16 entry traditional LSQ.

Table 5 shows that restricting the number of subscription slots in each IQ entry has very little impact on overall performance. We attribute this to the well known fact [6] that the majority of dynamic instructions have only a single consumer, 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 cores has two subscription slots. Figure 3 shows the scaling behavior of the subscription-based IQ compared to a traditional organization. Additionally, we also show 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 the results in [29] for small IQ sizes. The largest combination of IQ and AL shows only a 1.1% difference in absolute performance between the best and worst combinations.

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 4 shows the performance of the baseline core using either a 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 SVW and 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.

As mentioned previously in Section 6.2, the MAT can save even more hardware by sharing most bits of each counter among neighboring entries in the hash table. Table 6 shows the impact on performance as we increase the number of counters sharing one set of upper bits. While the performance impact is minimal, the numbers are noisy, since intuitively more sharing should produce more false positives in the hash table and therefore lower performance. For the federated core, we share one set of upper bits between eight entries, so each entry only uses 1 + $\frac{4}{8}$ bits for the counter and an additional $\frac{1}{8}$ bit for the shared exception bit.

Figure 5 shows the impact on performance of the individual design changes of the federated core. Each energy saving or lower complexity feature is turned OFF individually to show its (negative) impact on overall performance. To separate out the impact of those features which can be applied a traditional OOO core from the extra constraints imposed by federating two scalar cores, the two limitations which are exclusive a federated cores are shown on the left of the figure. The "lightweight" core (please also see Section 8.3) is a dedicated 2-way OOO core with all the energy

[7]We have 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%

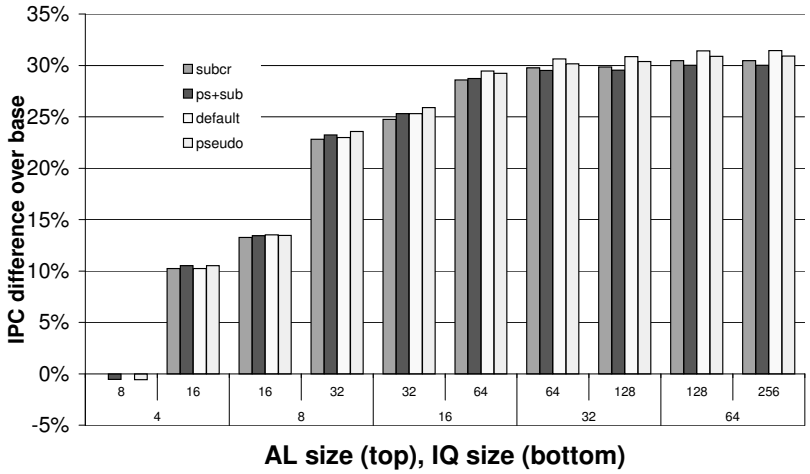

Figure 3: Increase in arithmetic mean IPC for different IQ designs as the sizes of the IQ and the AL are increased. The default configuration uses a CAM-based IQ with oldest-first scheduling. The percent difference is in comparison to a default configuration with a 4-entry IQ and an 8-entry AL.

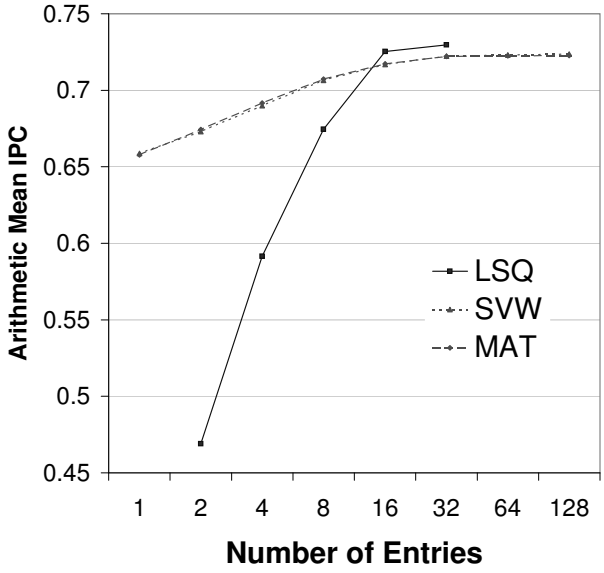

Figure 4: Scaling of arithmetic mean IPC for LSQ, SVW, and MAT as the number of entries is increased. The lines for the SVW and MAT are almost indistinguishable.

# of Slots	Change in IPC
1	-0.71%
2	-0.34%
4	-0.04%
8	0.00%

Table 5: Impact on arithmetic mean IPC of number of subscriber slots in the IQ.

Sharing Degree	Change in IPC
2	-0.46%
4	+0.02%
8	-0.05%
16	-0.18%

Table 6: Impact on arithmetic mean IPC of sharing the higher order bits of each counter in the MAT.

Core	Size in mm^2
1-way in-order	1.914
Federated OOO	3.970
Lightweight 2-way OOO	3.945
2-way OOO	5.067
4-way OOO	11.189

Table 7: Estimated sizes for different core types in 45nm technology.

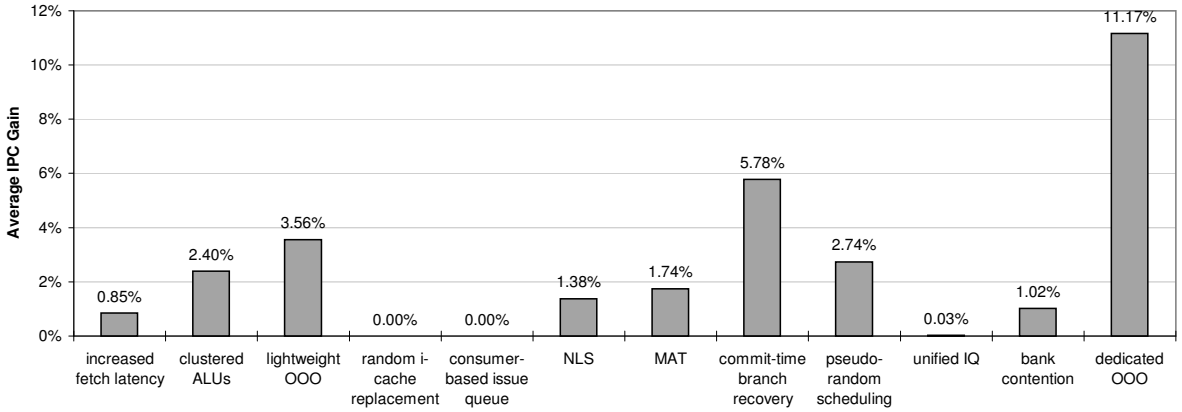

Figure 5: We show the performance impact of each individual feature by turning them OFF individually. The base case everything is compared against is the federated core as we have described it, and the relative increase in average IPC across all Spec2000 benchmarks is shown. The lightweight core shown on the left has all the limitations of the federated core except for the longer frontend pipeline and extra forwarding delay because of alu clustering, which are shown next to it.

saving features of the federated core applied. While most of the individual limitations have only a very small effect on performance, commit time branch decreases average IPC by over 5%.

8.2 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 extra area and performance, but would also like to discuss some alternative design choices using the techniques we have presented which 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 (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. We did not use this configuration in our final analysis because we wanted to err on the side of the simplest design. Clearly this would slightly improve federation's performance and energy efficiency.

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 I-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 I-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 from having to read out only one way most of the time, but reduce the NLS from 6,144 bits to 1,536 bits. While the performance impact of moving branch prediction to the decode stage is only 0.5%, stalling on non-return indirect branches affects some programs significantly.

8.3 Area Impact of Federation

Estimating the sizes of different types of cores and the area overhead needed for 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

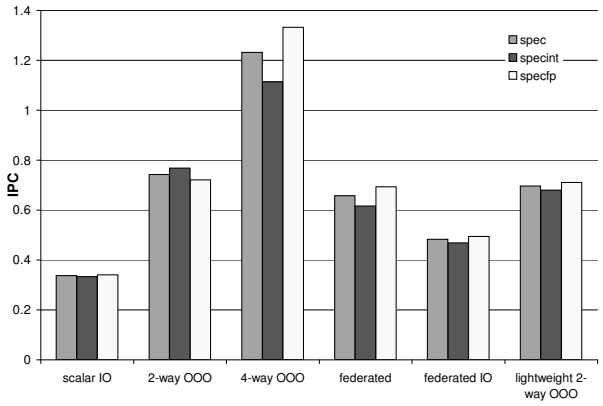

Figure 6: IPC for six different processor configurations across (a) integer and (b) floating point Spec2000 benchmarks.

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. For the federated core we included the area of the extra wiring overhead assuming the same wire pitch and spacing as in [19]. These final area estimates are shown in Table 7. 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 in [8], even though our assumptions and baseline design are somewhat different.

The area of the federated core was calculated by adding the areas of all the major new functional units to the area of two scalar in-order cores. We estimated the area needed by the major inter-core wiring listed in Table 3 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 280 nm wire pitch as used in [19]. In contrast to that 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 less than $0.05mm^2$, since the wires do not have to cross over multiple large cores, but only connect two immediately adjacent small cores. We assume that the additional logic in Table 2 is small enough to be negligible or already incorporated in the areas of the new functional units.

We attempted to estimate the area overhead as precisely as possible. However, the important point here is not the exact number but the order of magnitude. We contend that even if the actual area overhead were much more than our estimate, federation would still be an attractive approach for markets where limited thread count is important. For example, if the overhead were 10% per pair of cores instead of 3.7%, adding federating capability to a 32 core system would cost 1.6 scalar cores but would approximately double performance for each thread running in federated mode. A single OOO core costs more (2.65 scalar cores) and helps much less: federation can boost performance for 16 threads in such a system while the OOO core, even with SMT, would at best accommodate 2-4 threads!

8.4 Overall Performance and Energy Efficiency Impact of Federation

The overall performance of 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 the two scalar cores; dedicated 2-way OOO and 4-way OOO cores; and a dedicated 2-way OOO core, designated lightweight OOO, using the new functional units of the federated core, but having no extra ALU bypass latency, nor the extra frontend stages due to wiring overhead. This core shows the energy savings possible in a dedicated core using the same techniques as the federated core.

The resources to each core are listed in Table 4. The federated in-order core has the same resources as the 2-way OOO core except for those related to OOO execution. The overall performance of these cores is shown in Figure 6, with their average power consumption shown in Figure 8. The 4-way OOO core achieves about twice the IPC of the federated core, but uses about three times the power, while the dedicated 2-way OOO core achieves 12.9% higher performance than the federated core but uses 30.1% more power. The dedicated in-order core and the federated in-order core have substantially lower performance, which is not fully offset by their lower power consumption. This can be partially attributed to the fact that all cores—except for the 4-way OOO core which has larger caches—have similar

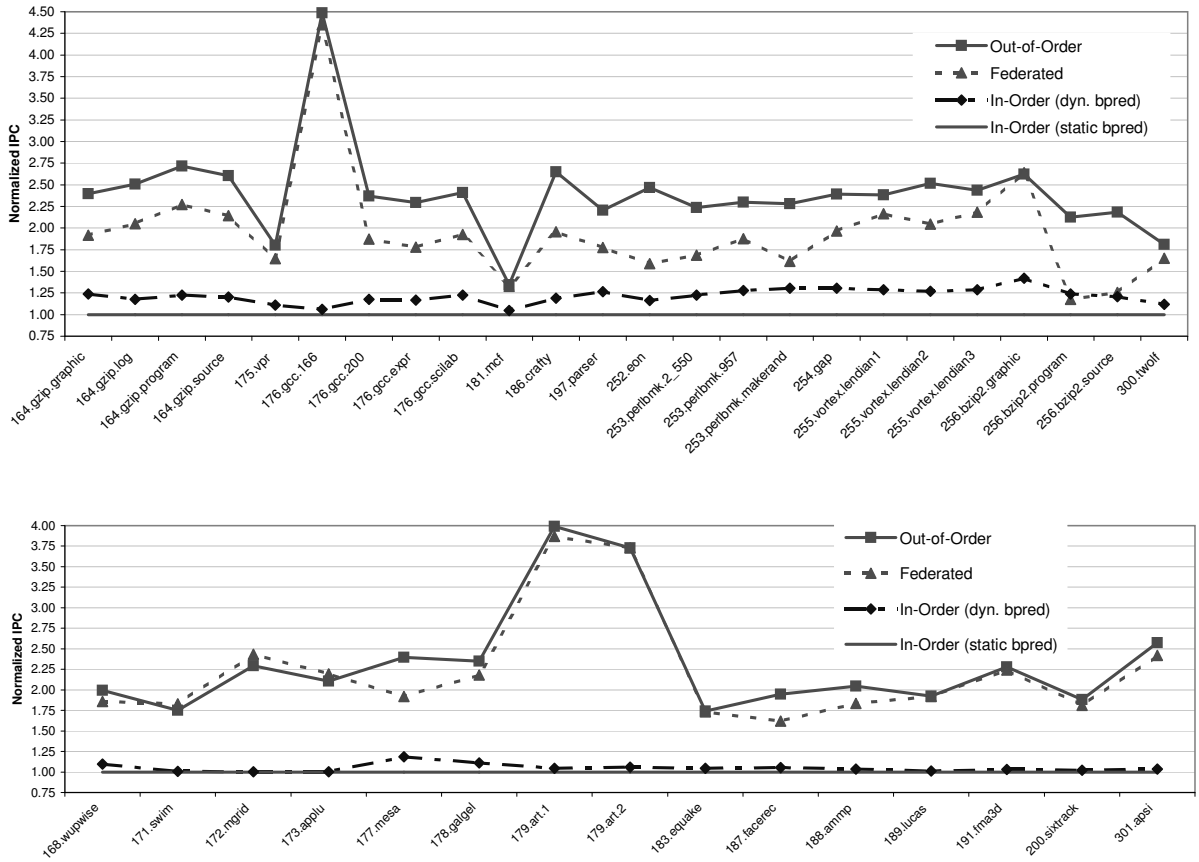

Figure 7: Normalized IPC for four different processor configurations across integer and floating point Spec2000 benchmarks. The OOO core is a dedicated 2-way core and the in-order cores are the baseline scalar cores with dynamic or static branch prediction. IPC is normalized to the IPC of the in-order core with static branch prediction. Note that the IPC axis begins at 0.75.

amounts of leakage in their caches and that the savings in active power are offset to some degree by the static leakage power.

Figure 7 shows the performance of different configurations for each trace in SpecInt and SpecFP. The benefits of federation (and even OOOE) are not evenly distributed. A notable example is the performance of the federated core on two of the bzip2 traces. Both of these traces generate a significant number of memory order violations, which leads to a large number of pipeline flushes. It should be noted that this is not an aliasing problem in the hash table, but a consequence of allowing loads to execute earlier than in configurations not using the MAT. Delaying loads by a few cycles would eliminate a large fraction of these pipeline flushes.

Figure 8 shows relative power requirements of the various organizations, and Figure 9 shows that the average $BIPS^3/Watt$ is useful for evaluating the energy efficiency of the different cores[8]. The high-performance 4-way OOO core has a large advantage over the smaller cores in energy efficiency, because it is able to use its higher power to achieve substantially better performance. The dedicated 2-way OOO core has better efficiency than the federated OOO core in SpecInt, but lower efficiency in SpecFP. The two in-order cores have the lowest energy efficiency, even though they have the lowest absolute power consumption. This is mostly due to leakage power, which penalizes cores with longer execution times (lower performance). To measure both the power- and area-efficiency of the different

[8] $\frac{BIPS^3}{Watt}$ 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.

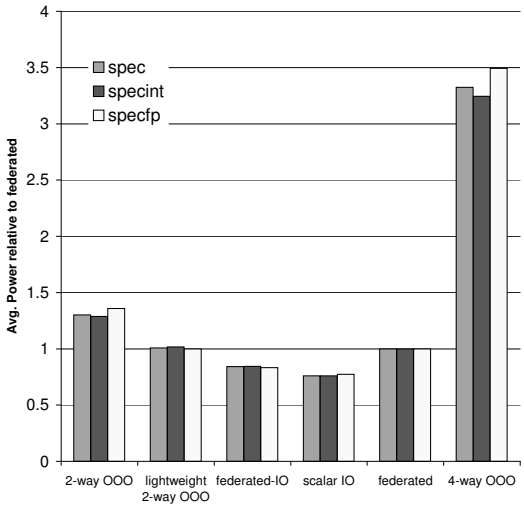

Figure 8: Normalized arith. mean power usage for six different processor configurations and for SpecInt, SpecFP and the whole SPEC2000 suite.

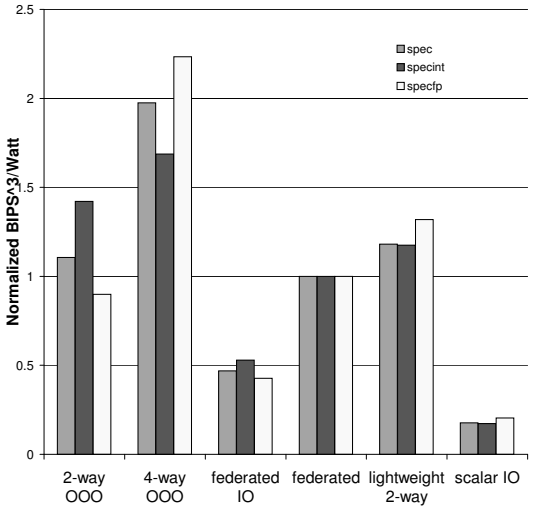

Figure 9: Normalized arith. mean $\frac{BIPS^3}{Watt}$ for six different processor configurations for SpecInt, SpecFP and the whole SPEC2000 suite.

cores, Figure 10 shows the $\frac{BIPS^3}{Watt \cdot mm^2}$ of the different configurations. The purpose of this metric is to account for the area cost of attaining a certain $\frac{BIPS^3}{Watt}$ 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 underlying scalar cores. We are investigating a metric based on *extra* area required by a particular organization. Nevertheless, in terms of $\frac{BIPS^3}{Watt \cdot mm^2}$, lightweight 2-way OOO has the best outcome. This shows that much of the benefit of federation actually comes from making the OOO engine leaner. However, because this comes at the area expense of a dedicated core and would cost at least 2 scalar cores, we argue that federated 2-way OOO is preferable, because it comes at the vanishingly small area overhead of only 3.7% per *pair* of scalar cores (i.e., less than 1/25 of a scalar core). In terms of $\frac{BIPS^3}{Watt \cdot mm^2}$, federation outperforms the dedicated, traditional 2-way OOO core by 13.3% and the 4-way core by 30%.

9 Conclusions and Future Work

Manycore chips of dozens or more simple but multithreaded 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 92.4% better than the in-order core, 30% lower average power than a dedicated 2-way OOO core, and competitive energy efficiency compared to a 2-way OOO core. Using a consumer-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, OOO core, as the "lightweight OOO" results show. However, even a lightweight dedicated OOO core would still come at a high cost in area. The most important advantage of federation is that it can be added without sacrificing the ability to use those constituent scalar cores as multi-threaded, throughput-oriented cores. Federation requires several new structures, but with very low area overhead—less than 2KB of new SRAM tables and less than 0.25KB of new register-type structures in the pipeline per *pair* of cores—only 3.7% area overhead per pair. Put another way, this means that for a set of 32 cores, the area of federation for each pair only adds an aggregate area equivalent to 0.59 cores or 0.373 MB of L2 cache. As a result, federation actually provides greater energy efficiency per unit area—specifically, 13.3% better $\frac{BIPS^3}{Watt \cdot mm^2}$ than a dedicated 2-way OOO core, and 30% 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

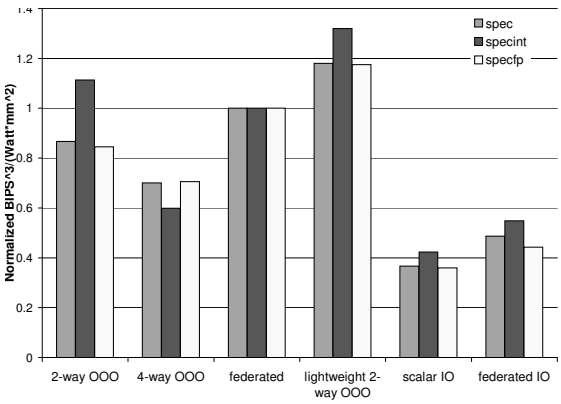

Figure 10: Normalized arith. mean $\frac{BIPS^3}{Watt \cdot mm^2}$ for six different processor configurations for SpecInt, SpecFP and the whole SPEC2000 suite.

and 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 and deal with workloads that have different amounts of parallelism during different phases of execution. Furthermore, federation does not require any recompilation to take advantage of the extra performance it offers. Even if the baseline in-order cores are different from those assumed in this paper, the techniques and benefits of federation would still apply.

We are currently working to extend the single-core analysis presented in this paper in order to characterize the performance and power impact of federation on an entire multicore or manycore processor under various workload assumptions. This will allow us to study the performance and energy improvements provided by federation as the number of cores scales, and account for the accuracy and overhead of runtime decisions to federate and unfederate pairs of cores. Previous work on scheduling for heterogeneous cores [23] is likely to be applicable for these reconfiguration decisions. Such decisions may also be a good target for dynamic optimization.

Acknowledgements

This grant was supported by NSF grant nos. CCR-0306404, CNS-0509245, and CNS-061527. We would also like to thank Michael Frank, Doug Burger, Mircea Stan, and Jeremy Sheaffer for their helpful feedback.

References

[1] K. Aingaran, P. Kongetira, and K. Olukotun. Niagara: a 32-way Multithreaded Sparc Processor. *Micro, IEEE*, 25, 2005.
[2] AMD. ATI Radeon HD 2900 Technology GPU Specifications.
[3] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 18 2006.
[4] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a Framework for Architectural-Level Power Analysis and Optimizations. In *ISCA*, pages 83–94, 2000.
[5] D. Burger, T. M. Austin, and S. Bennett. Evaluating Future Microprocessors: The Simplescalar Tool Set. Technical Report CS-TR-1996-1308, University of Wisconsin-Madison, 1996.
[6] J. A. Butts and G. S. Sohi. Characterizing and Predicting Value Degree of Use. In *MICRO*, pages 15–26, 2002.
[7] B. Calder and D. Grunwald. Next Cache Line and Set Prediction. In *ISCA*, pages 287–296, 1995.
[8] D. Carmean. Future CPU Architectures: The Shift from Traditional Models. Intel Higher Education Lecture Series.
[9] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture Optimizations for Exploiting Memory-Level Parallelism. *SIGARCH Comput. Archit. News*, 32(2):76, 2004.
[10] J. D. Davis, J. Laudon, and K. Olukotun. Maximizing CMP Throughput with Mediocre Cores. In *PACT*, pages 51–62, 2005.

[11] R. Dolbeau and A. Seznec. CASH: Revisiting Hardware Sharing in Single-Chip Parallel Processors. *J. of Instruction-Level Parallelism*, 6, 2004.
[12] J. Farrell and T. Fischer. Issue logic for a 600-MHz out-of-order execution microprocessor. *Solid-State Circuits, IEEE Journal of*, 33(5):707–712, 1998.
[13] A. Garg, F. Castro, M. Huang, D. Chaver, L. Pinuel, and M. Prieto. Substituting Associative Load Queue with Simple Hash Tables in Out-of-Order Microprocessors. In *ISLPED*, pages 268–273, 2006.
[14] A. Glew. MLP yes! ILP no! ASPLOS Wild and. Crazy Idea Session98.
[15] E. Grochowski, R. Ronen, J. Shen, and H. Wang. Best of Both Latency and Throughput. In *ICCD*, pages 236–243, 2004.
[16] P. Hester. 2006 AMD Analyst Day Presentation.
[17] H. P. Hofstee. Power Efficient Processor Architecture and The Cell Processor. In *HPCA*, pages 258–262, 2005.
[18] M. Huang, J. Renau, and J. Torrellas. Energy-Efficient Hybrid Wakeup Logic. *Low power electronics and design*, 2002.
[19] E. İpek, M. Kırman, N. Kırman, and J. Martínez. Core Fusion: Accommodating Software Diversity in Chip Multiprocessors. In *ISCA*, 2007.
[20] T. Johnson and U. Nawathe. An 8-core, 64-thread, 64-bit Power Efficient Sparc Soc. In *ISPD'07*, pages 2–2, 2007.
[21] R. Kessler, E. McLellan, and D. Webb. The Alpha 21264 Microprocessor Architecture. In *ICCD*, 1998.
[22] R. Kumar, N. Jouppi, and D. Tullsen. Conjoined-Core Chip Multiprocessing. In *MICRO*, pages 195–206, 2004.
[23] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *ISCA*, page 64, 2004.
[24] F. J. Mesa-Martinez, J. Nayfach-Battilan, and J. Renau. Power Model Validation Through Thermal Measurements. In *ISCA*, 2007.
[25] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors. In *HPCA*, page 129, 2003.
[26] NVIDIA. NVIDIA CUDA Compute Unified Device Architecture Programming Guide. Technical report, NVIDIA Corporation, Feb. 2007. Version 0.8.
[27] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, and T. J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, Aug. 2005.
[28] A. Roth. Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization. In *ISCA*, pages 458–468, 2005.
[29] P. G. Sassone, J. R. II, E. Brekelbaum, G. H. Loh, and B. Black. Matrix Scheduler Reloaded. In *ISCA*, 2007.
[30] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler. Scalable Hardware Memory Disambiguation for High ILP Processors. In *MICRO*, page 399, 2003.
[31] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides. Design Tradeoffs for the Alpha EV8 Conditional Branch Predictor. In *ISCA*, pages 295–306, 2002.
[32] T. Sha, M. M. K. Martin, and A. Roth. Scalable Store-Load Forwarding via Store Queue Index Prediction. In *MICRO*, pages 159–170, 2005.
[33] T. Sha, M. M. K. Martin, and A. Roth. NoSQ: Store-Load Communication without a Store Queue. In *MICRO*, pages 285–296, 2006.
[34] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *ASPLOS*, pages 45–57, 2002.
[35] S. Subramaniam and G. H. Loh. Fire-and-Forget: Load/Store Scheduling with No Store Queue at All. In *MICRO*, pages 273–284, 2006.
[36] M. Tremblay and J. M. O'Connor. UltraSparc I: A Four-Issue Processor Supporting Multimedia. *IEEE Micro*, 16(2):42–50, 1996.
[37] H. Zhong, S. A. Lieberman, and S. A. Mahlke. Extending Multicore Architectures to Exploit Hybrid parallelism in Single-thread Applications. In *HPCA*, pages 25–36, 2007.