Computer Sciences Department

Forwardflow: Scalable, RAM-Based Dataflow Execution

Dan Gibson
David Wood

Technical Report #1656

September 2009

Forwardflow: Scalable, RAM-Based Dataflow Execution

Dan Gibson and David Wood

{gibson,david}@cs.wisc.edu
University of Wisconsin-Madison

Abstract

Power (and thermal) limits have forced an industry-wide shift from increasingly complex uniprocessors to multicore chips with 4, 8, and even 16 simpler processor cores. Yet Amdahl's Law suggests that these cores should not be too simple, lest they exacerbate even a parallel application's sequential bottlenecks. Furthermore, running all cores at full speed will soon exceed the chip's power envelope. Ideally, future CMPs should use cores that trade-off power and performance, allowing the system to scale up a core's instruction-level parallelism (ILP) and memory-level parallelism (MLP) to improve sequential performance.

This work presents the Forwardflow microarchitecture, which executes instructions out-of-order using RAM-based structures in lieu of non-scalable CAM- or matrix-based mechanisms. Forwardflow dynamically builds an explicit internal dataflow representation from a conventional ISA, using forward dependence pointers to guide instruction wakeup, selection, and issue. Because all of Forwardflow's major data structures are RAM-based, the instruction window scales large enough to tolerate long memory access times.

1 Introduction

The last several years have witnessed a paradigm shift in the microprocessor industry, from chips holding one increasingly complex out-of-order core to chips holding 4, 8, and even 16 simpler cores [1, 12, 32]. While Moore's law continues to promise more transistors [7], power and thermal concerns have driven the industry to focus on more power-efficient multicore designs [14]. By focussing on thread-level parallelism (TLP), rather than primarily instruction-level parallelism (ILP) within a single thread, microarchitects hope to improve applications' overall power efficiency. Some researchers project that this trend will continue until chips have a thousand cores [3].

But at least two fundamental problems undermine this vision. First, limiting power dissipation to fit the chip's target power envelope means limiting transistor switching (and leakage). Chakraborty, et al. show that the Simultaneously Active Fraction (SAF)—i.e., the fraction of active transistors—decreases with each technology generation [4]. This implies that on future chips (if not current ones), all cores cannot be simultaneously computing at full speed.

Second, Amdahl's Law still applies. Even well-parallelized applications have sequential bottlenecks that limit their parallel speedup (and most applications are not currently parallel at all). A thousand simple cores may maximize performance in an application's parallel section, but simple cores exacerbate the sequential bottleneck by providing limited ILP. Hill and Marty's multicore model [11] leads to the conclusion that "researchers should seek methods of increasing core performance even at high cost." In other words, rather than simply double the number of simple cores when the transistor count doubles, architects should use some of the additional transistors to increase core complexity and thus single-thread performance instead.

Together, these two problems suggest that future CMPs will need *scalable cores*, that is, cores that can trade off power and performance. Heterogeneous CMPs scale cores *statically*, provisioning some cores with more resources and some with less [19]. Composable core designs scale power and performance by *dynamically* merging two or more

cores into a larger core [15, 13]. By appropriately mapping threads and enabling cores, both approaches represent initial steps towards a truly scalable core design.

Scaling core performance means scaling core resources to extract additional ILP, either by statically provisioning cores differently or by dynamically (de-)allocating core resources. And as memory latencies continue to dominate performance, it also means scaling memory-level parallelism (MLP) to service multiple cache misses concurrently [8]. Scaling MLP requires increasing a core's lookahead, the number of instructions it examines for independent memory operations [5]. Conventional core microarchitectures do not scale well because latency, complexity, and power limit their instruction windows and hence their lookahead. Alternative microarchitectures such as runahead microarchitectures achieve significant MLP, but at the expense of sacrificing ILP [5, 21].

Achieving both scalable ILP and MLP requires a novel core microarchitecture. To achieve sufficient lookahead, the core must avoid using a conventional broadcast-based instruction scheduler—where each instruction's completion wakes up all dependent instructions in a single cycle. Broadcast-based instruction schedulers waste power because over 80% of results wakeup zero or one dependent instructions [24, 27]. Instead, scalable core microarchitectures should leverage *explicit successor representations* [23, 24, 34], that explicitly identify dependent instructions and enable directed wakeup.

This work presents the *Forwardflow* core microarchitecture, which *exploits the synergy between explicit successor representation and single-successor dominance* to build a highly-scalable instruction window. As illustrated in Figure 1, Forwardflow represents inter-instruction dependences via forward pointers. Instructions directly identify the first successor; subsequent successors are represented via a linked list distributed throughout the Dataflow Queue (DQ). Forward pointers order instruction wakeup by true data dependence first and program order second, allowing data-independent operations to execute out-of-order. Forwardflow pipelines pointer accesses, yielding no performance loss when issuing the first consumer of a particular value, and delaying subsequent successors (if any) proportional to their position in the value chain. This optimizes the common case of singleton successor instructions, but adds some overhead for high fan-out values. Forwardflow co-locates operand values and scheduling data in the DQ, but splits the DQ into multiple banks to limit wire length and increase execution parallelism using limited ports. Forwardflow naturally scales power and performance depending upon how many DQ banks a specific core provisions, enables, and utilizes.

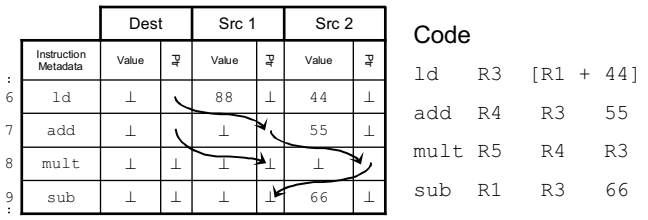


Figure 1. Dataflow Queue example

2 Background and Related Work

Forwardflow is motivated by earlier work on static and dynamic heterogeneous CMPs. Kumar et al. demonstrate that dynamically mapping threads to statically scaled cores can reduce power consumption [17] and improve performance of multiprogrammed workloads [18]. Core Fusion [13] dynamically fuses largely conventional cores, but relies on non-scalable result broadcast between cores. Like Forwardflow, TFlex eliminates broadcast using forward point-

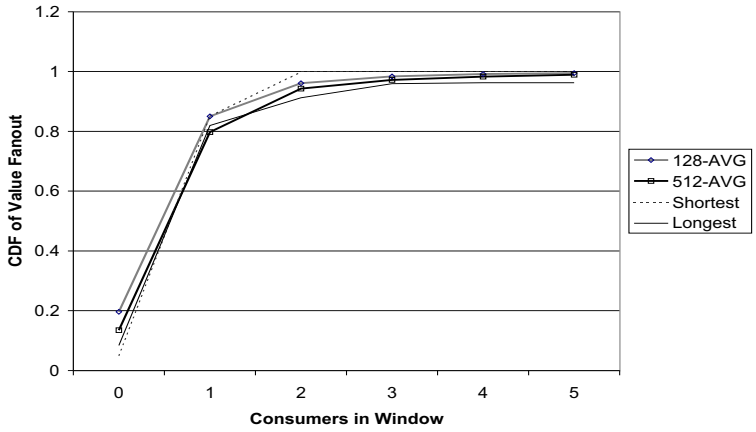


Figure 2. Cumulative Distribution Function of Consumers per Value in 128-entry and 512-entry Windows

ers, but requires a novel ISA and compiler to produce them [15], while Forwardflow generates pointers dynamically from a conventional ISA.

Forwardflow builds on earlier research on exploiting ILP and MLP to improve (processor) core performance [6, 10, 21, 22, 24, 27, 28, 30, 34, 35]. Briefly, these systems examine a predicted future instruction stream—the *instruction window*—to locate independent operations and memory accesses to execute in parallel. The *instruction scheduler* determines when an instruction is ready to execute (wakeup) and when to actually execute it (selection). In general, instruction windows are scalable because they are RAM-based, while most instruction schedulers are not because they rely on CAM-based [35] or matrix-based [10, 27] broadcast for wakeup and priority encoders for selection.

A non-scalable instruction scheduler limits how much ILP a core can exploit, due to a phenomenon called *IQ (scheduler) Clog* [33] where the scheduler fills with instructions dependent on a long-latency operation such as a cache miss. Various optimizations attack this problem by steering dependent instructions into queues [22], moving dependent instructions to a separate buffer [20, 26, 30], and tracking dependences on only one source operand [16]. These proposals ameliorate, but do not eliminate, the poor scalability of traditional instruction schedulers.

Fundamentally, broadcast-based instruction schedulers waste power because most instructions have few successors [24, 27]. Figure 2 plots the CDF of the average number of dependent instructions for 128-entry and 512-entry CAM-based schedulers, averaged over the SPEC CPU 2006 benchmarks. The figure also shows the CDF of the `libquantum` (`leslie3d`) benchmark, which have the greatest (least) average value fanout. Notably, for essentially all workloads, more than 80% of all produced values wakeup zero or one dependent instructions. This single-consumer dominance argues against broadcast-based schedulers for power-constrained systems like future CMPs.

Forwardflow builds upon previous pointer-based scheduling approaches [23, 24, 34]. Forwardflow is most closely related to Direct Wakeup [24], which uses pointers to optimize the single successor case but falls back to a matrix-like approach for instructions with multiple successors. Forwardflow takes a holistic approach and uses pointers to represent all dependences. Furthermore, by replacing the physical register file with the highly-banked DQ, Forwardflow enables highly scalable very-large-window designs.

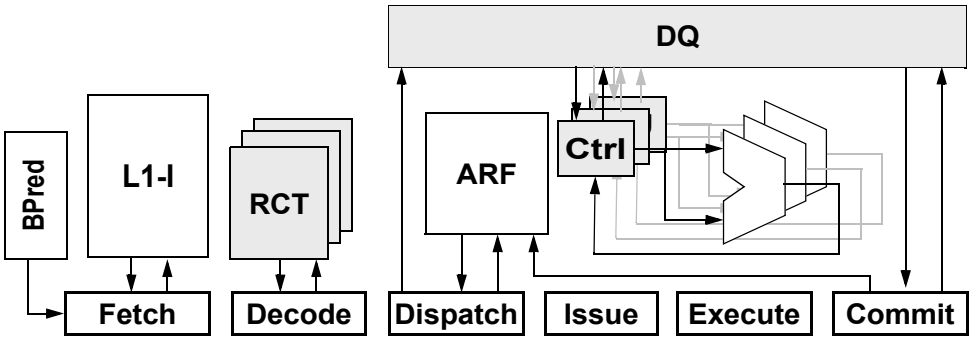


Figure 3. Pipeline diagram of the Forwardflow architecture. Forwardflow-specific structures are shaded.

Other microarchitectures eliminate the scheduler scalability problem by focusing on MLP, at the expense of ILP. Runahead Execution [6, 5, 21] uses a checkpoint/restore mechanism to speculatively commit instructions dependent on a long-latency cache miss and speculatively execute independent instructions, discarding all speculative state when the miss completes. Runahead dramatically increases lookahead, and thus memory-level parallelism, but wastes significant power by re-executing true-path instructions multiple times. In contrast, Forwardflow never re-executes true-path instructions.

Forwardflow also draws inspiration from dataflow architectures, including TRIPS [25], WaveScalar [31], and the MIT Tagged-Token machine [2]. However, Forwardflow focuses on binary compatibility with existing ISAs.

3 Forwardflow Architecture

At the highest level, the Forwardflow pipeline (Figure 3) is not unlike traditional out-of-order microarchitectures. Fetch fetches instructions on a predicted execution path and Decode detects and handles potential data dependences, analogous to traditional renaming. Dispatch inserts instructions into the Dataflow Queue (DQ) and instructions issue when their operands become available. When instructions complete, scheduling logic wakes and selects dependent instructions for execution. Instructions commit in-order from the DQ. To simplify discussion, we describe these operations as single-cycle, but most are pipelined multicycle operations in the simulation model.

3.1 Fetch and Decode

In Forwardflow, Fetch proceeds no differently than other high-performance microarchitectures; Section 4 summarizes the specific design assumptions. Decode produces all information needed for Dispatch, which inserts the instruction into the DQ and updates the forward pointer chains. Decode must determine which pointer chains, if any, each instruction belongs to. It does this using the Register Consumer Table (RCT), which tracks the *tails* of all active pointer chains in the DQ. Indexed by the architectural register name, the RCT looks much like a traditional rename table except that it records the most-recent instruction (and operand slot) to *reference* a given architectural register. Each instruction that writes a register begins a new value chain, but instructions that read registers also update the RCT to maintain the forward pointer chain for subsequent successors. The RCT also identifies registers last written by a committed instruction and thus whose values reside in the Architectural Register File (ARF).

The RCT is implemented as a RAM-based structure. Since the port requirements of the RCT are significant (up to two reads and three writes per decoded instruction per cycle), we expect it to be implemented aggressively and with

some duplication. Fortunately, the RCT is a small structure: each entry consists only $2 \cdot \lceil \log_2 N_{DQEntries} \rceil + 4$ bits, and the number of entries is determined by the number of architectural registers in the target architecture (e.g., an ISA with 72 architected registers requires only 1584 bits of storage with a 512-entry DQ).

3.2 Dispatch

The Dataflow Queue (DQ) is the heart of the Forwardflow architecture, and is involved in instruction dispatch, issue, completion, and commit. The DQ is essentially a CAM-free Register Update Unit [29], in that it schedules and orders instructions, but also maintains operand values. Each DQ entry holds an instruction's metadata (e.g., opcode, ALU control signals, destination architectural register name), three data values, and three forward data pointers, representing up to two source operands and one destination operand per instruction. Value and pointer fields have empty/full and valid bits, respectively, to indicate whether they contain valid information. Dispatching an instruction allocates a DQ entry, but updates the pointer fields of *previously* dispatched instructions. Specifically, *an instruction's DQ insertion will update zero, one, or two pointers belonging to previous instructions in the DQ to establish correct forward dependences.* To reduce port requirements, each DQ field is implemented as an individually addressed RAM.

Figure 4 illustrates the dispatch process for a simple code sequence, highlighting both the common case of a single successor (the `R4` chain) and the uncommon case of multiple successors (the `R3` chain). Fields read are bordered with thick lines; fields written are shaded. The bottom symbol ($\perp$) is used to indicate NULL pointers (i.e., cleared pointer valid bits) and cleared empty/full bits.

In the example, Decode determines that the `ld` instruction is ready to issue at Dispatch because both source operands are available (`R1`'s value, `88`, is available in the ARF, since its busy bit in the RCT is zero, and the immediate operand, `44`, is extracted from the instruction). Decode updates the RCT to indicate that `ld` produces `R3` (but does not add the `ld` to `R1`'s value chain, as `R1` remains available in the ARF). Dispatch reads the ARF to obtain `R1`'s value, writes both operands into the DQ, and issues the `ld` immediately. When the `add` is decoded, it consults the RCT and finds that `R3`'s previous use was at the `ld`'s destination field, and thus Dispatch updates the pointer from `ld`'s destination to the `add`'s first source operand. Like the `ld`, the `add`'s immediate operand (`55`) is written into the DQ at dispatch. Dispatching the `add` also reads the `ld`'s result empty/full bit. Had the `ld`'s value been present in the DQ, the dispatch of the `add` would stall while reading the value array.

The `mult`'s decode consults the RCT, and discovers that both operands, `R3` and `R4`, are not yet available and were last referenced by the `add`'s source 1 operand and the `add`'s destination operand, respectively. Dispatch of the `mult` therefore checks for available results in both the `add`'s source 1 value array and destination value array, and appends the `mult` to `R3`'s and `R4`'s pointer chains. Finally, like the `add`, the `sub` appends itself to the `R3` pointer chain, and writes its dispatch-time ready operand (`66`) into the DQ.

Values for instruction operands may be obtained in four ways, each of which are handled differently in Forwardflow:

- Immediate operands, extracted from the instruction itself, are written into the instruction's appropriate operand value array in Dispatch (e.g., the `add`'s second operand, `55`).
- The Architectural Register File (ARF) is read in Dispatch to provide committed values to dispatching instructions (e.g, the `ld`'s first source operand, `R1`). Values from the ARF are written into the instruction's operand value array, to ensure that values are local to instructions and can be accessed at issue-time without consulting potentially distant structures (e.g., a register file).
- Values produced by earlier in-flight instructions that have not yet executed (i.e., values not available at the consumer's dispatch) will be delivered to the instruction by the pointer chasing hardware (this will be the case for the `add`, `mult`, and `sub` instructions in the example).

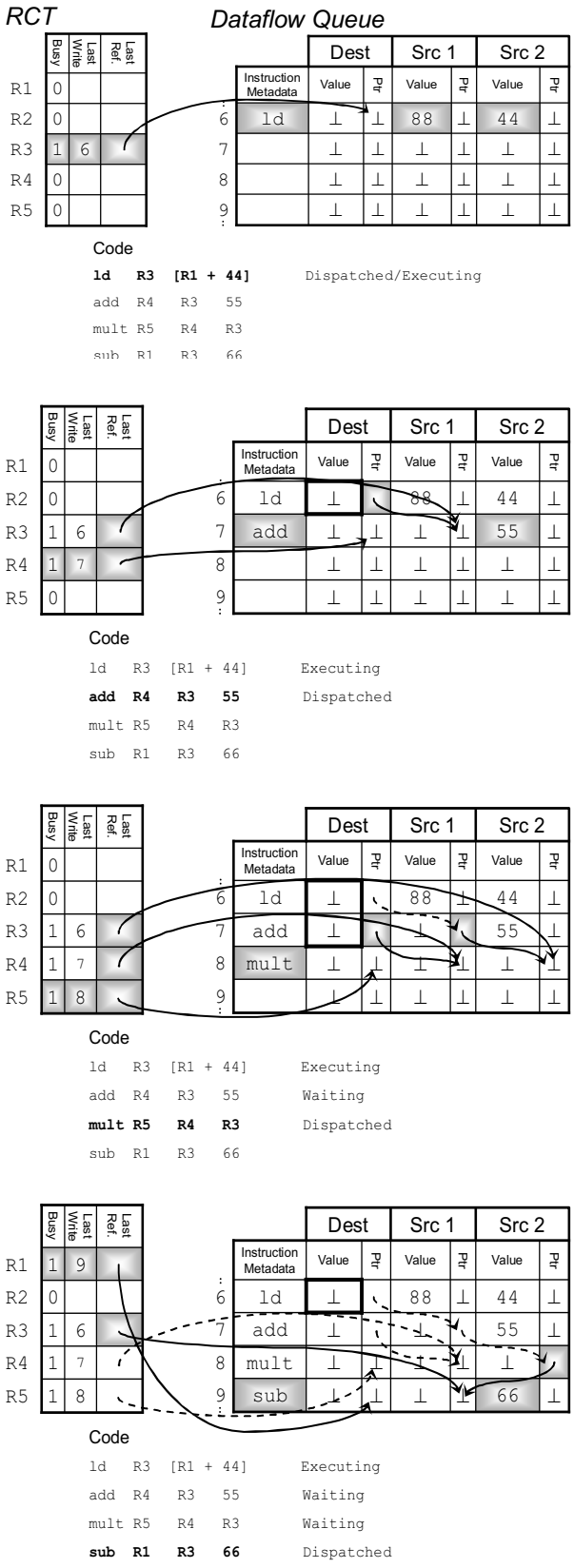


Figure 4. Dispatch Example

- Values from earlier in-flight instructions that have already executed (identified via empty/full bits on value arrays) are read from the previous successor's (or producer's) value array and written into the dispatching instruction's value array (does not appear in the example). Obtaining a value in this manner can cause a single-cycle dispatch stall, depending on DQ bank conflicts.

3.3 Wakeup, Selection, and Issue

Once an instruction has been inserted into the DQ, it waits until its unavailable source operands are delivered by the execution management logic. Each instruction's DQ entry number (i.e., its address in the RAM) accompanies the instruction though the execution pipeline. When an instruction nears completion in its functional pipeline, pointer chasing hardware reads the instruction's destination value pointer. This pointer defines the value chain for the result value, and, in a distributed manner, locations of all successors through transitive pointer chasing. The complete traversal of a chain is a multicycle operation, and successors beyond the first will wakeup (and potentially issue) with delay linearly proportional to their position in the chain.

The wakeup process is illustrated in Figure 5. For simplicity, the example assumes chains are followed only after values are available, though for performance, wakeup is initiated in advance of actual instruction completion in a pipelined manner.

Upon completion of the `ld`, the memory value (`99`) is written into the DQ, and the `ld`'s destination pointer is followed to the first successor, the `add`. Whenever a pointer is followed to a new DQ entry, available source operands and instruction metadata are read speculatively, anticipating that the arriving value will enable the current instruction to issue (a

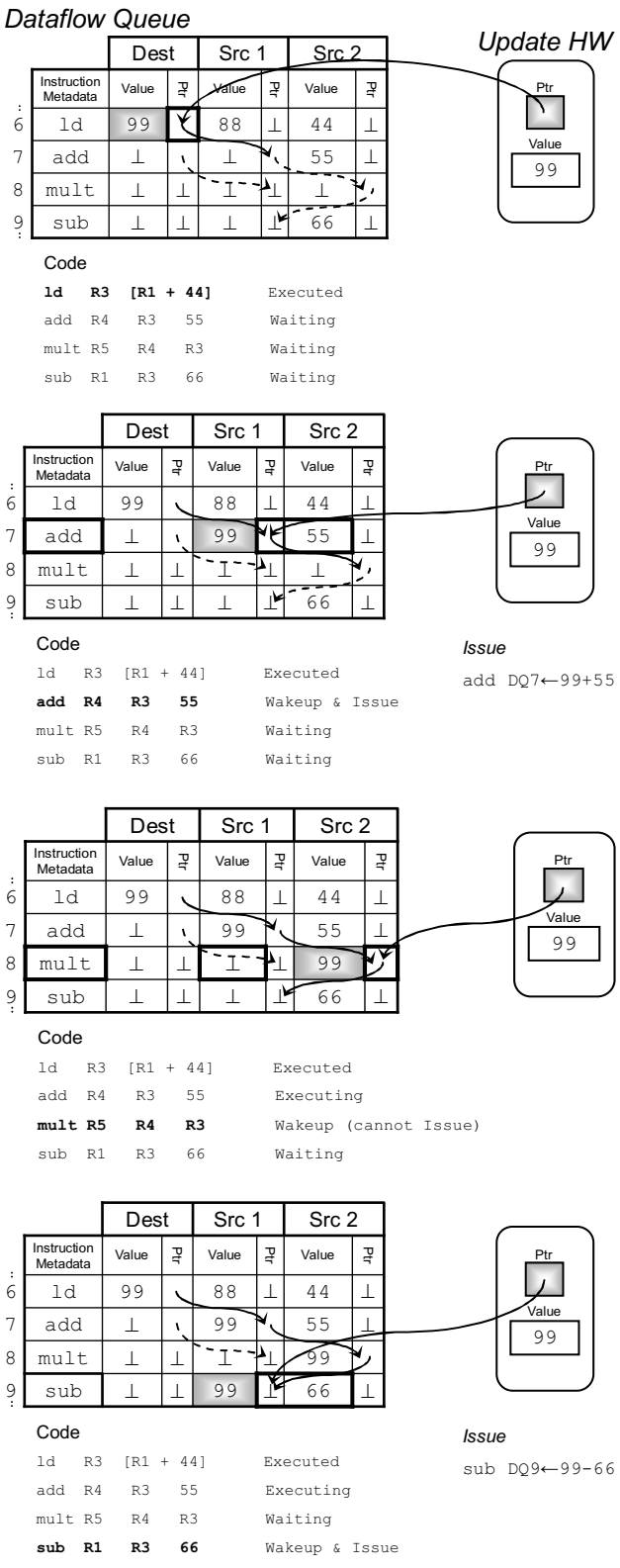


Figure 5. Wakeup Example

common case [16]). Thus, in the next cycle, the `add`'s metadata and source 2 value are read, and, coupled with the arriving value of `99`, the `add` may now be issued. Concurrently, the update hardware reads the `add`'s source 1 pointer, discovering the `mult` as the next successor.

As with the `add`, the `mult`'s metadata, other source operand, and next pointer field are read. In this case, the source 1 operand is unavailable, and the `mult` will issue at a later time (when the `add`'s destination pointer chain is walked). Finally, following the `mult`'s source 2 pointer to the `sub` delivers `99` to the `sub`'s first operand, enabling the `sub` to issue. At this point, a NULL pointer is discovered at the `sub` instruction, indicating the end of the value chain.

In cases where all instruction operands are available at dispatch-time (e.g., the `ld`), the instruction will not be visited by pointer-walking hardware, and wakeup cannot occur as part of the walk of an earlier producer's wakeup chain. To handle this case, an alternate issue path exists from the dispatch logic directly to the issue arbiters, bypassing the DQ. Dispatch stalls if issue resources are unavailable. This case is sufficiently rare (a dispatch-time ready instruction coupled with momentary unavailability of functional pipelines) that front-end stalls of this nature do not significantly affect performance. For fairness of evaluation, we augment our baseline CAM-based out-of-order design to also leverage the dispatch-time-ready optimization. Other instructions behind the stalling instruction need not wait, as program order is maintained by the DQ, and operand availability indicates correct dataflow order.

Instructions are removed from the head of the DQ and committed once they have been executed (nominally, when the empty/full bit on the destination operand's value field has been set). Commit logic removes the head instruction from the DQ by updating the queue's head pointer and writes to the Architectural Register File where applicable. If the RCT's last writer field matches the committing DQ entry, the RCT's busy bit is cleared and subsequent successors may read the value directly from the ARF. The commit logic is not on the critical path of instruction execution, and the write to the ARF is not timing critical as long as space is not needed in the DQ for instruction dispatch. A summary of DQ operations (read/write self/other) and accessed fields is given in Table 1. It should be noted

Table 1. Potential DQ Field Access Types by Operation. (Read/Write Self/Other)

Operation	Metadata	Source 1 Value Empty/Full Bit	Source 1 Value	Source 1 Pointer	Source 2 Value Empty/Full Bit	Source 2 Value	Source 2 Pointer	Destination Value Empty/Full Bit	Destination Value	Destination Pointer
Dispatch*	WS	WS	RO/WS	WO	WS	RO/WS	WO	RO	RO	WO
S1Update	RS	WS	WS	RS	RS	RS				
S2Update	RS	RS	RS		WS	WS	RS			
Complete								WS	WS	RS
Commit	RS			WS$^+$			WS$^+$	WS	RS	WS$^+$

*Dispatch may read other entries in the DQ prior to writing available values into the DQ. Most dispatch-time operations are conditional, depending on decoded instruction state.

$^+$Pointer valid bits are cleared at commit. The pointer itself is not accessed. Alternatively, this action may be relegated to the dispatch logic.

that the table designates which access *may* occur, it does not imply that all accesses are performed for each instruction, nor that the accesses are performed atomically with respect to other accesses.

As stated above, the pointer-walking hardware is responsible for issuing instructions to functional units during traversal. Should a particular instruction be unable to issue because of a control hazard (i.e. all functional units are busy), the pointer walk must stall until the instruction can issue normally. Nominally, this condition is only a minor performance overhead. Rarely, a second control hazard can arise when pointer chain that would normally begin its walk requires the use of stalled pointer-walking control circuitry. This forms a circular dependence, as the functional unit cannot accept a new operation (i.e., the current result must first be collected from the functional unit) and the pointer-walking hardware must stall until it can issue the current instruction, resulting in deadlock. The intersection of these two control hazards is rare (one observed occurrence per 33 billion committed instructions), and can be ameliorated by modest buffering. Should deadlock still arise, the circular dependence is easily detected (i.e., all functional units are stalled and the update hardware is stalled), and can be resolved with a pipeline flush.

3.4 Banked Design and Port Requirements

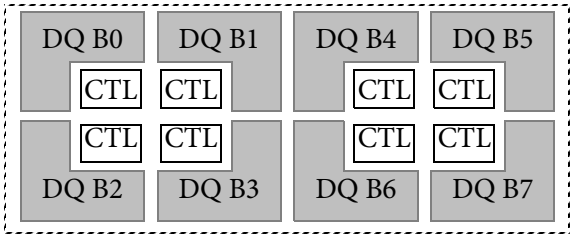


Figure 6. Hypothetical Eight-Bank Hierarchical DQ Floorplan

The number of value chains that may be followed concurrently in a given cycle is bounded by the number of banks (and ports) on the DQ. Pointers that designate operands in a distant bank must traverse a significant chip area. Figure 6 illustrates a Forwardflow floorplan that arranges eight DQ banks in groups of four; pointers that cross bank groups incur additional latency.

Forwardflow's DQ has been designed to deliver adequate performance with only modest port requirements. While performance can be improved by the allocation of additional ports, DQ scalability would be sacrificed. To this end, we have designed Forwardflow to require only two ports per bank of the DQ. One port is dedicated to pointer-chasing hardware; the port type varies depending on which DQ field is referenced (e.g. read-only on pointer arrays, read/write on value arrays). Dispatch and commit logic share a second port on the DQ, and the dispatch logic is given priority access to the port. Contention for the shared port is rare, as the DQ is banked on at least one high-order bit of the DQ entry number (i.e., contention arises only when the DQ is nearly empty or nearly full). The DQ is also banked on low-order bits of the entry number, to provide sufficient bandwidth for multiple dispatches and/or commits per cycle.

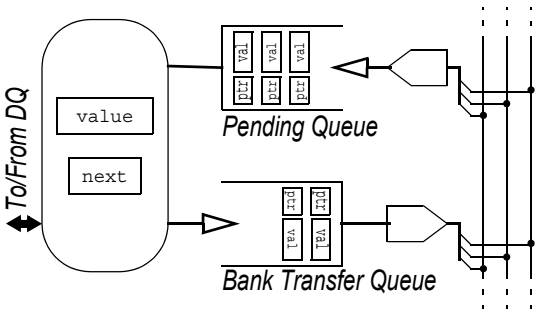
Since the DQ is built entirely of banked RAMs, it can scale to much larger sizes than a traditional CAM-based instruction scheduler. Each instruction slot in the DQ requires an estimated $200 + 3 \cdot \lceil \log_2 N_{DQEntries} \rceil$ bits of storage (the amount of metadata per instruction varies by architecture), not including optional parity bits (one bit per entry per RAM). Thus, a 512-entry Dataflow Queue requires approximately 14KB of storage (about half the size of typical L1 cache).

3.5 Pointer-Chasing Hardware

In our design, each bank of the DQ is serviced by an independent instance of the pointer-chasing hardware shown in Figure 7, consisting of a ***next*** pointer register, a current ***value*** register, a *pending queue* of pointer/value pairs, and (possibly buffered) ports to the interconnect between the banks of the DQ. The logical behavior is described in the accompanying algorithm. Since DQ entry numbers accompany instructions through functional pipelines, pointers to destination fields can be inferred as instructions complete execution.

During a given cycle, the update hardware for a particular bank will attempt to follow exactly one pointer. If no pointer is available (line 8), the DQ is not accessed by the update hardware, thereby conserving power. Otherwise, if `next` designates a non-destination field (i.e. one of the two source operands), the remaining source operand (if present) and instruction opcode are read from the DQ, and the instruction is passed to issue arbitration (line 15). If arbitration for issue fails, the update hardware stalls on the current `next` pointer and will attempt to issue again on the following cycle.

The update hardware writes the arriving value into the DQ (line 18) and reads the pointer at `next` (line 19), following the list to the next successor. If the pointer designates a DQ entry assigned to a different bank, the pair `<next,value>` is placed in the bank transfer queue (line 23), and will traverse the interconnect in the next cycle.




```
1     // Handle pending queue
2     if next == NULL:
3        next = in.ptr
4        value = in.val
5        in.pop()
6
7     if next == NULL:
8        return // No work to do
9
10    // Try to issue, if possible
11    if type(next) != Dest &&
12       dq[next].otherval.isPresent:
13       val2 = dq[next].otherval
14       opcode = dq[next].meta
15       if !Issue(opcode, val, val2):
16          return // Stall
17
18    dq[next].val = value
19    next = dq[next].ptr
20
21    // Handle DQ bank transfer
22    if bank(next) != bank(this):
23       out.push(next,value)
24       next = NULL
```

Figure 7. Pointer-Chasing Hardware and Algorithm

The pending and bank transfer queues should be provisioned with sufficient space to make interconnect stalls rare. In practice, small queues suffice; we have empirically observed that a five-entry pending queue and a two-entry bank transfer queue are sufficient to tolerate interconnect congestion during busy periods.

The inter-DQ-bank interconnect itself is comprised of a first-level crossbar between neighboring banks (refer to Figure 6) for fast communication between logically adjacent DQ entries. A second-level crossbar connects each bank group, with additional communication delay. For maximum performance, the update hardware optimizes the case where `next` is initially NULL, the pending queue is empty, and a new pointer/value pair arrives from the interconnect. This constitutes the most critical path in the case of a DQ bank transfer (i.e., a pointer chain crosses a bank boundary and arrives at an otherwise unutilized bank).

3.6 Control Speculation

Like other out-of-order machines, Forwardflow relies on dynamic branch and target prediction to improve ILP and increase utilization of key structures. Branch recovery mechanisms must restore the RCT's state as it was before the instructions following the branch were decoded, and invalidate all false-path instructions. The former is accomplished by checkpointing the RCT on predicted branches, a technique identical to the checkpointing of a register rename table. To accomplish the latter, we augment the pointer fields with valid bits, which are checkpointed with RCTs on branch predictions and restored on misprediction events, as in [24]. Neither checkpointing or checkpoint-restore are critical-path operations (a handful of cycles), since branch resolution latency can be effectively overlapped with front-end pipeline latency. The hardware for implementing RCT (and pointer valid bit) checkpoints is similar in nature to the Working/Architectural Register File in the UltraSPARC-III+ [9], though our porting requirements are less and our timing requirements are greatly reduced (e.g. one RCT checkpoint per cycle would be an acceptable upper bound).

4 Conclusions

This paper describes and evaluates the Forwardflow core microarchitecture, a scalable, RAM-based implementation of out-of-order execution leveraging forward data pointers to implement instruction wakeup and selection. Forwardflow replaces CAM-or matrix-based broadcast scheduling logic and the physical register file with an efficient, RAM-based Dataflow Queue (DQ), thereby reducing power consumption. Forwardflow's multi-banked DQ scales gracefully from small to large instruction windows, allowing the system or designer to trade-off power and performance depending upon how many DQ banks a specific core provisions, enables, and uses. Forwardflow's scalability makes it an attractive microarchitecture for future statically or dynamically heterogeneous CMPs.

REFERENCES

[1] AMD Corporation. AMD Introduces the World's Most Advanced x86 Processor, Designed for the Demanding Datacenter. http://www.amd.com/us-en/Corporate/VirtualPressRoom/0,,51_104_543_15008 1197% 68,00.html, Sept. 2007.

[2] K. Arvind and R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Transactions on Computers*, pages 300–318, Mar. 1990.

[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 Technical Report No. UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[4] K. Chakraborty, P. M. Wells, and G. S. Sohi. Computation Spreading: Employing Hardware Migration to Specialize CMP Cores On-the-fly. In *ASPLOS 12*, Oct. 2006.

[5] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture Optimizations for Exploiting Memory-Level Parallelism. In *ISCA 31*, pages 76–87, June 2004.

[6] J. Dundas and T. Mudge. Improving Data Cache Performance by Pre-Executing Instructions Under a Cache Miss. In *Proceedings of the 1997 International Conference on Supercomputing*, pages 68–75, July 1997.

[7] I. T. R. for Semiconductors. ITRS 2006 Update. Semiconductor Industry Association, 2006. http://www.itrs.net/Links/2006Update/2006UpdateFinal.htm.

[8] A. Glew. MLP yes! ILP no! Memory Level Parallelism, or, why I no longer worry about IPC, Oct. 1998.

[9] R. Heald et al. A Third-generation SPARC V9 64-b Microprocessor. *IEEE Journal of Solid-State Circuits*, 35(11):1526–1538, Nov. 2000.

[10] A. Henstrom. US Patent #6,557,095: Scheduling operations using a dependency matrix, Dec. 1999.

[11] M. D. Hill and M. R. Marty. Amdahl's Law in the Multicore Era. *IEEE Computer*, pages 33–38, July 2008.

[12] Intel. First the Tick, Now the Tock: Next Generation IntelÆ Microarchitecture (Nehalem). http://www.intel.com/technology/architecture-silicon/next-gen/whitepaper.pd% f, 2008.

[13] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez. Core Fusion: Accomodating Software Diversity in Chip Multiprocessors. In *ISCA 34*, June 2007.

[14] N. Jouppi. The Future Evolution of High-Performance Microprocessors. MICRO-38 Keynote Address, Nov. 2005.
[15] C. Kim, S. Sethumadhavan, M. S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler. Composable Lightweight Processors. In *MICRO 40*, Dec. 2007.
[16] I. Kim and M. H. Lipasti. Half-price architecture. In *ISCA 30*, pages 28–38, June 2003.
[17] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2003.
[18] R. Kumar, D. Tullsen, P. Ranganathan, N. Jouppi, and K. Farkas. Single-ISA Heterogeneous Multi-core Architectures for Multithreaded Workload Performance. In *ISCA 31*, pages 64–75, June 2004.
[19] R. Kumar, D. M. Tullsen, and N. P. Jouppi. Core architecture optimization for heterogeneous chip multiprocessors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2006.
[20] A. R. Lebeck, T. Li, E. Rotenberg, J. Koppanalil, and J. P. Patwardhan. A Large, Fast Instruction Window for Tolerating Cache Misses. In *ISCA 29*, May 2002.
[21] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead Execution: An Effective Alternative to Large Instruction Windows. *IEEE Micro*, 23(6):20–25, Nov/Dec 2003.
[22] S. Palacharla and J. E. Smith. Complexity-Effective Superscalar Processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 206–218, June 1997.
[23] S. E. Raasch, N. L. Binkert, and S. K. Reinhardt. A scalable instruction queue design using dependence chains. In *ISCA 29*, pages 318–329, May 2002.
[24] M. A. Ramirez, A. Cristal, A. V. Veidenbaum, L. Villa, and M. Valero. Direct Instruction Wakeup for Out-of-Order Processors. In *IWIA '04: Proceedings of the Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA'04)*, pages 2–9, Washington, DC, USA, 2004. IEEE Computer Society.
[25] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. Moore. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 422–433, June 2003.
[26] S. R. Sarangi, W. Liu, J. Torrellas, and Y. Zhou. ReSlice: Selective Re-Execution of Long-Retired Misspeculated Instructions Using Forward Slicing. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, Nov. 2005.
[27] P. Sassone, J. R. II, E. Brekelbaum, G. Loh, and B. Black. Matrix Scheduler Reloaded. In *ISCA 34*, pages 335–346, June 2007.
[28] G. S. Sohi. Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers. *IEEE Transactions on Computers*, pages 349–359, Mar. 1990.
[29] G. S. Sohi and S. Vajapeyam. Instruction Issue Logic for High-Performance Interruptable Pipelined Processors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 27–34, June 1987.
[30] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual Flow Pipelines. In *ASPLOS 11*, Oct. 2004.
[31] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. WaveScalar. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 228–241, Dec. 2003.
[32] M. Tremblay and S. Chaudhry. A Third-Generation 65nm 16-Core 32-Thread Plus 32-Scout-Thread CMT SPARC Processor. In *ISSCC Conference Proceedings*, 2008.
[33] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proceedings of the 23th Annual International Symposium on Computer Architecture*, pages 191–202, May 1996.
[34] R. Vivekanandham, B. Amrutur, and R. Govindarajan. A scalable low power issue queue for large instruction window processors. In *Proc. of the 20th Intnl. Conf. on Supercomputing*, pages 167–176, June 2006.
[35] K. C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–40, Apr. 1996.