

# Federation: Repurposing Scalar Cores for Out-of-Order Execution



Michael Boyer, David Tarjan, Kevin Skadron  
 University of Virginia Department of Computer Science  
 boyer@cs.virginia.edu  
 SRC Task 1607.001

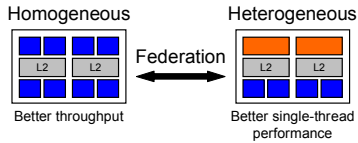


## Abstract

Manycore architectures with hundreds of threads are likely to use small, in-order cores with simple pipelines but multiple thread contexts per core. This approach maximizes throughput, but only with enough threads to keep most thread contexts occupied. To deal with limited thread counts, we describe how to *federate* a pair of neighboring in-order cores in order to form an out-of-order core at runtime. It can be accomplished with less than 2KB of extra hardware per pair while providing nearly double the performance of a single, scalar core and nearly the same performance as a traditional, dedicated 2-way out-of-order core. Federation thus allows a manycore architecture to provide good performance across a wider range of thread counts.

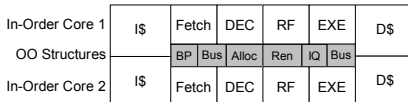
## Motivation

When designing a manycore processor, there is a fundamental tradeoff between the complexity of each core and the total number of cores that can fit in a given area. For high thread counts, a homogeneous architecture composed of simple cores provides the best performance; for lower thread counts, some dedicated heavyweight cores (a heterogeneous architecture) would be beneficial. How can we construct an architecture that performs well across a wide range of thread counts?



## Federation

Federation bridges this divide by allowing a baseline homogeneous architecture to dynamically add heterogeneity at runtime. It achieves this by allowing each pair of scalar in-order (IO) cores to *federate* to form a single 2-way out-of-order (OO) core. The diagram below shows an example layout of a pair of in-order cores with the additional OO structures between the two cores.



The main design goal of Federation is to achieve out-of-order execution with minimal added complexity. To this end, a number of traditionally complex OO structures were replaced with simpler, more area- and power-efficient designs.

## Subscription-Based Issue Queue

Instead of a traditional CAM-based issue queue (IQ), we use a simple table in which consumers explicitly "subscribe" to their producers.

Example: add r1,r2,r3  
 mult r3,r4,r5

1) Add is allocated

Slot	Instruction	Ready Bits	Subscriber Slots
1	add r1,r2,r3	0 0	- -
2	-	0 0	- -

2) Mult is allocated and subscribes to add

Slot	Instruction	Ready Bits	Subscriber Slots
1	add r1,r2,r3	0 0	2 -
2	mult r3,r4,r5	0 0	- -

The number of subscriber slots is a design choice. We have found that two slots per entry are sufficient to achieve performance matching that of a traditional IQ. In addition, replacing the traditional age-based instruction scheduler with a simpler, pseudo-random scheduler saves even more area while only reducing performance by 2.67% on average.

## Memory Alias Table (MAT)

We replace the traditional Load-Store Queue (LSQ) design with a much simpler Memory Alias Table (MAT), which allows loads to issue in the presence of unresolved stores and then detects memory order violations after they have occurred. The MAT is a hash table of small counters indexed by memory address. Upon execution, a load indexes into the table and increments the corresponding counter. Upon commit, a store indexes into the table and checks the counter value; if it is non-zero, it sets a flag. When the offending load reaches the commit stage, it detects this flag and is then flushed and re-executed. This approach ensures both correctness and forward progress.

Example: st 0xC1, r2  
 ld 0xC1, r1

Address	Counter	Flag
0	0	-
1	0	-

1) Load executes first (out-of-order) and increments counter

Address	Counter	Flag
0	0	-
1	1	-

2) Store commits, checks counter, sees non-zero value, and sets flag

Address	Counter	Flag
0	0	-
1	1	1

3) Load reaches commit, sees flag, flushes the pipeline, and re-executes

Address	Counter	Flag
0	0	-
1	0	-

The MAT uses much less area than a traditional LSQ but only reduces performance by an average of 1.71%.

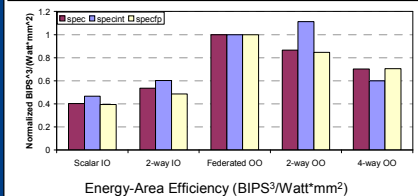
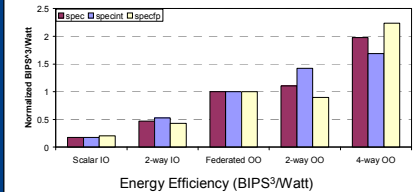
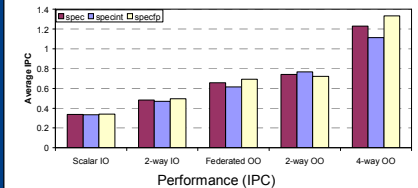
## Area Overhead

To estimate realistic areas for the different core types that we studied, we measured the sizes of the different functional units of an AMD Opteron processor in 130nm technology from a publicly available die photo and scaled the functional unit areas to 45nm. The final area estimates shown below were calculated from the areas of each core type's constituent units, scaled with capacity and port numbers. The scalar core cost for each OO core type is the number of in-order cores that would need to be re-typed in order to add one of those cores.

Core Type	Area (mm <sup>2</sup> )	Scalar Core Cost
1-way IO	1.74	-
Federated OO	3.97	0.07
2-way OO	5.07	2.65
4-way OO	11.19	5.85

Based on these estimates, the overall area overhead of federation is only 3.7% per pair of scalar cores.

## Results



## Future Work

- Study the impact of Federation at the system level.
- Model the area and power impact of Federation in greater detail.
- Investigate scheduling policies: deciding both when to federate and de-federate pairs of cores and how to map threads onto potentially heterogeneous cores.

## Related Publications

D. Tarjan, M. Boyer, and K. Skadron. "Federation: Repurposing Scalar Cores for Out-of-Order Instruction Issue." In *Proceedings of the 45th annual Conference on Design Automation (DAC)*, June 2008.

D. Tarjan, M. Boyer, and K. Skadron. "Federation: Out-of-Order Execution using Simple In-Order Cores." University of Virginia Department of Computer Science Technical Report CS-2007-11, August 2007.

## Technology Transfer

- Liaison interactions: Quarterly meetings with Srilatha Manne (AMD) and Perry Wang & Jamison Collins (Intel)
- Publications:
  - DAC 2008 (see above)
  - Longer technical report published on SRC website
  - Journal version currently in submission
- Presentation by Kevin Skadron at Intel, December 2007 (hosted by Pradeep Dubey)
- Internship at Intel, Summer 2007 (unrelated to this work)

## Acknowledgments

This work was supported by an SRC GRC Fellowship, SRC grant no. 2007-HJ-1607, and NSF grant nos. CCR-0306404, CNS-0509245, and IIS-0612049. We would also like to thank Michael Frank, Doug Burger, Mircea Stan, and Jeremy Sheaffer for their helpful comments.

## Biography



Michael Boyer was awarded the Bachelor of Science Degree in Computer Engineering from Union College in 2006. As an undergraduate he received several awards, including the Loughry Prize for best senior project in Computer Engineering. He participated in the pilot for the SRC Undergraduate Research Assistants Program, presenting his research at the Graduate Fellows Conference in 2004. He has also completed three internships with Intel. He is currently a graduate student at the University of Virginia, where he is researching adaptive computer architectures under the guidance of Professor Kevin Skadron. He was recently awarded an AMD/Mahboob Khan Fellowship from SRC.