Federation: Repurposing Scalar Cores for Out-of-Order Instruction Issue

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

ABSTRACT

Future SoCs will contain multiple cores. For workloads with significant parallelism, prior work has shown the benefit of many small, multi-threaded, scalar cores. For workloads that require better single-thread performance, a dedicated, larger core can help but comes at a large opportunity cost in the number of scalar cores that could be provisioned instead. This paper proposes a way to repurpose a pair of scalar cores into a 2-way out-of-order issue core with minimal area overhead. "Federating" scalar cores in this way nevertheless achieves comparable performance to a dedicated out-of-order core and dissipates less power as well.

Categories and Subject Descriptors

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

General Terms

Design, Performance

Keywords

Federation, CMP, multicore, out-of-order

1. INTRODUCTION

Embedded systems composed of multiple processing cores are becoming increasingly popular. There is a tradeoff between the complexity 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.* [5] and Carmean [4] show that maximum aggregate throughput is achieved by using a large number of highly multi-threaded scalar cores. However, for applications with limited parallelism or tasks that require low latency, performance is 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 a dedicated out-of-order (OO) core to the existing scalar cores. Unfortunately, for a given die area, this dedicated core comes at the cost of multiple, multi-threaded scalar cores, reducing the aggregate throughput of the system. Simultaneous multi-threading (SMT) of an OO core cannot compensate for the lost thread capacity because the area and power cost per context is much higher than with scalar cores [5].

Instead, we show that a dedicated out-of-order core is not needed. *Federation* uses very modest additional hardware to allow a pair of scalar cores to act as a 2-way OO core. This achieves 89% the performance of a dedicated 2-way OO core without the associated area cost. A dedicated 2-way OO core costs 2.65 multi-threaded scalar cores while the extra hardware to federate a pair of scalar cores into an OO core costs less than 0.08 scalar cores.

The key insights that make federation work are that it is possible to approximate traditional out-of-order issue with much more efficient hardware structures, replacing CAMs and broadcast networks with simple lookup tables; and that these "lightweight" OO structures can be placed between a pair of scalar cores and use the fetch, decode, register-file, cache, and datapath of the scalar cores to achieve an ensemble that is competitive in performance with an OO core. Merely using these lightweight OO structures in a dedicated OO core is not sufficient; this only reduces the area cost of the dedicated core from 2.65 to 2.06 scalar cores. The key is that *appending* these lightweight structures between two scalar cores provides OO execution with minimal cost.

Federated cores are best suited for workloads which 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.

2. RELATED WORK

Previous work on combining several smaller cores into a single larger, more capable core was performed by İpek *et al.* [8] and Kim *et al.* [10]. These designs focus on maximizing performance rather than minimizing area or power and assume that the underlying cores already support OO execution. The Voltron architecture from Zhong *et al.* [15] allows multiple in-order VLIW cores of a chip multiprocessor (CMP) to combine into a larger VLIW core. Both Kim and Zhong's work require special compilers and ISAs. Federation does not assume an advanced compiler and is applicable to existing RISC, CISC, and VLIW ISAs. All these techniques, including federation, are forms of *dynamic* cores as advocated by Hill and Marty [6].

3. OUT-OF-ORDER PIPELINE

To demonstrate the benefits of our approach, we augment a baseline multicore system to support federation. We assume that the baseline architecture is throughput-oriented and is therefore composed of multi-threaded, scalar cores. Specifically, we assume that each core contains four hard-

DAC 2008, June 8-13, Anaheim, California, USA

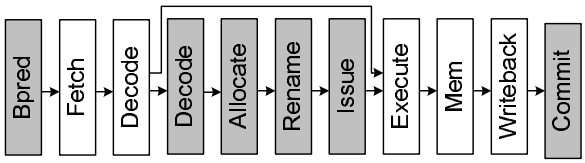

Figure 1: The pipeline of a federated core, with the new pipeline stages in shaded boxes.

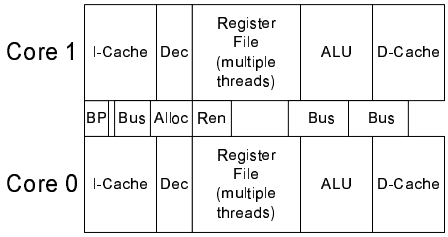

Figure 2: A simplified floorplan showing the arrangement of two in-order cores with the new structures necessary for federation in the area between the cores.

ware thread-contexts. But as Table 4 shows, the area overhead of multi-threading is not very large, making federation an attractive option even for single-threaded cores.

The main goal of federation is to add OO execution capability to the existing in-order cores with as little area overhead as possible. Thus, each federated core is relatively simple compared to many dedicated OO implementations. Specifically, each federated core is single-threaded 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 shaded boxes. A simplified view of a possible floorplan is shown in Figure 2.

In order to minimize 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 1 lists the sizes of the new structures required to support OO execution, as well as whether or not each structure could potentially be implemented by re-using the existing register file of the underlying multi-threaded core, while Table 2 lists the new wiring required. The following subsections provide an overview of the operation of each pipeline stage in the federated core, along with a discussion of the design tradeoffs that were made. Additional details appear in [14].

Branch Prediction: Branch prediction is implemented using Next Line and Set prediction (NLS) [3, 9] instead of a branch target buffer.

Fetch: The fetch stage starts by receiving a predicted cache line from the 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 request to the instruction cache. When federated, the individual instructions caches of the two cores are combined into a single cache with double the associativity and random replacement.

Decode: 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

Structure	Size (Bits)	Type	Reuses RF
Branch Predictor (NLS)	6,144	SRAM	No
Branch Predictor (Bimodal)	4,096	SRAM	No
Unified Register File	4,096	Reg	Yes
Smaller Structures	3056	both	No
Worst Case Total (Bits)	10,496 SRAM 6,844 Register		
Assumed Total (Bits)	10,496 SRAM 1,852 Register		

Table 1: Area estimates for the new structures added to the baseline in-order processor. The worst case total assumes that none of the structures can reuse the register file.

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	64 bits

Table 2: The size of wires that must be added to the baseline core in order to support federation.

the instruction to the second core, buffering the first instruction in a pipeline register.

Allocate: During the allocate stage, each instruction checks for space in several structures required for OO execution and stalls the frontend if space is not available.

Rename: 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.

Issue: The area and power constraints of our design prevent the implementation of a traditional CAM-based issue queue (IQ). Instead, 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 scheme evaluated by Huang *et al.* [7].

In addition to the usual opcode, register ids, and immediates, each IQ entry also holds several consumer id fields and two ready bits; the ready bits are set when the left and right operands become available, respectively. When issued, an instruction checks its consumer fields and sets the appropriate ready bits for its consumer(s). If both input operands are ready (i.e., both ready bits are set), the ready signal for that entry is sent to the scheduler. The number of consumer fields per entry is a design choice; we found that having only two fields per entry had a negligible impact on performance.

The normal scheduling logic for an out-of-order processor attempts to issue the oldest instructions first. Achieving this age-prioritization is costly in terms of both area and power. Instead, we implement a much simpler pseudo-random scheduler [13] which uses a simple static priority encoder and does not take into account the age of instructions. For a small instruction window, this simplified scheduler only reduces performance by around 1%.

For simplicity, we use a naïve scheduling algorithm that only schedules instructions on core one if core zero is already busy. Additionally, all load and store instructions are issued

only to core zero to avoid the need to maintain memory ordering across the two cores.

Execute: Each instruction executes normally on the core to which it was assigned during the issue stage. The only change to the bypass network is the addition of circuitry for copying the result to the register file of the opposite core.

Memory Access: The two individual data caches are merged into a unified cache in the same way as the instruction caches, by doubling the associativity and assigning half of the ways of the unified cache to each individual cache.

To eliminate the LSQ, we employ a similar approach to the Store Vulnerability Window (SVW) [12] by using a Memory Alias Table (MAT), which provides approximately the same functionality and performance as the SVW while using an order of magnitude fewer bits per entry. We do not allow memory bypassing; instead, we flush the pipeline when we detect that a load and a store instruction which access the same address have executed out-of-order. The MAT is a simple hash table indexed by memory address, where each entry is a small counter which is incremented by a load on issue and decremented on commit. Stores check the MAT on commit and cause a pipeline flush if they find their counter to be non-zero, which indicates that a younger load has collided with the store and received a stale value. Sharing the most significant counter bits between neighboring entries reduces the cost to less than two bits per entry. We found that a MAT reduces performance by only 2% compared to a large LSQ—for further details, see [14].

Write Back: Similar to the Alpha 21264 [9], all writes go to the register files on both cores to avoid generating explicit copy instructions for consumers on the other core.

Commit: Branch mispredictions are handled at commit time, obviating the need to maintain multiple snapshots of the speculative rename table or walk the active list in the case of a branch misprediction. Commit time branch recovery reduces performance by 5%. This cost can be reduced by handling mispredictions at branch resolution. Adding just two snaphshots of the rename table almost completely eliminates this performance degradation. However, since our focus is on simplicity, results in this paper assume commit time branch recovery for the federated core.

4. SIMULATION SETUP

We evaluate our design using a simulator based on the SimpleScalar 3.0 framework [1] with Wattch extensions [2]. Static power has been adjusted to be 25% of max power, which is closer to recently reported data [11]. We use the full SPEC2000 suite with reference inputs compiled for the Alpha instruction set. See [14] for more details.

5. RESULTS

The federated core is compared against: 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; and dedicated 2-way and 4-way OO cores. The parameters of the non-federated cores are shown in Table 3.

5.1 Area Impact of Federation

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 final area estimates for the different cores, shown in Table 4, were calculated from the areas of their constituent units, scaled with capacity and port numbers. The "lightweight" core is a dedicated 2-way OO core with the same area-efficient hardware resources as the federated core. It is interesting to note that the ratio of the area of the 4-way OO core to the area of the in-order core is close to the 5-to-1 ratio in [4], even though our assumptions and baseline design are somewhat different.

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

Table 3: Simulation parameters for different cores.

Core	Size in mm^2
1-way in-order	1.739
1-way in-order MT	1.914
Federated OO	3.970
Lightweight 2-way OO	3.945
2-way OO	5.067
4-way OO	11.189

Table 4: Estimated core sizes in 45nm technology.

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 for the major inter-core wiring listed in Table 2 by calculating the width of the widest new unit (the integer and floating point rename tables laid out side-by-side) and assuming the same 280nm wire pitch used in [8].

Based on the final area estimate, the overall area overhead of federation is only 3.7% per pair of scalar cores. We attempted to estimate this overhead as precisely as possible; however, the point here is not the exact number, but the order of magnitude. Suppose, for example, that the overhead were 10% instead of 3.7% per pair. Then federating 32 cores (16 pairs) would cost 160%—less than two scalar cores—but could approximately double performance for up to 16 threads. A single OO core costs more (2.65 scalar cores) and helps much fewer threads (even with SMT, the OO core would at best accommodate 2–4 threads).

5.2 Performance-Energy Impact of Federation

The overall performance and average power consumption of the different core types is shown in Figure 3. The 4-way OO core achieves about twice the IPC of the federated core but uses about three times the power, while the dedicated 2-way OO 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 OO core which has larger caches—have similar amounts of leakage in

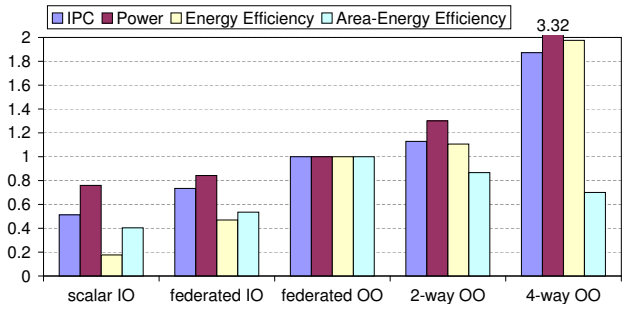

Figure 3: Arithmetic mean IPC, power, $\frac{BIPS^3}{Watt}$, and $\frac{BIPS^3}{Watt \cdot mm^2}$, all normalized to federated OO.

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

Figure 3 also shows the average energy efficiency, expressed in $BIPS^3/Watt$, for the different core types. The high-performance 4-way OO 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 OO core has better efficiency than the federated OO 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. Once again, this is mostly due to leakage power, which penalizes cores with longer execution times.

To measure both the power- and area-efficiency of the different cores, Figure 3 also 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. In terms of $\frac{BIPS^3}{Watt \cdot mm^2}$, federation outperforms the dedicated, traditional 2-way OO core by 13.3% and the 4-way core by 30%.

6. CONCLUSIONS

Multicore embedded systems composed of many simple but multi-threaded cores will need the ability to cope with limited thread count by boosting their per-thread performance. This paper shows how 2-way OO capability can be built from very simple, in-order cores, with 92.4% better performance than the in-order core, 30% lower average power than a dedicated 2-way OO core, and competitive energy efficiency compared to a 2-way OO 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 OO pipeline work. 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, for a set of 32 scalar cores, augmenting each pair to support federation only adds an aggregate area equivalent to 0.59 cores or 0.373 MB of L2 cache. Federation provides greater energy efficiency per unit area than the dedicated cores—specifically, 13.3% better $\frac{BIPS^3}{Watt \cdot mm^2}$ than a 2-way OO core and 30% better than a 4-way OO core.

The option of adding federation removes the need to choose between high throughput with many small cores or high single-thread performance with aggressive OO 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 multicore chip to trade off throughput for latency on a very fine-grained level at runtime. Federation thus allows multicore SoCs to provide higher performance across a wider spectrum of workloads with different amounts of thread-level parallelism, as well as deal with workloads which exhibit different amounts of parallelism across different phases of execution.

7. ACKNOWLEDGMENTS

This work was supported by NSF grant nos. CCR-0306404, CNS-0509245, and IIS-0612049. We would also like to thank Michael Frank, Doug Burger, Mircea Stan, Jeremy Sheaffer and the anonymous reviewers for their helpful comments.

8. REFERENCES

[1] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Computer*, pages 59–67, Feb. 2002.

[2] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a Framework for Architectural-Level Power Analysis and Optimizations. In *ISCA*, 2000.

[3] B. Calder and D. Grunwald. Next Cache Line and Set Prediction. In *ISCA*, 1995.

[4] D. Carmean. Future CPU Architectures: The Shift from Traditional Models. Intel Higher Education Lecture Series.

[5] J. D. Davis, J. Laudon, and K. Olukotun. Maximizing CMP Throughput with Mediocre Cores. In *PACT*, 2005.

[6] M. D. Hill and M. R. Marty. Amdahl's Law in the Multicore Era. *IEEE Computer*. To appear.

[7] M. Huang, J. Renau, and J. Torrellas. Energy-Efficient Hybrid Wakeup Logic. In *ISLPED*, 2002.

[8] E. İpek, M. Kırman, N. Kırman, and J. Martínez. Core Fusion: Accommodating Software Diversity in Chip Multiprocessors. In *ISCA*, 2007.

[9] R. Kessler, E. McLellan, and D. Webb. The Alpha 21264 Microprocessor Architecture. In *ICCD*, 1998.

[10] C. Kim, S. Sethumadhavan, M. S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler. Composable Lightweight Processors. In *MICRO*, 2007.

[11] F. J. Mesa-Martinez, J. Nayfach-Battilan, and J. Renau. Power Model Validation Through Thermal Measurements. In *ISCA*, 2007.

[12] A. Roth. Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization. In *ISCA*, 2005.

[13] P. G. Sassone, J. R. II, E. Brekelbaum, G. H. Loh, and B. Black. Matrix Scheduler Reloaded. In *ISCA*, 2007.

[14] D. Tarjan, M. Boyer, and K. Skadron. Federation: Out-of-Order Execution using Simple In-Order Cores. Technical Report CS-2007-11, Dept. of Comp. Sci., Univ. of Virginia, Aug. 2007.

[15] H. Zhong, S. A. Lieberman, and S. A. Mahlke. Extending Multicore Architectures to Exploit Hybrid parallelism in Single-thread Applications. In *HPCA*, 2007.