Forwardflow: A Scalable Core for Power-Constrained CMPs

Dan Gibson
Computer Sciences Department
University of Wisconsin—Madison
1210 W. Dayton St.
Madison, WI 53706
gibson@cs.wisc.edu

David A. Wood
Computer Sciences Department
University of Wisconsin—Madison
1210 W. Dayton St.
Madison, WI 53706
david@cs.wisc.edu

ABSTRACT

Chip Multiprocessors (CMPs) are now commodity hardware, but commoditization of parallel software remains elusive. In the near term, the current trend of increased core-per-socket count will continue, despite a lack of parallel software to exercise the hardware. Future CMPs must deliver thread-level parallelism when software provides threads to run, but must also continue to deliver performance gains for single threads by exploiting instruction-level parallelism and memory-level parallelism. However, power limitations will prevent conventional cores from exploiting both simultaneously.

This work presents the Forwardflow Architecture, which can scale its execution logic up to run single threads, or down to run multiple threads in a CMP. Forwardflow dynamically builds an explicit internal dataflow representation from a conventional instruction set architecture, using forward dependence pointers to guide instruction wakeup, selection, and issue. Forwardflow's backend is organized into discrete units that can be individually (de-)activated, allowing each core's performance to be scaled by system software at the architectural level.

On single threads, Forwardflow core scaling yields a mean runtime reduction of 21% for a 37% increase in power consumption. For multithreaded workloads, a Forwardflow-based CMP allows system software to select the performance point that best matches available power.

Categories and Subject Descriptors

C.1: [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors), Other Architecture Styles—*Adaptable architectures*

General Terms

Performance, Design

Keywords

Chip Multiprocessor (CMP), Power, Scalable Core

If you were plowing a field, which would you rather use: Two strong oxen or 1024 chickens?

—Attributed to Seymour Cray

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 a handful of simpler cores [13, 32]. While Moore's Law continues to promise more transistors [8], power and thermal concerns have driven the industry to focus on more power-efficient multicore designs. Microarchitects hope to improve applications' overall efficiency by focussing on thread-level parallelism (TLP), rather than instruction-level parallelism (ILP) within a single thread.

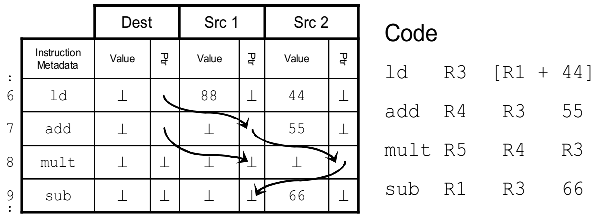
At least two fundamental problems undermine this vision. First, microprocessor vendors are already shipping products in which not all cores can simultaneously operate at full speed due to power constraints [14]. This trend is likely to continue, as the fraction of active transistors decreases with each technology generation [6, 34].

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 sequential bottlenecks 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 budget some of the additional transistors to increase single-thread performance instead.

Together, these two problems motivate *scalable cores*: cores that can trade off power and performance as the situation merits. Scaling core performance means scaling core resources to extract additional ILP, either by statically provisioning cores differently or by dynamically (de)allocating core resources. Conventional core microarchitectures have evolved largely in the uniprocessor domain, and scaling their microarchitectural structures in the CMP domain poses significant complexity and power challenges.

In a scalable core, resource allocation changes over time. Cores must not rely on powered-off components to function correctly when scaled down, and must not wastefully broad-

ISCA'10, June 19–23, 2010, Saint-Malo, France.



		Dest		Src 1		Src 2	
	Instruction Metadata	Value	Ptr	Value	Ptr	Value	Ptr
:							
6	ld	⊥		88	⊥	44	⊥
7	add	⊥		⊥		55	⊥
8	mult	⊥	⊥	⊥	⊥	⊥	
9	sub	⊥	⊥	⊥	⊥	66	⊥
:							

Code

```
ld   R3   [R1 + 44]
add  R4    R3   55
mult R5    R4   R3
sub  R1    R3   66
```

Figure 1. Dataflow Queue Example

cast across large structures when scaled up. Designers of scalable cores should avoid structures that are difficult to scale, like centralized register files and bypassing networks. Instead, they should focus on structures that can be easily disaggregated, and powered-on incrementally to improve core performance independent of other structures.

This work presents the *Forwardflow Architecture*, a scalable core design targeted at power-constrained CMPs leveraging a modular instruction window. Forwardflow represents inter-instruction dependences via a linked list of *forward pointers* [23, 24, 35]. Instructions, values, and data dependences reside in a distributed *Dataflow Queue* (DQ), as illustrated in Figure 1. The DQ is comprised of independent banks and pipelines, which can be activated or de-activated by system software to scale a core's execution resources.

In a Forwardflow-based CMP, single-thread performance can be increased by scaling up a single core (19% runtime reduction on SPEC INT 2006, 25% on SPEC FP 2006, 9% on the Wisconsin Commercial Workload Suite). Other cores can be scaled down (e.g., with DVFS [14]) or disabled to stay within the power budget. Even for multi-threaded workloads, scaling is still valuable when the power consumed does not exhaust the supply. Forwardflow cores can continue to scale performance up until a desired power budget has been reached.

Forwardflow's design delivers both high-performance and energy efficiency. Overall, Forwardflow cores are more efficient than a traditional core baseline in 44 of 47 studied benchmarks. No one configuration is most efficient in all cases, but because Forwardflow cores can scale, they enable system software to optimize a CMP for the desired metric, whether it be performance, energy efficiency, or low chip-wide power consumption.

2 TOWARD SCALABLE CORES

The most important feature of scalable (processor) cores is that they have multiple operating configurations at varied power/performance points. Scalable cores can *scale up*, allowing single-threaded applications to aggressively exploit ILP and MLP to the limits of available power, or can *scale down* to exploit TLP with more modest (and less power-hungry) single-thread performance. To compete with traditional designs, a scalable core should have a nominal operating point at which it delivers performance comparable to a traditional out-of-order core at comparable power, and should offer more aggressive configurations when scaled up. In other words, performance itself should not be sacrificed for performance scaling.

Canonical work in this domain, *Core Fusion* [15] and *Composable Lightweight Processors* [16], compose entire cores to scale all pipeline resources at the same rate. Our work differs by observing that many workloads do not effectively utilize even relatively narrow instruction fetch (e.g., four instructions per cycle). To do so, Little's Law suggests that a core must maintain enough instructions in flight to match the product of fetch width and the average time between dispatch and commit (or squash). These buffered instructions constitute an *instruction window*—the predicted future execution path. As memory latencies increase, cores require large windows to exploit even modest fetch bandwidth. Our work builds on this insight by focusing on scaling window size to expose parallelism in memory-intensive workloads.

However, not all instructions in the window are alike. In a typical out-of-order design, instructions not yet eligible for execution reside in an *instruction scheduler*. The scheduler determines when an instruction is ready to execute (wakeup) and when to actually execute it (selection), based on operand availability. In general, instruction *windows* are easily scalable because they are SRAM-based, while many instruction *schedulers* are not because they rely on CAM-based [36] or matrix-based [10, 27] broadcast for wakeup and priority encoders for selection.

Because many workloads are limited by latency to memory, it is important for high-performance cores—and scalable cores when scaled up—to service as many memory accesses concurrently as possible. However, a non-scalable instruction scheduler limits how much MLP a core can exploit, due to a phenomenon called *IQ (scheduler) clog* [33], in which the scheduler fills with instructions dependent on a long-latency operation (such as a cache miss). Optimizations exist to attack this problem, e.g., by steering dependent instructions into queues [22], moving dependent instructions to a separate buffer [26], and tracking dependences on only one source operand [17]. These proposals ameliorate, but do not eliminate, the poor scalability of traditional instruction schedulers.

Runahead Execution, Waiting Instruction Buffers, and Continual Flow Pipelines address the scheduler problem by draining a scheduler of non-ready instructions (e.g., by buffering instructions [18,31] or by simply discarding them [7, 21]). Drained instructions are no longer eligible for scheduling, and therefore they cannot wake and issue until they are re-inserted into the scheduler.

Another approach to improving scheduler scalability is to name the first successor of a value explicitly with a pointer [23, 24, 35], thereby reducing the likelihood of broadcasts in the scheduling hardware—broadcasts become corner cases but are still necessary. These approaches reduce scheduler complexity and power in fixed-size cores because most instructions have few successors [24, 27]. Our core design exploits this observation ubiquitously, by using

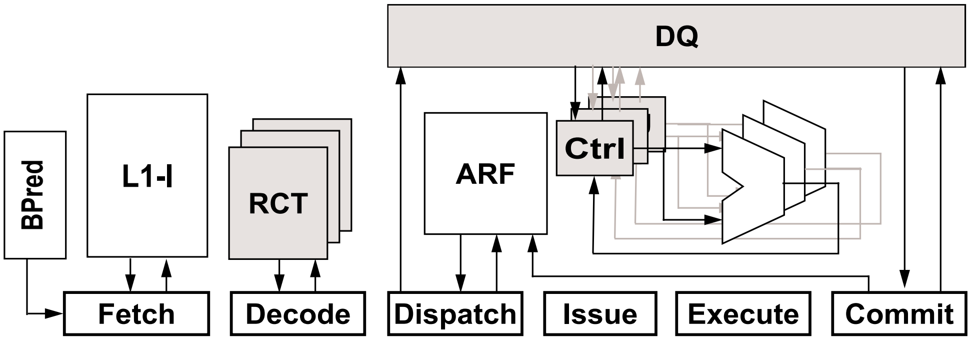


Figure 2. Pipeline diagram of the Forwardflow architecture. Forwardflow-specific structures are shaded.

pointers to represent not only the first register dependency, but *all* register and memory dependences (via NoSQ [28]), thereby *completely eliminating broadcasts*. Like the dataflow architectures that inspired this work [3,25], our design leverages these same dependence pointers to direct data flow within the core itself, rather than relying on a centralized physical register file.

3 FORWARDFLOW ARCHITECTURE

In Forwardflow cores, inter-instruction dependences are represented as linked lists of *forward pointers* [23, 24, 35], instead of using physical register identifiers to label values. These pointers, along with values for each operand, are stored in the *Dataflow Queue* (DQ), shown in Figure 1, which takes the place of the traditional scheduler and centralized physical register file. Instead of broadcasting, DQ update hardware chases pointers to implement instruction wakeup. Though most dependence lists are short, serializing wakeup causes some slowdown. However, the use of pointers throughout the design enables a large, multi-banked DQ implementation, which allows independent lists to be chased concurrently.

At the highest level, the Forwardflow pipeline (Figure 2) is not unlike traditional out-of-order microarchitectures. The Fetch stage 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.

3.1 Frontend

In Forwardflow, Fetch proceeds no differently than other high-performance microarchitectures. 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 resembles 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 which values can be read at dispatch-time from the Architectural Register File (ARF).

The RCT is implemented as a RAM-based table. Since the port requirements of the RCT are significant, we expect it to be implemented aggressively and with some duplication. Fortunately, the RCT itself is small: each entry requires only $2 \cdot \lceil \log_2 N_{DQEntries} \rceil + 4$ bits.

3.2 The Dataflow Queue (DQ)

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 [30], in that it schedules and orders instructions, but also maintains operand values. Each entry in the DQ holds an instruction's metadata (e.g., opcode, ALU control signals, destination architectural register name), three data values, and three forward 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 *earlier* instructions in the DQ to establish correct forward dependences.

Figure 3 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 (⊥) 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 as 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.

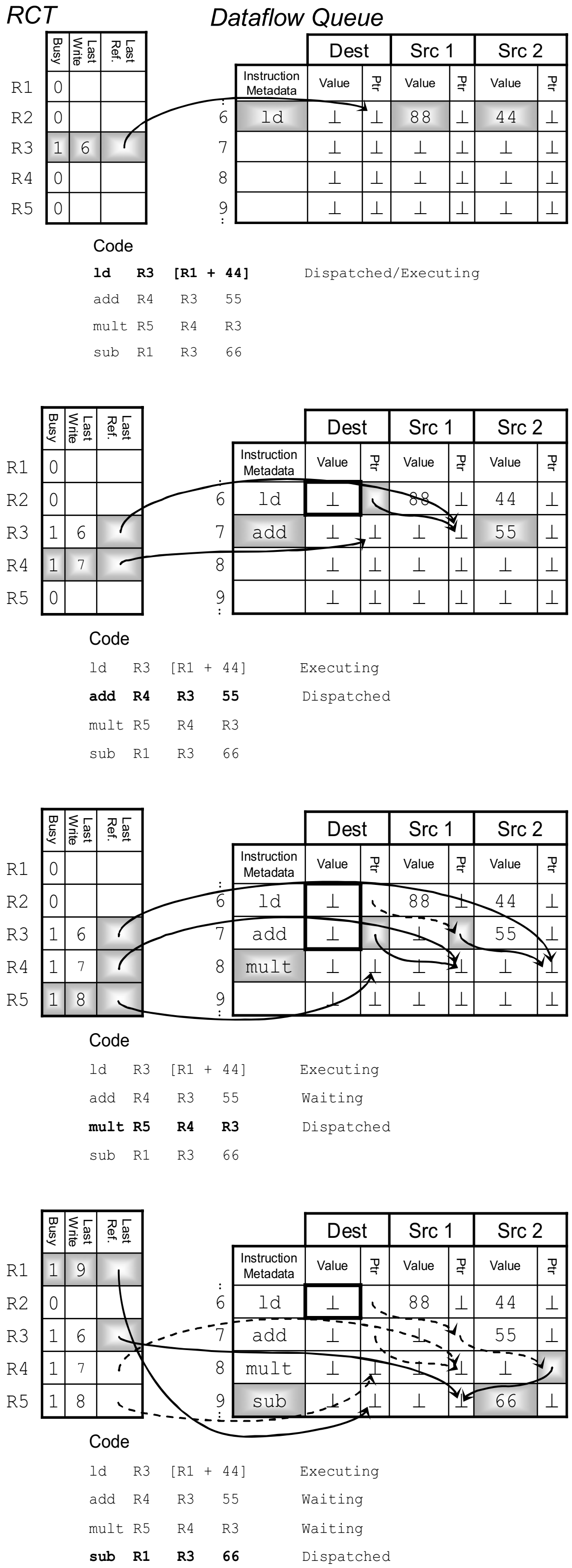
The mult's decode consults the RCT, and discovers that both operands, R3 and R4, are not yet available and were last referenced by the add's source 1 operand and the add's destination operand, respectively. Dispatch of the mult therefore checks for available results in both the add's source 1 value array 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 centralized register file).
- Values produced by earlier in-flight instructions that have not yet executed (i.e., values not available at the successor's dispatch) will be delivered to the instruction by the pointer chasing hardware (e.g., this will be the case for the add, mult, and sub instructions).
- 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).

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 succes-



```
ld   R3  [R1 + 44]      Dispatched/Executing
add  R4   R3   55
mult R5   R4   R3
sub  R1   R3   66
```

```
ld   R3  [R1 + 44]      Executing
add  R4   R3   55       Dispatched
mult R5   R4   R3
sub  R1   R3   66
```

```
ld   R3  [R1 + 44]      Executing
add  R4   R3   55       Waiting
mult R5   R4   R3       Dispatched
sub  R1   R3   66
```

```
ld   R3  [R1 + 44]      Executing
add  R4   R3   55       Waiting
mult R5   R4   R3       Waiting
sub  R1   R3   66       Dispatched
```

Figure 3. Dispatch Example

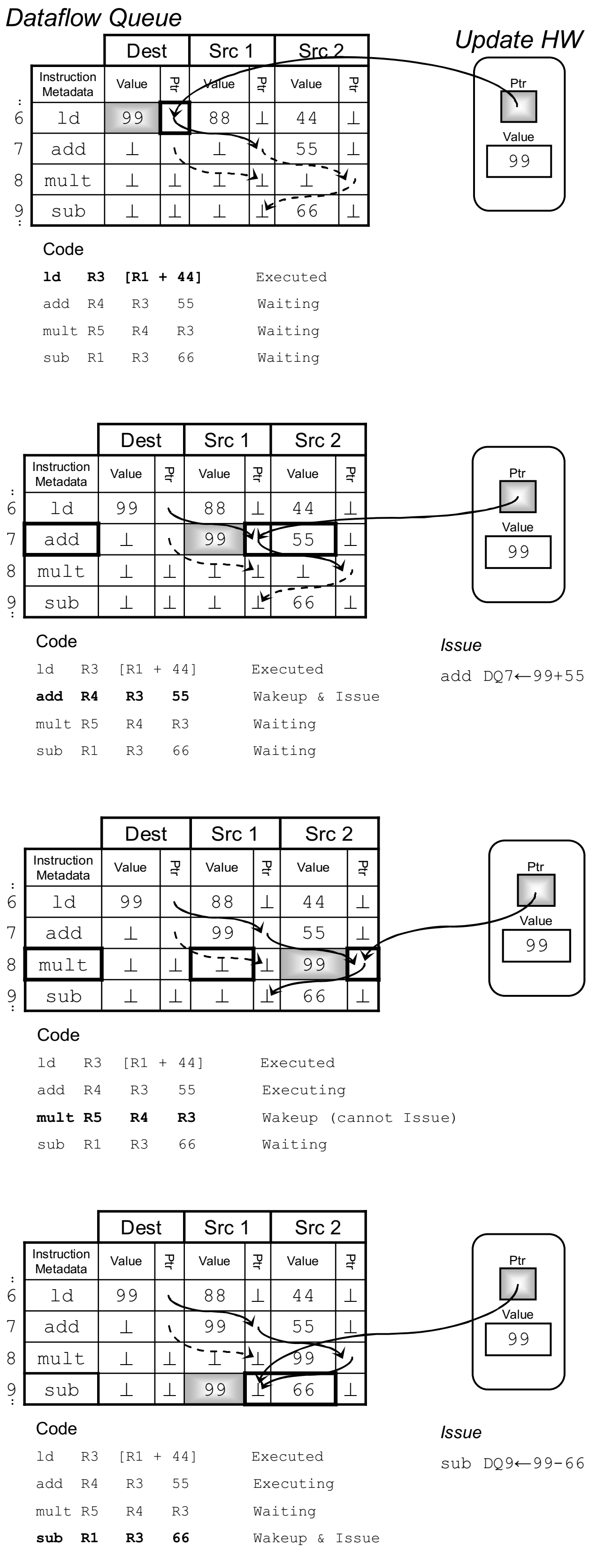
sors through transitive pointer chasing. The complete traversal of a chain is a multicycle operation, and successors beyond the first will wakeup (and potentially issue) with delay linearly proportional to their position in the chain.

The wakeup process is illustrated in Figure 4. 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 common case [17]). 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 chased). 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.

After instructions have been executed (i.e., when the empty/full bit on the destination operand's field has been set), instructions are removed from the head of the DQ and committed in program order. Commit logic removes the head instruction from the DQ by updating the DQ's head pointer and writes to the ARF 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.

As stated above, the pointer chasing hardware is responsible for issuing instructions to functional units during traversal. Should a particular instruction be unable to issue because of a structural hazard (i.e., all functional units are busy), the pointer chase must stall until the instruction can issue normally. Nominally, this condition is only a minor performance overhead. Rarely, a second structural hazard can arise when pointer chain that would normally begin its chase requires the use of stalled pointer-chasing 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-chasing hardware must stall until it can issue the current instruction, resulting in deadlock. The intersection of these two control hazards is rare, 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; ordering



	Instruction Metadata	Dest Value	Ptr	Src 1 Value	Ptr	Src 2 Value	Ptr
6	ld	99		88	⊥	44	⊥
7	add	⊥		⊥		55	⊥
8	mult	⊥	⊥	⊥	⊥	⊥	
9	sub	⊥	⊥	⊥	⊥	66	⊥

Code

```
ld   R3  [R1 + 44]      Executed
add  R4   R3   55       Waiting
mult R5   R4   R3       Waiting
sub  R1   R3   66       Waiting
```

	Instruction Metadata	Dest Value	Ptr	Src 1 Value	Ptr	Src 2 Value	Ptr
6	ld	99		88	⊥	44	⊥
7	add	⊥		99		55	⊥
8	mult	⊥	⊥	⊥	⊥	⊥	
9	sub	⊥	⊥	⊥	⊥	66	⊥

Code

```
ld   R3  [R1 + 44]      Executed
add  R4   R3   55       Wakeup & Issue
mult R5   R4   R3       Waiting
sub  R1   R3   66       Waiting
```

Issue

add DQ7←99+55

	Instruction Metadata	Dest Value	Ptr	Src 1 Value	Ptr	Src 2 Value	Ptr
6	ld	99		88	⊥	44	⊥
7	add	⊥		99		55	⊥
8	mult	⊥	⊥	⊥	⊥	99	
9	sub	⊥	⊥	⊥	⊥	66	⊥

Code

```
ld   R3  [R1 + 44]      Executed
add  R4   R3   55       Executing
mult R5   R4   R3       Wakeup (cannot Issue)
sub  R1   R3   66       Waiting
```

	Instruction Metadata	Dest Value	Ptr	Src 1 Value	Ptr	Src 2 Value	Ptr
6	ld	99		88	⊥	44	⊥
7	add	⊥		99		55	⊥
8	mult	⊥	⊥	⊥	⊥	99	
9	sub	⊥	⊥	99	⊥	66	⊥

Code

```
ld   R3  [R1 + 44]      Executed
add  R4   R3   55       Executing
mult R5   R4   R3       Waiting
sub  R1   R3   66       Wakeup & Issue
```

Issue

sub DQ9←99-66

Figure 4. Wakeup Example

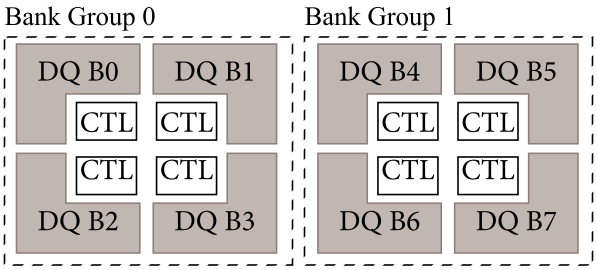


Figure 5. Eight-Bank Hierarchical DQ Floorplan

properties of functional units guarantees forward progress of at least one instruction.

3.4 Distributed Implementation

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 5 illustrates a conceptual Forwardflow floorplan that arranges eight DQ banks in groups of four; pointers that cross bank groups incur additional latency. Functional pipelines are associated with groups, so execution resources can scale with window size.

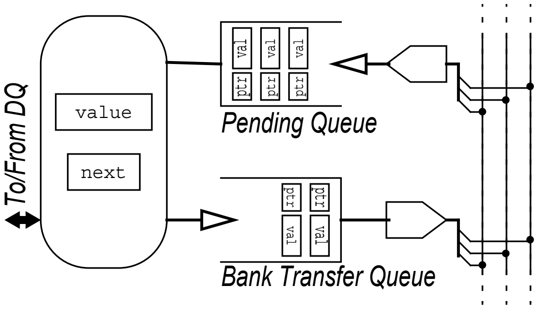
The DQ is sub-banked on low-order bits of the DQ entry number to support concurrent access to contiguous elements. Sub-banking delivers ample bandwidth to dispatch and commit logic—which access the DQ contiguously—without adding additional ports. Each field of the DQ is implemented as a separate SRAM (e.g., value fields are separate from each pointer field, etc.), to further simplify the design, and to enable greater concurrency in the DQ management logic.

Since the DQ is built entirely of small SRAMs, it can scale to much larger sizes than a traditional instruction scheduler, yet is accessed at finer granularity than a ROB. Each entry in the DQ requires an estimated $200 + 3 \cdot \lceil \log_2 N_{DQEntries} \rceil$ bits of storage across all fields.

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 6, consisting of a ***next*** pointer register, a current ***value*** register, a *pending queue* of pointer/value pairs, and buffered ports to the interconnect between the banks of the DQ. The logical behavior is described in the accompanying algorithm, which runs every cycle. 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



```
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 6. Pointer Chasing Hardware and Algorithm

instruction is passed to issue arbitration (line 15). If arbitration for issue fails, the update hardware stalls on the current `next` pointer and will 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.

The inter-DQ-bank interconnect itself is comprised of a first-level crossbar between neighboring banks (refer to Figure 5) 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.

3.6 Control Speculation

Like other out-of-order machines, Forwardflow relies on dynamic branch and target prediction to improve pipeline

Table 1. Configuration Parameters

Component	***OoO***	***F-1***	***F-2***	***F-4***
Window Size	128	128 (One Bank Group)	256 (Two Bank Groups)	512 (Four Bank Groups)
Scheduler Type	Hybrid [12]	Forwardflow		
Scheduler Size	Unified 32-entry	Full Window		
Functional Units	2xI-ALU/2xFP-ALU/ 2xD-MEM	2xI-ALU/2xFP-ALU/ 2xD-MEM	4xI-ALU/4xFP-ALU/ 2xD-MEM	8xI-ALU/8xFP-ALU/ 2xD-MEM
Branch Prediction	YAGS 4K PHT 2K Exception Table, 2KB BTB, 16-entry RAS			
Disambiguation	NoSQ [28] 1024-entry predictor, 1024-entry double-buffered SSBF			
Fetch-Dispatch Time	Min. 7 Cycles			
L1-I Cache	32KB, 4-way, 64B line, 4-cycle pipelined, 2 lines per cycle, 2 processor-side ports			
L1-D Cache	32KB, 4-way, 64B line, 4-cycle LTU, write-through, write-invalidate, included by L2			
L2 Cache	1 MB, 8-way, 4 banks, 64B line, 11 (12) cycle load (store) latency, write back, private			
L3 Cache	8 MB, 16-way, 8 banks, 64B line, 24 cycle latency, shared			
Main Memory	2 QPI-like Links (Up to 64GB/s), 300 cycle latency			
Coherence	MOESI-based Directory Protocol			
On-Chip Interconnect	2D Mesh, 16B bidirectional links, one transfer per cycle, 1-cycle 5-ary routers, 5 virtual channels per link			

utilization. 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 [24].

Not all forms of recovery can be handled by restoring from checkpoints. When exceptions or interrupts occur, it is legal to flush all instructions that follow the excepting instruction and quiesce the pipeline. This allows decode to resume with an empty RCT. So long as recoveries of this type are rare cases, performance impact is limited.

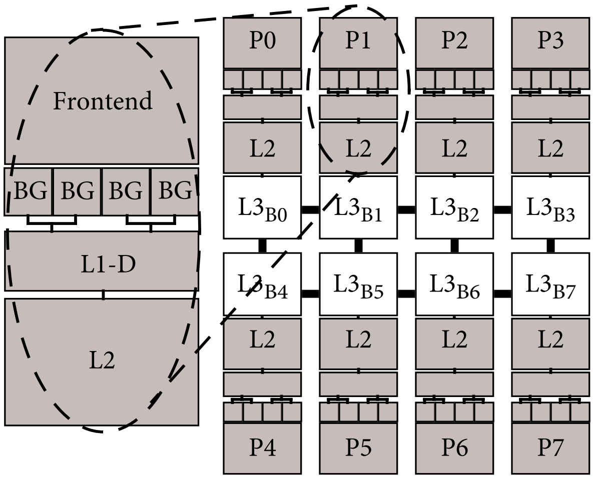


Figure 7. 8-Way CMP Target

3.7 Scaling Forwardflow Cores

The use of pointers throughout the design enables Forwardflow's control logic to handle window size reconfiguration gracefully—pointers are already oblivious of the structure to which they *point*. Since the DQ is managed as a FIFO, it is a simple matter to modify the head and tail pointer wraparound logic to accommodate variable DQ capacities that are powers of two, using simple modulo-2^N logic. DQ size can be abstracted as a power-control state [14], manageable by system software. Unused bank groups can safely be power-gated. While some prior work has examined techniques for dynamically adapting a core's aggressiveness in hardware [2], for now, we leave this policy decision to software. This work is agnostic of the precise mechanism that software might use to make scaling decisions, but many solutions are possible, including static profiling, online monitoring, and dynamic adaptation.

4 METHODOLOGY

Our target machine is an 8-way CMP, pictured in Figure 7. Each node consists of a core, a private L1/L2 cache hierarchy, and one bank of a large, shared L3. We assume each core and private cache hierarchy operates in its own voltage domain, and furthermore that bank groups in the Forwardflow designs can be independently powered off. We evaluate Forwardflow using full-system cycle-accurate simulation, using Virtutech Simics [19], GEMS's Ruby [20], and in-house timing-first processor models. We simulate SPEC CPU 2006 [9], SPEC OMP [4], and Wisconsin commercial workloads [1]. We report ED and ED^2 as our measures of energy efficiency.

We include a traditional out-of-order implementation as our baseline (*OoO*), to show that a nominal Forwardflow core performs comparably to a more traditional architecture. All target machines use NoSQ [28] for memory disambiguation. Our implementation of NoSQ represents memory dependences by inserting artificial register dependences,

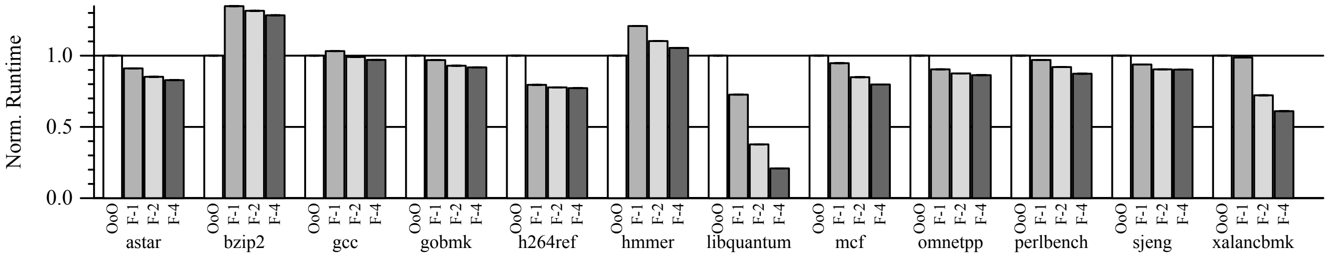


Figure 8. SPEC INT 2006 Runtime

which are enforced by the DQ as though a genuine register dependency existed. NoSQ's predictions are verified via store vulnerability filtering and load replay at commit-time.

We evaluate three Forwardflow configurations with progressively larger DQs: *F-1*, *F-2*, and *F-4*. *F-1* has the same window size and the same number of execution resources as *OoO*. *F-1* has a four-way banked DQ; each bank has 32 entries. With the exception of cache bandwidth and capacity (held constant, as we do not scale the L1-D or D-TLB), *F-2* has twice the execution resources of *F-1*—it represents two *F-1* backends powered-on, similar to Figure 5. *F-4* again doubles backend resources (i.e., four powered-on backends), for an aggregate window size of 512 entries, peak issue rate of 18 (8 INT, 8 FP, 2 MEM).

Our target machines run unmodified SPARCv9 operating systems and binaries. We model hardware-assisted TLB fill and register window exceptions for all target machines. We simulate each benchmark for one hundred million instructions. Multiple runs are used to achieve tight 95% confidence intervals (error bars are not visible in most cases). Benchmarks are fast-forwarded past their initialization phases, during which page tables, TLBs, predictors, and caches are warmed. We have augmented our simulators with Wattch [5] and CACTI 5 [29], which provide architectural-level approximations of power consumed by logic and memory structures in the 32nm process. Our model assumes aggressive clock gating of logic structures not in use, with no reactivation delay. L2 and L3 caches are implemented with variable bias to control leakage and Low Standby Power Devices [29] are used throughout the design, accounting for low overall leakage power.

5 EVALUATION

We first consider the behavior of single threads on single cores of our 8-way CMP. We assume the seven unused cores are in an off state in which they consume no power (this assumption also applies to their private L2 caches, but *not* to their shared L3 banks—single-thread benchmarks observe the full L3 capacity). We believe this mode of operation will not be uncommon in future chips, as commodity multi-threading remains elusive for many workloads.

A scalable core allows core configuration to be customized by system software, so it is possible to have the best of all worlds in most situations. We envision that this capability will complement DVFS in the positive scaling direction (i.e., can be used to scale performance up beyond what frequency scaling alone can provide). However, for due diligence, we evaluate all configurations, with the full expectation that not all points will be favorable for all configurations.

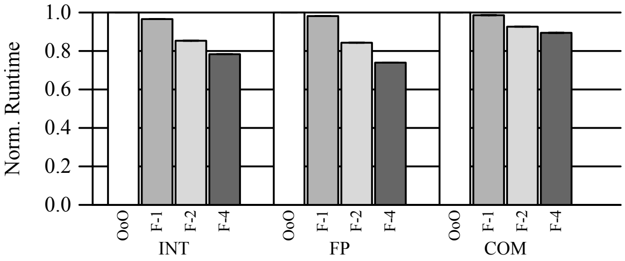


Figure 9. a) INT, FP, and COM Mean Runtime

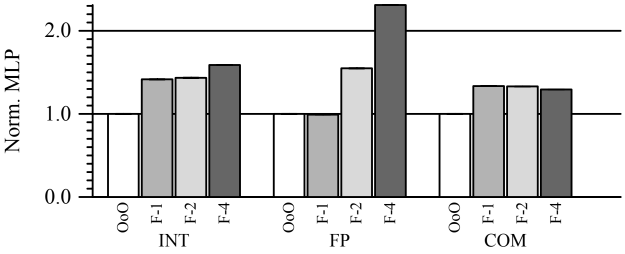


Figure 9. b) INT, FP, and COM Mean MLP

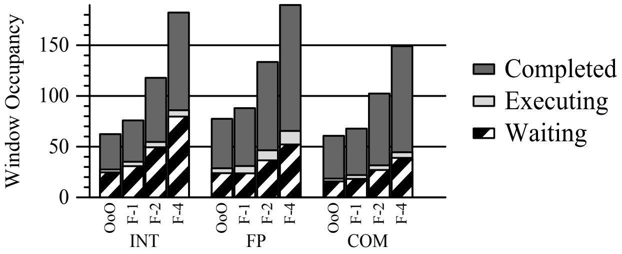


Figure 9. c) INT, FP, and COM Categorized Mean Window Occupancy

5.1 Single-Threaded Performance Results

Figure 8 presents runtimes for individual SPEC INT 2006 benchmarks, normalized to *OoO*. Figure 9.a presents the geometric means for SPEC INT, SPEC FP and single-thread versions of the commercial workloads. Overall, these show the expected result that performance improves as window size increases: the mean runtime of *F-4* is 21% less than *F-1* (23% vs. *OoO*)—indicating that Forwardflow delivers performance scaling for single threaded workloads.

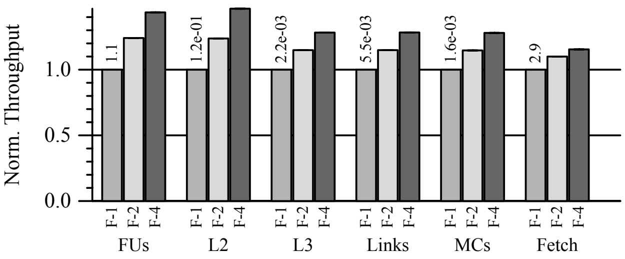


Figure 10. Scaling Effects on Throughput

To gain insight into the behavior of our design, we examine the MLP and categorized window occupancy of each configuration in Figures 9.b and 9.c, respectively. In the latter, we classify post-dispatch instructions into three categories: *Waiting* instructions are not yet eligible to execute because of an unavailable operand (these instructions are scheduler-resident in *OoO*). *Executing* instructions are currently being executed by the functional pipelines, or are outstanding in the memory system. *Completed* instructions have finished execution, but have not yet committed, due to an earlier Waiting or Executing instruction.

MLP and occupancy of Executing instructions (the observed ILP) are of key importance to performance. Across all single-thread benchmarks, univariate regression of Forwardflow runtimes with respect to MLP and Executing occupancy yields median $R^2 = 0.92$ and $R^2 = 0.94$, respectively, suggesting that both play an important role in performance scaling.

OoO suffers from IQ clog in several cases (most dramatically in `libquantum`), limiting performance. Instructions dependent on cache misses fill *OoO*'s scheduler, preventing dispatch—even if ready instructions follow in the subsequent instruction stream. The effect of IQ clog is evident in Figure 9.c: *OoO*'s mean Waiting occupancy is very close to the scheduler size, indicating that the scheduler is often full. The Forwardflow designs, with no disjoint scheduler, do not suffer from scheduler clog (and can therefore accommodate more Waiting instructions). This yields a small runtime reduction of 2.4% for *F-1* versus *OoO*, in spite of the latency of serialized wakeup in Forwardflow cores.

Further insight is offered by considering the throughput of several on-chip structures (Figure 10) as the core is scaled from *F-1* to *F-4*. Not surprisingly, scaling the Forwardflow DQ increases functional unit throughput (*FUs*), but also more aggressively exercises *unscaled* portions of the chip. Throughput at L2 and L3 input ports, on-chip links, memory controllers, and the fetch logic all increase substantially when the core is scaled up. In other words, core scaling enables single threads to better utilize shared chip resources (nominally provisioned for many cores), and to better utilize portions of the core that are not scaled (e.g., *Fetch*).

In three cases, no Forwardflow configuration exceeds the performance of *OoO*. `bzip2` and `hmmer` both exhibit high L1-D hit rates, and thus the latency of serialized wakeup cannot be hidden by accesses to memory (`gromacs` from SPEC FP 2006 behaves similarly). Some benchmarks are not particularly sensitive to changes in window size (e.g., `gcc`, `gobmk`, `sjeng`), as these benchmarks are control intensive, and performance gains are quickly lost to branch misprediction. `sjeng`'s performance actually *degrades* as the window size grows beyond a certain point. Forwardflow's performance does not scale perfectly: scaling larger comes at the cost of increased wire delay when communicating with distant DQ elements. Increased operand network delay, and increased delay in dispatching instructions to distant DQ banks can overcome the benefit gained from increased window capacity and issue bandwidth.

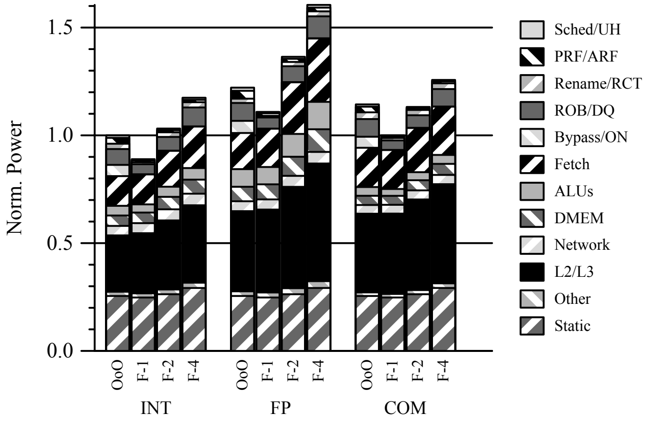


Figure 11. Single-Thread Power Breakdown

5.2 Single-Threaded Power

Figure 11 presents the mean chip-wide power breakdown for each configuration. All power results are normalized to the harmonic mean power consumed by *OoO* when running SPEC INT. We organize power into twelve categories: of those that are not self-explanatory, "Other" includes NoSQ and control logic not suitable for other categories, "Network" refers to the on-chip inter-processor network, "DMEM" includes D-TLBs and L1-D caches, "Fetch" includes I-TLBs and L1-I caches, "Bypass/ON" represents power consumed by *OoO*'s bypassing network or Forwardflow's operand network, and "Sched/UH" represents *OoO*'s scheduler and Forwardflow's update hardware, respectively.

The general trend of power consumption is fairly uniform across all workloads: *F-1* consumes 9.9% less power overall than *OoO,* due to Forwardflow's efficient SRAM-based design. *F-2* and *F-4* tend to exceed *OoO*'s consumption (7.2% and 23% respectively), constituting a dynamic power range of 37% between *F-1* and *F-4*. However, merely exceeding the power consumed by a single active *OoO* core does not imply that the CMP's power budget is exceeded—the power required to operate all cores simultaneously (e.g., Figure 13) is substantially higher than operating *F-4* alone for any one benchmark.

The larger Forwardflow configurations consume more power than *OoO* overall, but much of this increase in power consumption arises because activity increases *elsewhere* on

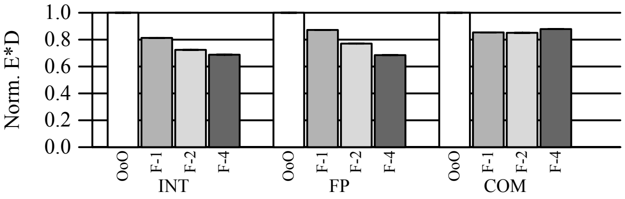


Figure 12. a) Normalized ED

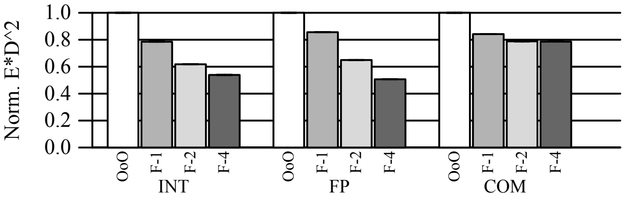


Figure 12. b) Normalized ED^2

the CMP—from the graph, it is evident that the power consumed from Forwardflow-specific components (Sched, RF, RCT, DQ) does not increase as substantially as other core components (e.g., Fetch) or from the on-chip caches (L2/L3). This is a desired result, and follows from the throughputs shown in Figure 10. Forwardflow can scale core performance without concentrating power dissipation in the scaled components.

Figure 12 presents mean ED and ED^2. Because *F-1* is both faster and consumes less power than *OoO*, it is not surprising that *F-1* is more efficient overall by these metrics. One of the three Forwardflow configurations minimizes ED and ED^2 in 30 of 33 of the single-threaded benchmarks. However, *no single configuration is most efficient for all workloads*—12 benchmarks minimize ED with *F-1*, 6 with *F-2*, and 12 with *F-4*. ED^2, with a heavier weight on performance, is optimized in 13 benchmarks by *F-2*, and in 17 by *F-4*. Because no one configuration best suits all workloads, scalable cores in general and Forwardflow cores in particular enable system software to optimize the CMP for peak performance, lowest power consumption, or optimum efficiency, as the situation merits.

5.3 Multithreaded Workloads

We next evaluate the CMP when it is fully utilized by a multithreaded application: either a commercial workload, or one of the SPEC OMP benchmarks. We again evaluate the performance, power, and efficiency of each configuration, but now all eight cores on our CMP are in an active state. In the multithreaded domain, the configuration space becomes significantly more complicated, as the possibility of heterogeneity arises. Though we leave any evaluation of this heterogeneity for future work, we observe that there are a large cross-product of design points between all-*F-1* and all-*F-4*, considering the number of available benchmarks.

Figure 13 plots performance and power of multithreaded benchmarks. Unlike previous runtime graphs, Figure 13 plots speedup normalized to that of *OoO* (*OoO* itself does not explicitly appear on the plots). Each benchmark is represented by a line, beginning with the power/performance point of *F-1*, and continuing to those of *F-2* and *F-4*. Power is normalized to the same scale as Figure 11: Y=1.0 is the power consumed by one *OoO*-core running SPEC INT 2006. Note that SPEC OMP benchmark `art` is absent from the figure, as its behavior is dominated by TLB fill, and does not appear at the chosen scale because of very low overall power consumption.

As with the single-threaded workloads, most of the multithreaded workloads scale in both power consumption and performance from *F-1* to *F-4*, though some benchmarks do not scale at all (e.g., `ammp` and `wupwise` in SPEC OMP). Mean runtime reduction is 12% (8.2%) for OMP (commercial workloads), and is accompanied by an increase in chip power of 32% (40%).

We observe a substantial power range in the objective benchmarks, as each exercises the available resources in a different manner. Depending on the available power budget, it may not be possible to run even scaled-down cores at full speed (e.g., DVFS may be required): *F-1* running `apsi` consumes more than 12x the power of an average single thread of SPEC INT on *OoO*. With the aggressive clock gating used in this study, these ranges are possible, as an individual `apsi` thread is comparable in power consumption to the most power-hungry threads of SPEC CPU (and, of course, there are eight such threads in our OMP benchmarks).

Because we set no quantitative upper bound on our CMP power envelope, we cannot conclude which configurations

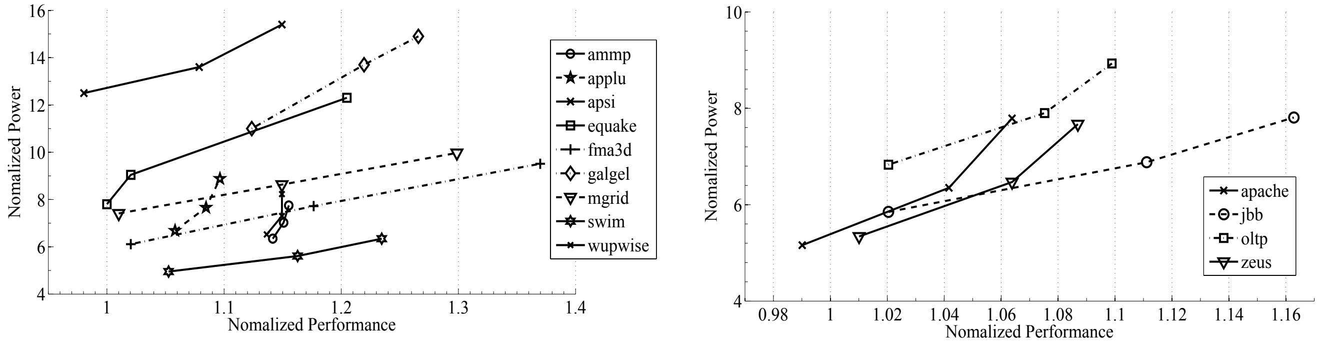


Figure 13. SPEC OMP (left) and Commercial Workload (right) Power vs. Performance

are best. Instead, we briefly consider two hypothetical power budgets. The first is Y=8.0, the power budget at which the CMP is provisioned to allow each core to operate at *OoO*'s power when running SPEC INT. Under this constraint, *OoO* can run 9 of 14 benchmarks at full speed (i.e., without any DVFS to hold power below the accepted maximum). At least one Forwardflow configuration can run without DVFS for all but two of these benchmarks, and several benchmarks (e.g., `equake`, `fma3d`, `jbb`) can safely scale to *F-4*. A second point of interest is Y=14.6, at which all *OoO* configurations can run without employing DVFS. At this power level, all Forwardflow configurations are feasible, except *F-4* running `galgel` or `apsi`. Between these extremes of power budget, Forwardflow configurations—both the evaluated homogeneous and the possible heterogeneous configurations—provide a wealth of dynamic power range for system software to exploit.

As in the single-thread experiments, no single configuration is most efficient for all 14 benchmarks. ED is minimized by *F-1* in 9 cases, by *F-2* in 1, and by *F-4* in the remaining 4. As before, additional performance weight in the ED^2 metric skews the minimum toward more aggressive designs, to *F-1* for 6 benchmarks, *F-2* for 1, and *F-4* for the remaining 7. *OoO* never minimizes ED, nor ED^2.

6 CONCLUSIONS AND FUTURE WORK

Though Moore's Law endures, the fraction of simultaneously active transistors is dropping, and architects must find new methods to deliver both ILP and TLP with a single chip design. Single threads are likely to remain common workloads in many scenarios, but the transition to threading should not be hampered by lack of available parallelism in the hardware. To this end, we have re-evaluated core microarchitecture to design a processor that can scale its execution resources to match the available TLP. Our design, Forwardflow, is a scalable core architecture implementing out-of-order execution with manageable size and complexity. Forwardflow's execution resources can be scaled up to improve single-thread performance by 21% when few threads are available, allowing greater utilization of CMP resources by single threads than a traditional design. We evaluate a Forwardflow design that is ISA-compatible with existing SPARCv9 binaries and operating systems.

The Forwardflow core design itself is efficient and disaggregated. It replaces centralized scheduling logic and register files with a distributed, RAM-based Dataflow Queue (DQ), which can scale gracefully from small to large instruction windows, allowing the system to trade-off power and performance depending on how many DQ banks the system software provisions and enables. This design is more energy-efficient in 44 of 47 studied workloads (by either ED or ED^2 metrics).

Forwardflow joins and, we hope, will be followed by other proposals to address the key issue of single-thread performance in the CMP domain—the design of scalable cores alone is an area worthy of future exploration. Perhaps most importantly, in order for future scalable core designs to flourish, much research remains exploring scaling *policies*. In particular, it is not clear how system software (or some other controlling entity) should go about choosing a core configuration to best match a particular workload in absentia of advance profiling. The complexities of managing power consumption in a CMP are subtle, and as this work has shown, there is no single configuration that will be optimal in all cases. Moreover, benchmark behavior changes in time, and this should be considered in future work addressing these challenges.

ACKNOWLEDGEMENTS

This work is supported in part by the National Science Foundation (NSF), with grants CCR-0324878, CNS-0551401, CNS-0720565, and CCF-0916725, as well as donations from Microsoft and Sun Microsystems/Oracle. The views expressed herein are not necessarily those of the NSF, Microsoft or Sun Microsystems/Oracle. Prof. Wood has a significant financial interest in Microsoft. The authors would like to acknowledge Mark Hill, Yasuko Watanabe, Natalie Enright Jerger, and members of the Multifacet and Multiscalar research groups (past and present), for encouragement, advice, and support. We further acknowledge the attendees of the UW Computer Architecture Affiliates conference for spirited discussion and suggestions, and our anonymous reviewers for their very useful remarks.

REFERENCES

[1] A. R. Alameldeen, C. J. Mauer, M. Xu, P. J. Harper, M. M. K. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood. Evaluating Non-deterministic Multi-threaded Commercial Workloads. In *Proc. of the 5th Workshop on Computer Architecture Evaluation Using Commercial Workloads*, pages 30–38, Feb. 2002.

[2] D. Albonesi, R., Balasubramonian, S. Dropsbo, S. Dwarkadas, F. Friedman, M. Huang, V. Kursun, G. Magklis, M. Scott, G. Semeraro, P. Bose, A. Buyuktosunoglu, P. Cook, and S. Schuster. Dynamically tuning processor resources with adaptive processing. *IEEE Computer*, 36(2):49–58, Dec. 2003.

[3] 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.

[4] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. Jones, and B. Parady. SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. In *Workshop on OpenMP Applications and Tools*, pages 1–10, July 2001.

[5] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Proc. of the 27th Annual Intnl. Symp. on Computer Architecture*, pages 83–94, June 2000.

[6] K. Chakraborty, P. M. Wells, and G. S. Sohi. Computation Spreading: Employing Hardware Migration to Specialize CMP Cores On-the-fly. In *Proc. of the 12th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

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

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

[9] J. L. Henning. SPEC CPU2006 Benchmark Descriptions. *Computer Architecture News*, 34(4):1–17, 2006.

[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] M. Huang, J. Renau, and J. Torrellas. Energy-efficient hybrid wakeup logic. In *ISLPED '02: Proceedings of the 2002 international symposium on Low power electronics and design*, pages 196–201, New York, NY, USA, 2002. ACM.

[13] 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.

[14] Intel. Intel and Core i7 (Nehalem) Dynamic Power Management, 2008.

[15] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez. Core Fusion: Accomodating Software Diversity in Chip Multiprocessors. In *Proc. of the 34th Annual Intnl. Symp. on Computer Architecture*, June 2007.

[16] C. Kim, S. Sethumadhavan, M. S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler. Composable Lightweight Processors. In *Proc. of the 40th Annual IEEE/ACM International Symp. on Microarchitecture*, Dec. 2007.

[17] I. Kim and M. H. Lipasti. Half-price architecture. In *Proc. of the 30th Annual Intnl. Symp. on Computer Architecture*, pages 28–38, June 2003.

[18] A. R. Lebeck, T. Li, E. Rotenberg, J. Koppanalil, and J. P. Patwardhan. A Large, Fast Instruction Window for Tolerating Cache Misses. In *Proc. of the 29th Annual Intnl. Symp. on Computer Architecture*, May 2002.

[19] P. S. Magnusson et al. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[20] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News*, pages 92–99, Sept. 2005.

[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 *Proc. of the 24th Annual Intnl. Symp. 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 *Proc. of the 29th Annual Intnl. Symp. on Computer Architecture*, 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 *Proc. of the 38th Annual IEEE/ACM International Symp. on Microarchitecture*, Nov. 2005.

[27] P. Sassone, J. R. II, E. Brekelbaum, G. Loh, and B. Black. Matrix Scheduler Reloaded. In *Proc. of the 34th Annual Intnl. Symp. on Computer Architecture*, pages 335–346, June 2007.

[28] T. Sha, M. M. K. Martin, and A. Roth. NoSQ: Store-Load Communication without a Store Queue. In *Proc. of the 39th Annual IEEE/ACM International Symp. on Microarchitecture*, pages 285–296, Dec. 2006.

[29] T. Shyamkumar, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. CACTI 5.1. Technical Report HPL-2008-20, Hewlett Packard Labs, 2008.

[30] G. S. Sohi and S. Vajapeyam. Instruction Issue Logic for High-Performance Interruptable Pipelined Processors. In *Proc. of the 14th Annual Intnl. Symp. on Computer Architecture*, pages 27–34, June 1987.

[31] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual Flow Pipelines. In *Proc. of the 11th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 2004.

[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 *Proc. of the 23th Annual Intnl. Symp. on Computer Architecture*, pages 191–202, May 1996.

[34] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation Cores: Reducing the Energy of Mature Computations. In *Proc. of the 9th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

[35] 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.

[36] K. C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–40, Apr. 1996.