Appears in the Proceedings of the 2010 Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures (PESPMA) held in conjunction with the 37th International Symposium on Computer Architecture (ISCA)

ReFLEX: Block Atomic Execution on Conventional ISA Cores

Mark Gebhart*
*Department of Computer Science
The University of Texas at Austin
mgebhart@cs.utexas.edu

Stephen W. Keckler*†
†Architecture Research Group
NVIDIA
skeckler@nvidia.com

Abstract

Modern multicore chips target thread-level parallelism at the expense of increasing instruction-level parallelism from single threaded programs. While recent work has attempted to construct a wide-ILP machine from multiple simple cores, these approaches suffer from ISA overheads or scalability challenges. In this paper, we describe an architecture that is inspired by the scalability and flexibility of the TFLEX architecture, yet elides the unorthodox ISA and the overheads that stem from its dataflow execution model. Our results focus on the tradeoff between near out-of-order execution (small out-of-order window within a block of instructions) and far out-of-order execution across blocks. Experiments indicate that a small out-of-order window combined with block-level speculation enables our proposed ReFLEX architecture to achieve comparable performance and flexibility as TFLEX yet with simpler cores and a more conventional ISA.

1 Introduction

With the end of clock-rate scaling, current and future processors must rely on concurrency to provide continued performance scaling. Modern multicore processors target thread-level parallelism through a small number of simultaneous-multithreaded cores and data-level parallelism through short-vector extensions such as SSE. GPUs target both thread and data-level parallelism via a large number of massively multithreaded but (mostly) scalar cores. In either case, instruction-level parallelism for single-threaded programs or procedures is no longer the focus of multicore systems.

While emerging programs and workloads have a much greater focus on parallelism, many programs still have regions in which few or even one thread is operating. A common program paradigm is to alternate between serial and parallel phases, expanding thread count at the beginning of a parallel region and contracting back to a single thread which may process data prior to subsequent work creation. In such cases, Amdahl's law can dominate. We seek to harness the resources from multiple simple cores to exploit ILP when there are insufficient threads to saturate a multicore system.

Exploiting dynamic expansion and contraction of parallel threads in a manner that effectively utilizes the machine requires an agile system that can create work and synchronize quickly. Further, the capability to dynamically aggregate processing cores to form larger logical processors can enable the hardware to adapt to changing demands of parallel and sequential execution. Recent work on flexible core architectures that intend to exploit these capabilities include Core Fusion [5], Federation [15], and Composable Lightweight Processors (TFLEX) [7]. Of these, TFLEX is the most scalable enabling aggregation of up to 32 cores and performance scaling to 8 cores (SPEC-INT) and 32 cores (SPEC-FP).

However, TFLEX has several drawbacks. First, it requires a decomposition of a program into program regions (hyperblocks) of a fixed maximum size. Second, the dataflow oriented instruction set that was designed to scale ILP across many cores incurs substantial overheads for data movement and copying. Third, the microarchitecture requires sizable storage structures (reservation stations) to deploy the dataflow instruction set; because the reservation stations hold instructions that have both executed and are waiting to execute they can incur power overheads.

In this paper, we propose ReFLEX, a more conventional ISA architecture inspired by the advantages and capabilities of TFLEX. ReFLEX employs a conventional RISC-style architecture, augmented with block atomic semantics and full predication, and a microarchitecture that enables block-atomic execution without restricting blocks to a fixed minimum or maximum size. The ReFLEX microarchitecture consists of simple dual-issue cores with a small out-of-order window to capture ILP from nearby instructions. Consecutive blocks can be mapped to different cores to exploit speculation and far-range ILP. The ReFLEX microarchitecture manages the register file and memory in a distributed fashion, enabling register and

memory locality to be exploited within the cores. We view ReFLEX as a step toward architectures that are amenable to modern compiler optimization algorithms and that can ultimately support dynamic expansion and contraction of threads spanning the cores and their aggregation into processors.

The remainder of this paper first provides an overview of the ReFLEX ISA, execution model, and microarchitecture. It then compares the performance of ReFLEX to that of TFLEX, an architecture with greater complexity. Results show that with a small out-of-order window within a core, ReFLEX can attain comparable performance as the more complicated TFLEX processor.

2 Execution Model and Microarchitecture

2.1 ISA

The two fundamental changes to the ReFLEX ISA compared with a convention RISC ISA are support for block atomic execution and full predication. The ReFLEX ISA is similar to an EDGE ISA [1] except that instead of dataflow execution within a block, execution within a block has tradition RISC style in-order semantics. This avoids the overheads associated with dataflow execution, including fanout instructions, needed when a value is consumed by multiple instructions and nullification instructions, necessary to detect when a block has completed [4]. The results in Section 4 show that ReFLEX executes roughly 15% fewer instructions than TFLEX.

For the current version of this work, a block is defined as having up to 128 instructions with a single entry point and multiple exit points. There is no inherent reason that a ReFLEX block should have a fixed minimum or maximum size. At one extreme, atomic blocks could simply be basic blocks which would require little software support to form. However, larger block better amortize execution overheads. Register storage is partitioned into a global register file and block private data and predicate register files. Values in the private register files are valid only for the duration of a single dynamic instance of a block. Using this block atomic execution model has several benefits:

- **Out of order local commit:** Since the global machine state is only updated when blocks commit and blocks commit in-order, instructions within a block can be retired locally out-of-order. Only block outputs need to be buffered, rather than the speculative state of all instructions as done in a conventional reorder buffer.
- **Register hierarchy:** Since each block has its own set of temporary register names, the need to perform register renaming is reduced compared with a conventional processor. Additionally, the block private values are not stored to the globally shared structures, reducing pressure on these structures.
- **Distributed execution:** The execution overheads of distributed execution are amortized across the instructions in a block allowing for a more scalable system.

2.2 Dataflow Predication

ReFLEX uses predication to create large blocks which better amortize block execution overheads. All of the instructions, except for two constant generation instructions, support predication. Instructions use dataflow predication where an instruction can either be directly predicated or indirectly predicated when one of its operands is predicated [13]. Alternatively, a traditional full predication model could be used [8]. The advantage of dataflow predication is that the compiler is already targeted to TRIPS and TFLEX which use dataflow predication. Additionally, the encoding of dataflow predication is more compact since nested predication can be encoded with a single predicate. The disadvantage of dataflow predication is that more complex state must be tracked to ensure that only the intended instructions are executed. To implement dataflow predication, each block private register is augmented with a valid bit. When a block begins executing, all of the registers are invalid. When a value is written to a register, the valid bit is set. Instructions only execute if their predicates match and the valid bits for all source operands are set.

Since there are only 32 entries in the temporary register file (TRF) and up to 128 instructions in a block, the compiler must perform register allocation and reuse some of the entries in the TRF across multiple live ranges. Special care must be taken to update the valid bits after mispredicated instructions. Each instruction can be augmented with a specifier "C" which signals that the valid bit for the destination register should be cleared if the instruction does not execute. Figure 1 shows an example of how these modifiers are used. In this example, after register allocation R0 and R1 are used at both the top and bottom of the block. The predicated `addi` must be augmented with the "C" specifier because if P0 is false, P1 will not be produced. The predicated `addi` and `subi` will not execute, since the valid bit for P1 will not be set. If the predicated `addi` did not have the "C" specifier, the final `addi` in the block would read the value of R1 from the

```
read R0, G0
read R1, G1
add R2, R0, R1
teq P0, R0, R1
teqi_t<P0> P1, R0, 4
teqi_f<P0> P1, R0, 5
...
addi_t<P1> R62, R0, 1
subi_f<P1>  R62, R0, 1
addi R63, R62, 1
```

Original Code

```
read R0, G0
read R1, G1
add R2, R0, R1
teq P0, R0, R1
teqi_t<P0> P1, R0, 4
teqi_f<P0> P2, R0, 5
...
addi_t<P1> R1, R0, 1
subi_f<P1> R1, R0, 1
addi R0, R1, 1
```

After Register Allocation

```
read  R0, G0
read  R1, G1
add   R2, R0, R1
teq   P0, R0, R1
teqi_t<P0>    P1, R0, 4
teqi_f<P0>    P2, R0, 5
...
addi_t<P1> C R1, R0, 1
subi_f<P1>    R1, R0, 1
addi C R0, R1, 1
```

With Clear Specifiers

Figure 1: Example of dataflow predication, the `addi` instructions must clear their destination register if they are mispredicated to prevent consumers from seeing the values of R1 and R0 from previous live ranges (read instructions).

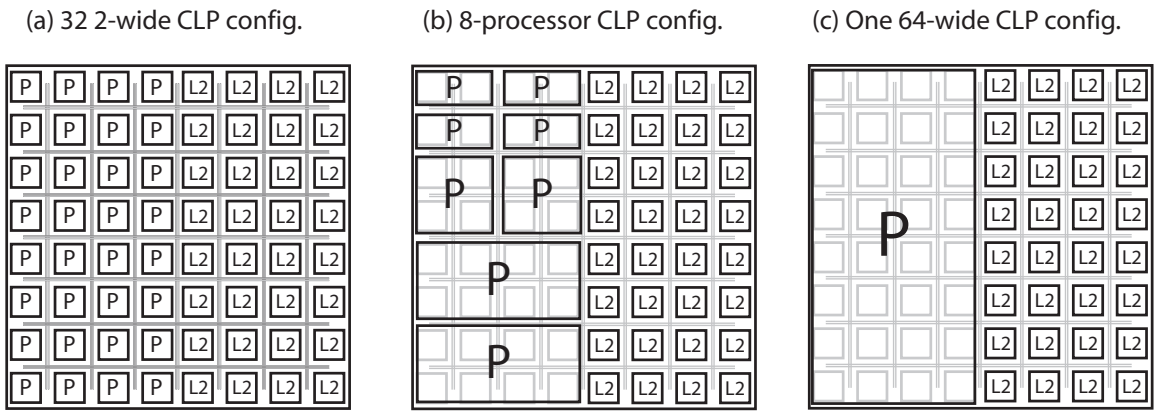


Figure 2: Composability of a 32 core system.

read of G1. The intended behavior is for the `addi` to not execute when P0 is false. The non-predicated `addi` must have the "C" specifier because if it does not execute, later consumers could read the value of R0 produced by the read instruction.

In general, all instructions that define a new live range should have the "C" specifier. The two exceptions to this rule are instructions that define the first use of a register in a block do not need the "C" specifier, since initially all registers are invalid. Second, when a register is defined by multiple predicated instructions, only the first instruction should have the "C" specifier. An example of this case is the lack of a "C" specifier on the predicated `subi` instructions in Figure 1.

2.3 Composability

Figure 2 shows some of the different ways a 32-core system can be configured. This ranges from using each core as a separate processor, allowing for 32 simple processors, aggregating all of the cores into a single processor, or some point in between. Each core has 128 registers; when multiple cores are composed, the program has access to a total of 128 registers which are partitioned across all composed cores. Each core has a 8KB L1 data cache and a 2KB L1 instruction cache. When multiple cores are composed, the caches are combined with address partitioning, giving the program access to 8KB times the number of cores of data cache and 2KB times the number of cores of instruction cache. Each block is assigned a home core which is responsible for performing branch prediction and detecting when a block completes. The full details can be found in the original TFLEX paper [7].

2.4 Microarchitecture

In this work we leverage the microarchitecture of TFLEX [7] with some changes to support the modified execution model. Figure 3 shows an overview of the microarchitecture of a 2 core ReFLEX system; the exact microarchitecture parameters

Parameter	Configuration
Instruction Supply	Partitioned 2KB (per core) I-cache (1-cycle hit) Local/GShare Tournament predictor(2K+256 bits, 3 cycle latency) speculative updates Num. entries: Local 128(L1) + 256(L2) Global: 1024, Choice: 1024, RAS: 16, CTB: 32, BTB: 256, BType: 512.
Execution	In-order / Out-of-order execution, CAM structured variable entry execution window Limited dual-issue (1 FP and 1 INT or 2 INT)
Data Supply	Partitioned 8KB D-cache (per core) (2-cycle hit, 2-way set-associative, 1-read port and 1-write port) 4MB decoupled S-NUCA L2 cache [6] (8-way set-associative, LRU-replacement) L2-hit latency varies from 5 cycles to 27 cycles depending on memory address Average (unloaded) main memory latency is 100 cycles; 64-entry LSQ / core

Table 1: Microarchitecture parameters.

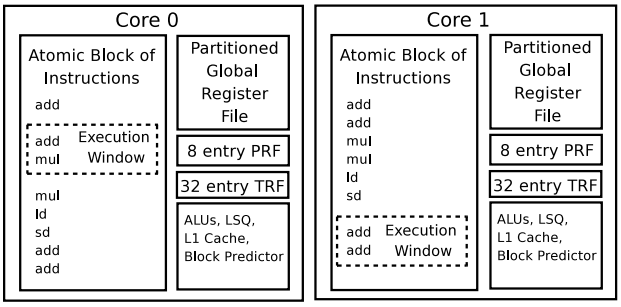


Figure 3: Microarchitecture of a 2 core system.

used for this study are shown in Table 1. To each core, we added a 32-entry temporary register file (TRF), a 8-entry predicate register file (PRF), and a small execution window to allow for out-of-order execution within a block to improve performance. The 128-entry global register file is partitioned across all cores. Dedicated read and write instructions are used to access the global register file. The 128 reservation stations per core used by TFLEX to support dataflow execution within a block are no longer needed. Like TFLEX, multiple blocks can be executed concurrently but they commit their outputs (writes to the global register file and stores to the memory system) in program order. A memory dependence predictor is used to allow memory operations to optimistically execute out of program order.

Global out-of-order execution can be achieved in two ways. First, since multiple blocks are in-flight concurrently, instructions in different blocks are free to execute out of global program order. Second, the instructions within a block are locally executed out-of-order. Figure 3 shows a system configured with an out-of-order execution window of size 2. Section 4 illustrates the performance tradeoff between exploiting near versus far parallelism.

Along with the valid bits needed to implement dataflow predication, the pending reads and writes to a register must also be tracked. These pending operations come from the local out-of-order execution. Pending reads and writes are identified by their instruction number within the block. An instruction may only execute if there are no pending writes to its source registers from earlier instructions and no pending reads to its destination register from earlier instructions. When WAW and WAR hazards are encountered, the processor stalls until they are resolved, unless register renaming is being used.

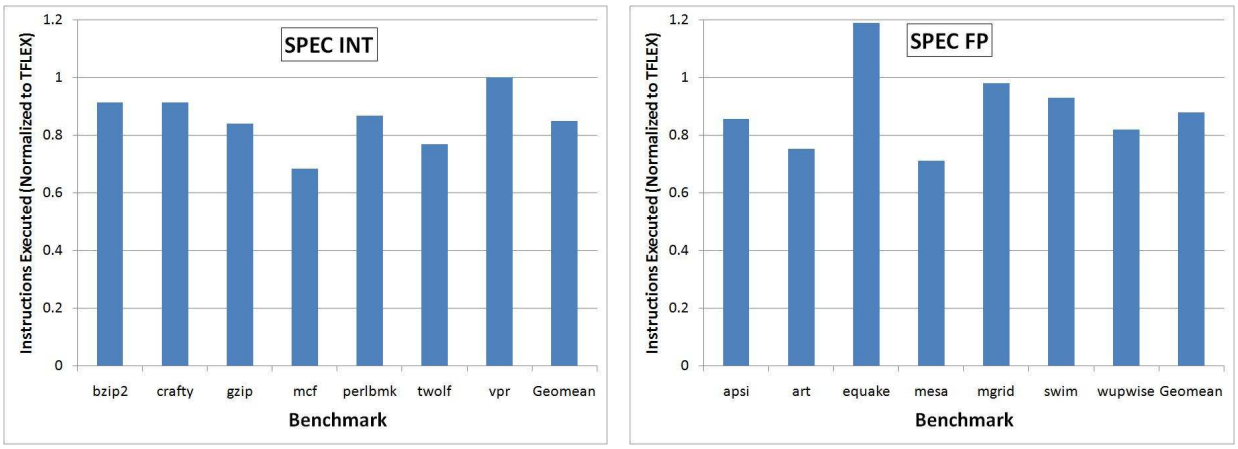
In this version of the work, each block can have up to 32 memory operations. Each core has a 64-entry LSQ which uses a NACK policy to stall memory operations when the LSQ is full [11]. The LSQs are addressed partitioned across the cores. Since the LSQ can reject a memory operation when it is full, each core has a 16-entry memory instruction skid buffer. Once issued, memory instructions are moved from the issue window to the skid buffer. Once an entry has been successfully allocated in the LSQ, the instruction is removed from the skid buffer. Since TFLEX has dedicated storage for all instructions in a block, a memory skid buffer is not needed.

3 Methodology

We use Simpoints [12] from the SPEC2K benchmarks to evaluate ReFLEX. Our current infrastructure supports a total of 14 of the C and Fortran benchmarks. Additionally, we evaluate ReFLEX on 28 of the EEMBC benchmarks [3]. We use the Scale compiler [9] to produce block atomic RISC code. The instructions within a block are scheduled by a modified TRIPS scheduler [2]. All of the simulations are done on a cycle level execution driven simulator.

4 Preliminary Results

In this section we present preliminary results, as we are still tuning the compiler for ReFLEX. The reduction in the number of instructions executed by ReFLEX compared with TFLEX is shown in Figure 4. These results are with an 8-core system with an execution window of 9 instructions without register renaming. The number of instructions executed across all of the different configurations for a



(a) (b)

Figure 4: Number of Instructions Executed Normalized to TFLEX for SPEC INT and SPEC FP

given benchmark varies by less than 1%. Therefore, results for just this one configuration are shown. Fewer instructions are executed by ReFLEX compared with TFLEX on the integer benchmarks than the floating point benchmarks. This is due to the complex control of the integer benchmarks, which requires more dataflow overhead instructions. Some benchmarks, such as vpr and equake see no reduction in instruction count compared with TFLEX. In fact, equake actually executes 19% more instructions on ReFLEX. For this benchmark, the compiler has formed blocks differently than when compiling for TFLEX and the blocks produced are less efficient, requiring more instructions to be executed. Overall, there is a savings of roughly 15% in the number of instructions that must be executed.

Figure 5 shows the geometric mean across all of the benchmarks normalized to a 1-core configuration where the baseline core executes a block in-order. Each line shows results for increasing the amount of out-of-order execution performed locally within a core. An execution window of 128 allows for full out-of-order execution across a single block. With an execution window of 128 instructions, ReFLEX is equivalent to TFLEX except for the dataflow overhead instructions present in TFLEX. When compiling for ReFLEX, the compiler forms different blocks than when compiling for TFLEX since it does not need to allocate space in the block for these dataflow overhead instructions. Generally this results in better performance; however, in some cases the ReFLEX blocks are less efficient. Tuning the compiler for ReFLEX is ongoing work. Results are shown for SPEC INT, SPEC FP, and EEMBC both with and without register renaming on the TRF and PRF. Increases in the size of the local execution window result in performance increases; however, a local window of 17 instructions captures nearly all of the performance of a maximally sized 128 entry execution window. The scalability varies greatly based on the benchmark suite. SPEC INT and EEMBC have limited scalability and performance generally peaks at 8 cores. SPEC FP sees performance increases up to 16 cores. When 32 cores are used, the performance degrades due to increases in communication costs and decreases in branch predictor accuracy at a large speculation depth.

On SPEC INT and EEMBC, where the amount of parallelism exploited is limited, the gains from performing register renaming are limited. For these benchmarks, the energy overhead of performing register renaming is not justified. On SPEC FP, register renaming increases the maximum speedup achieved from 4.5 to just under 5.5. In cases where absolute performance is critical, a large execution window coupled with register renaming provides the best performance. In other cases, energy can be saved by not performing register renaming and simply stalling when WAW and WAR hazards are encountered.

As the number of cores increases, the difference in performance between in-order and out-of-order cores increases. Out-of-order cores can better tolerate the increasing communication delays present in larger configurations. To achieve high performance, some degree of out-of-order execution is needed to exploit near parallelism, especially with a large number of cores. When only in-order cores are used, the performance is limited and 2 out-of-order cores with an execution window size

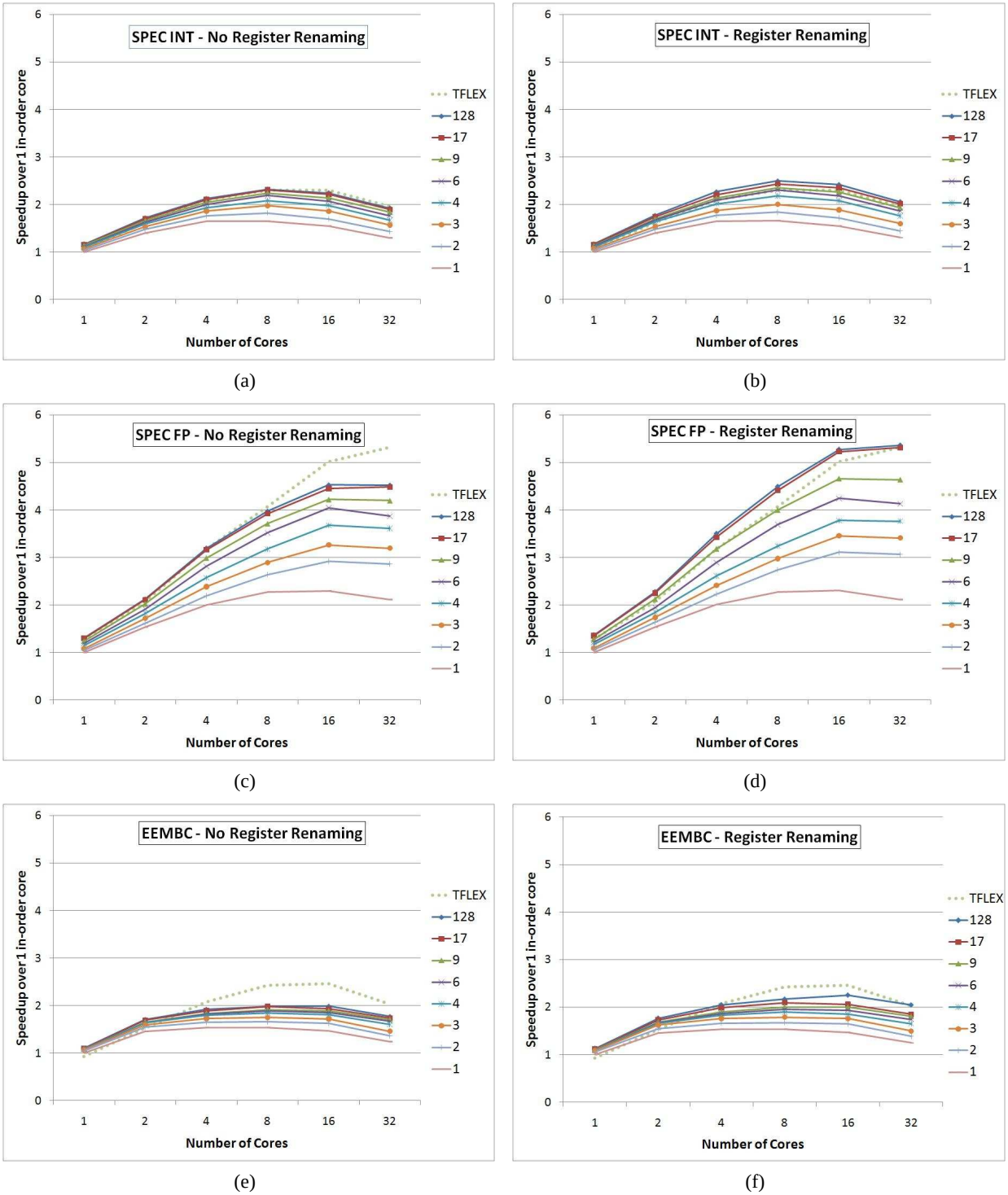


Figure 5: Scalability compared to 1 in-order core across SPEC INT, SPEC FP, and EEMBC both with and without register renaming on the PRF and the TRF

of 17 can match the performance of 16 in-order cores. Performance can be increased either by increasing the size of the local execution window or increasing the number of cores depending on area and power constraints. There is ongoing work to increase the scalability of TFLEX on benchmarks such as EEMBC and SPEC INT. Some of these mechanisms could also be applied to ReFLEX.

Figure 6 compares the performance of ReFLEX to TFLEX [7] across the three sets of benchmarks. TFLEX has an advantage in that it can choose to execute any ready instruction within the block but its performance is limited by the overhead instructions needed to perform dataflow execution within a block. When the local execution window is 9 entries or larger, ReFLEX is competitive with TFLEX. On SPEC FP register renaming is needed to remain competitive with larger number of cores. ReFLEX performs better on SPEC INT and EEMBC compared to TFLEX since the scalability of these benchmarks is limited, thus ReFLEX's limitations have less of an impact on performance.

As the number of cores increases, generally, performance decreases relative to TFLEX. With a large number of cores, the scheduling of instructions within a block has the potential to degrade the performance of not only the current block but also consumers of the current block's outputs. The scheduler must deal with two competing interests. It must place instructions that produce values consumed by other blocks early in the block, to produces these outputs as soon as possible. At the same time, it must place instructions that consume values from other blocks later in the block so that these consumers do not delay the execution of independent instructions while waiting for remote values to be produced. Instructions can be both a consumer of a value from a prior block and a producer of a value needed by a subsequent block. We found that the best scheduling heuristic was to order instructions based on their transitive distance to the top of the dataflow graph, where instructions with large distances being placed later in the block. We look across the likely block predecessors to include the dataflow graphs for prior blocks. This heuristic places instructions that are less likely to stall execution while waiting on predecessors earlier in the block. Those instructions that are more likely to have to wait on predecessors are placed later in the block.

Figure 7 shows the individual benchmark performance compared with TFLEX with a local window size of 9. The EEMBC benchmarks are omitted for space constraints due to the large number of benchmarks in the suite. Performance ranges from 50% worse than TFLEX to 80% better than TFLEX. This wide range is a function of the amount of fanout in each benchmark, which ReFLEX eliminates, and the importance of a large local execution window, where TFLEX has the advantage. Register renaming is particularly important to mcf, providing roughly a 20% performance improvement for larger number of cores.

5 Related Work

Block structured ISAs: Multiscalar processors [14] use software to divide a program into a collection of tasks where each task is similar to an atomic block. Predication is not used to grow the size of tasks but rather internal control flow is allowed. Most tasks have far fewer dynamic instructions than the number of static instructions in a ReFLEX block. Each task is executed by a separate processor and a single register file is keep globally consistent by connecting the processors in a ring and forwarding register values around this ring. This ring structure requires that tasks be mapped in a round robin fashion. Blocks in ReFLEX can be mapped to arbitrary cores which can be exploited for locality advantages. Additionally, the ring structure establishes a static processor granularity where one of the main goals of ReFLEX is to enable composable execution.

Melvin and Patt proposed a block structured ISA to increase the fetch bandwidth [10]. In their work, blocks are the atomic unit of work and instructions within a block execute in dataflow order. Their program structure is similar to an EDGE ISA without support for predication. They use both a global architectural register file and block local storage. Each block can have up to 256 instructions which requires a large amount of local storage. Since they perform dataflow execution within a block they face many of the challenges of TFLEX. Their main goal was to increase performance by exploiting ILP and a flexible microarchitecture that supported composability was not a focus.

Composable processors: There have been several projects that used traditional ISAs to construct composable processors. However, the per-instruction bookkeeping overheads associated with traditional ISAs limit scalability. Federation [15] considers using tightly coupled in-order cores to achieve out-of-order execution. This is similar to how ReFLEX achieves out-of-order execution across blocks. They focus on enabling this small

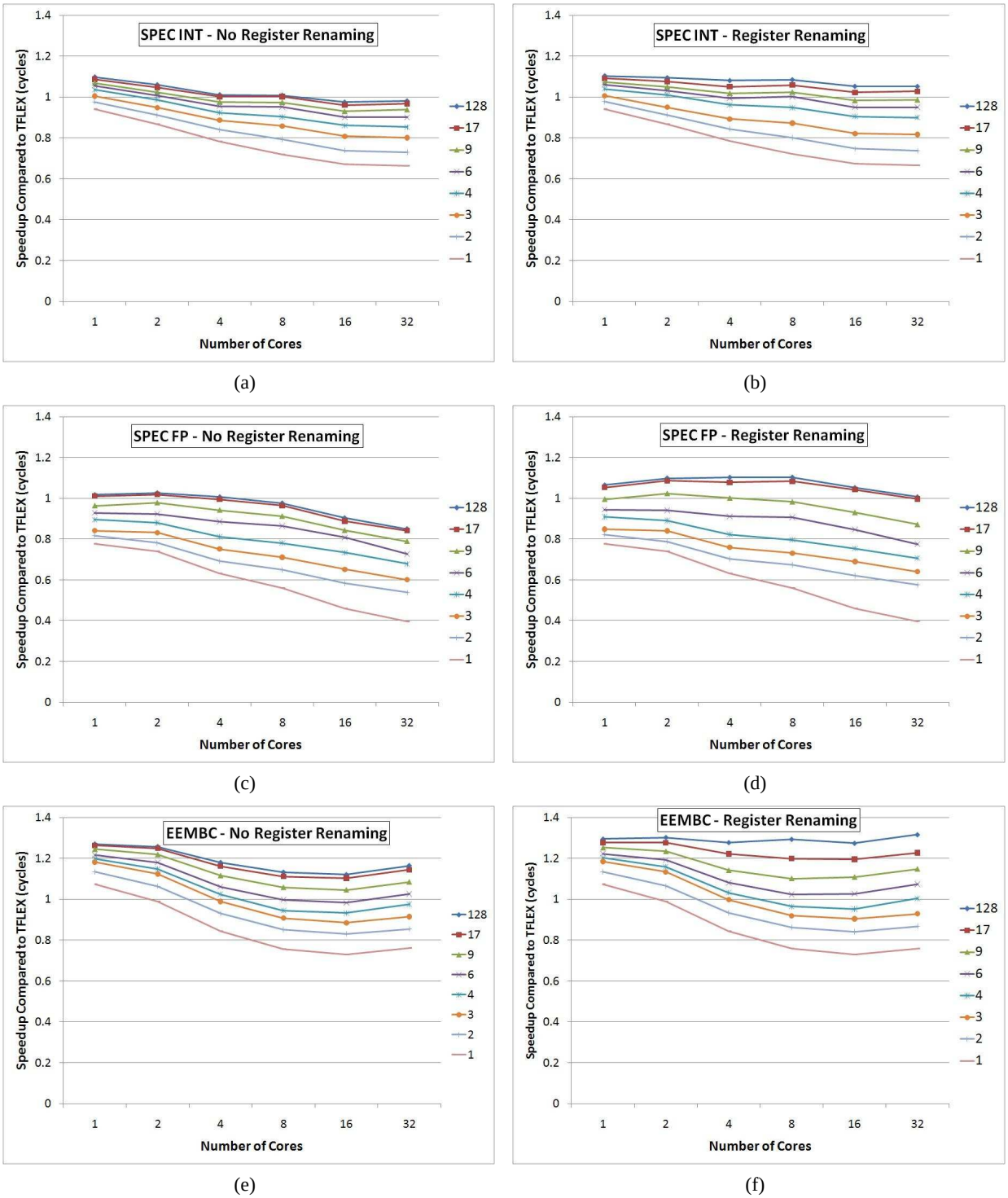


Figure 6: Performance compared to TFLEX across SPEC INT, SPEC FP, and EEMBC both with and without register renaming on the PRF and the TRF

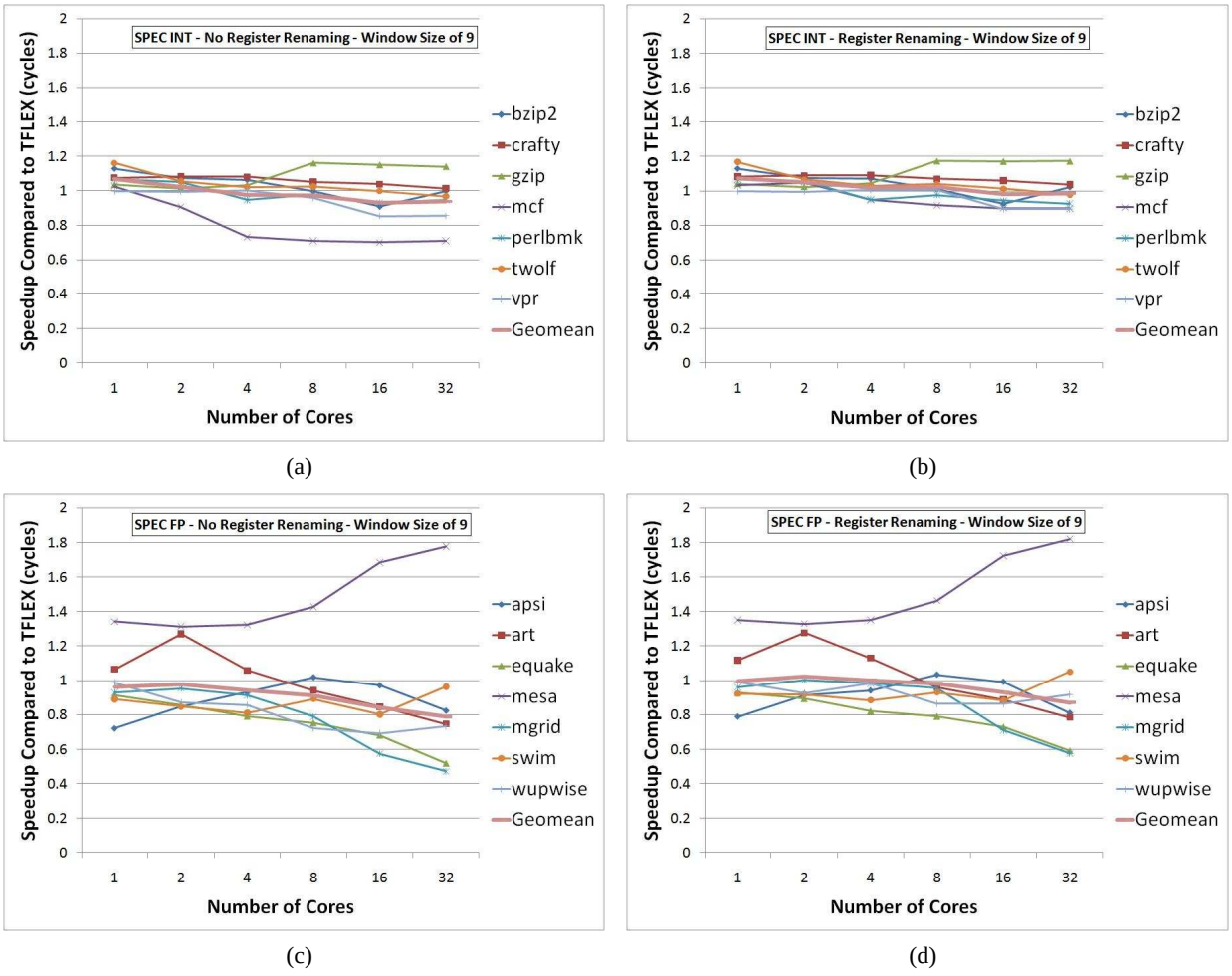


Figure 7: Performance compared to TFLEX for SPEC INT and SPEC FP both with and without register renaming on the PRF and the TRF and a fixed window size of 9 instructions

scale composability to avoid the area and power overheads of out-of-order cores. The focus of Federation is to join 2 or 4 cores while ReFLEX leverages its block atomic ISA to compose a larger number of cores. Federated cores exploit near parallelism while ReFLEX cores exploit far parallelism by executing different portions of the instruction stream.

Core Fusion [5] aims to use composable execution to exploit both ILP and TLP across an 8 core microarchitecture. One key advantage of this work is that it requires no changes to the ISA. When running a single sequential SPEC2K application Core Fusion is 30% and 50% faster than a single two-wide issue out-of-order core on SPEC INT and SPEC FP respectively. Section 4 shows that ReFLEX can achieve better scalability on SPEC2K due to its block structured ISA.

TFLEX [7] leverages an EDGE ISA, which uses block atomic execution and dataflow within a block, to enable composability. However, the dataflow execution within a block has high overheads. Up to 20% of the instructions executed are dataflow overhead instructions which often are on the critical path. Additionally, these dataflow overhead instructions limit the amount of useful work that can be put into a given sized block. Using composability to support dynamic expansion and contraction of parallel work was not studied.

6 Conclusions

In this work, we have shown that a RISC ISA, augmented with block semantics and support for predication, can efficiently execute on a distributed composable microarchitecture. This approach does not rely on register renaming or a reorder buffer to perform out-of-order execution. A ReFLEX processor, with a nine entry local execution window,

is competitive with the dataflow execution TFLEX processor. We are currently conducting a detailed power evaluation. Compared with TFLEX this model reduces the number of instructions executed by roughly 15% and replaces a 128-entry RAM based instruction window with a small (9 entry) CAM based execution window.

We are exploring the potential of internal control flow for reducing the number of fetches for blocks that are part of a tight loop. In order to take advantage of phase changes in the program we are exploring support for lightweight mechanisms to spawn and then contract parallel work when parallel sections are encountered. We believe that the block model is a good mechanism for supporting this requirement because spawning blocks is a lightweight operation. We also plan to study the algorithms that assign blocks to cores. Dependent blocks in serial sections should be scheduled on the same core while independent blocks in parallel sections need to be scheduled on different cores.

This model allows for more aspects of the processor to be dynamically tuned to adjust performance and rapidly adapt to the changing parallelism needs of a program. Further, the lightweight communication provided by this model can potentially exploit fine-grained parallelism that is not profitable to exploit on coarser grained designs. We view this work as a first step and future work will exploit the capabilities of ReFLEX to efficiently execute programs with varying degrees of parallelism.

Acknowledgments

This research is supported by NSF grant CCF-0936700. The authors thank Katherine Coons and Bertrand Maher for their help with the TRIPS tools.

References

[1] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder, and the TRIPS Team. Scaling to the End of Silicon with EDGE Architectures. *IEEE Computer*, 37(7):44–55, July 2004.

[2] K. Coons, X. Chen, S. Kushwaha, D. Burger, and K. McKinley. A Spatial Path Scheduling Algorithm for EDGE Architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 129–140, October 2006.

[3] `http://www.eembc.org`.

[4] M. Gebhart, B. A. Maher, K. E. Coons, J. Diamond, P. Gratz, M. Marino, N. Ranganathan, B. Robatmili, A. Smith, J. Burrill, S. W. Keckler, D. Burger, and K. S. McKinley. An Evaluation of the TRIPS Computer Systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1–12, March 2009.

[5] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez. Core Fusion: Accommodating Software Diversity in Chip Multiprocessors. In *International Symposium on Computer Architecture*, pages 186–197, 2007.

[6] C. Kim, D. Burger, and S. W. Keckler. An Adaptive Non-Uniform Cache Structure for Wire-Dominated On-Chip Caches. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–222, October 2002.

[7] C. Kim, S. Sethumadhavan, M. Govindan, N. Ranganathan, D. Gulati, S. W. Keckler, and D. Burger. Composable Lightweight Processors. In *International Symposium on Microarchitecture*, pages 281–294, December 2007.

[8] S. Mahlke, R. E. Hank, J. E. Mccormick, D. I. August, and W. W. Hwu. A Comparison of Full and Partial Predicated Execution Support for ILP Processors. In *In Proceedings of the 22th International Symposium on Computer Architecture*, pages 138–150, 1995.

[9] K. S. McKinley, J. Burrill, D. Burger, B. Cahoon, J. Gibson, J. E. B. Moss, A. Smith, Z. Wang, and C. Weems. The Scale Compiler. Technical report, University of Massachusetts, University of Texas, 2005.

[10] S. Melvin and Y. Patt. Enhancing Instruction Scheduling With a Block-Structured ISA. *International Journal on Parallel Processing*, 23(3):221–243, June 1995.

[11] S. Sethumadhavan, F. Roesner, J. S. Emer, D. Burger, and S. W. Keckler. Late-binding: Enabling unordered load-store queues. In *International Symposium on Computer Architecture*, pages 347–357, June 2007.

[12] T. Sherwood, E. Perelman, and B. Calder. Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, September 2001.

[13] A. Smith, R. Nagarajan, K. Sankaralingam, R. McDonald, D. Burger, S. W. Keckler, and K. S. McKinley. Dataflow Predication. In *International Symposium on Microarchitecture*, pages 89–102, December 2006.

[14] G. S. Sohi, S. Break, and T. N. Vijaykumar. Multiscalar Processors. In *International Symposium on Computer Architecture*, pages 414–425, 1995.

[15] D. Tarjan, M. Boyer, and K. Skadron. Federation: Repurposing Scalar Cores for Out-of-order Instruction Issue. In *DAC '08: Proceedings of the 45th annual Design Automation Conference*, pages 772–775, 2008.