2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

MorphCore: An Energy-Efficient Microarchitecture for High Performance ILP and High Throughput TLP

Khubaib[†] M. Aater Suleman[‡] Milad Hashemi[†] Chris Wilkerson[§] Yale N. Patt[†]

[†]The University of Texas at Austin
{khubaib,miladh,patt}@hps.utexas.edu

[‡]Calxeda/HPS
suleman@hps.utexas.edu

[§]Intel Labs, Hillsboro, OR
chris.wilkerson@intel.com

Abstract

Several researchers have recognized in recent years that today's workloads require a microarchitecture that can handle single-threaded code at high performance, and multi-threaded code at high throughput, while consuming no more energy than is necessary. This paper proposes MorphCore, a unique approach to satisfying these competing requirements, by starting with a traditional high performance out-of-order core and making minimal changes that can transform it into a highly-threaded in-order SMT core when necessary. The result is a microarchitecture that outperforms an aggressive 4-way SMT out-of-order core, "medium" out-of-order cores, small in-order cores, and CoreFusion. Compared to a 2-way SMT out-of-order core, MorphCore increases performance by 10% and reduces energy-delay-squared product by 22%.

1. Introduction

Traditional core microarchitectures do not adapt to the thread level parallelism (TLP) available in programs. In general, industry builds two types of cores: large out-of-order cores (e.g., Intel's Sandybridge, IBM's Power 7), and small cores (e.g., Intel's MIC a.k.a Larrabee, Sun's Niagara, ARM's A15). Large out-of-order (OOO) cores provide high single thread performance by exploiting Instruction-Level Parallelism (ILP), but they are power-inefficient for multi-threaded programs because they unnecessarily waste energy on exploiting ILP instead of leveraging the available TLP. In contrast, small cores do not waste energy on wide superscalar OOO execution, but rather provide high parallel throughput at the cost of poor single thread performance.

Heterogeneous (or Asymmetric) Chip Multiprocessors (ACMPs) [11, 22, 28] have been proposed to handle this software diversity. ACMPs provide one or few large cores for speedy execution of single-threaded programs, and many small cores for high throughput in multi-threaded programs. Unfortunately, ACMPs require that the number of large and small cores be fixed at design time, which inhibits adaptability to varying degrees of software thread-level parallelism.

To overcome this limitation of ACMPs, researchers have proposed CoreFusion-like architectures [16, 5, 17, 25, 24, 32, 9, 10]. They propose a chip with small cores to provide high throughput performance in multi-threaded programs. These small cores can dynamically "fuse" into a large core when executing single-threaded programs. Unfortunately, the fused large core has low performance and high power/energy consumption compared to a traditional out-of-order core for two reasons: 1) there are additional latencies among the pipeline stages of the fused core, thus, increasing the latencies of the core's "critical loops", and 2) mode switching requires instruction cache flushes and incurs the cost of data migration among the data caches of small cores.

To overcome these limitations, we propose MorphCore, an adaptive core microarchitecture that takes the opposite approach of previously proposed reconfigurable cores. Rather than fusing small cores into a large core, MorphCore uses a large out-of-order core as the base substrate and adds the capability of in-order SMT to exploit highly parallel code. MorphCore provides two modes of execution: OutOfOrder and InOrder. In OutOfOrder mode, MorphCore provides the single-thread performance of a traditional out-of-order core with *minimal* performance degradation. However, when TLP is available, MorphCore "morphs" into a highly-threaded in-order SMT core. This allows MorphCore to hide long latency operations by executing operations from different threads concurrently. Consequently, it can achieve a higher throughput than the out-of-order core. Since no migration of instructions or data needs to happen on mode switches, MorphCore can switch between modes with minimal penalty.

MorphCore is built on two key insights. First, a highly-threaded (i.e., 6-8 way SMT) in-order core can achieve the same or better performance as an out-of-order core. Second, a highly-threaded in-order SMT core can be built using a subset of the hardware required to build an aggressive out-of-order core. For example, we use the Physical Register File (PRF) in the out-of-order core as the architectural register files for the many SMT threads in InOrder mode. Similarly, we use the Reservation Station entries as an in-order instruction buffer and the execution pipeline of the out-of-order core as-is.

MorphCore is more energy-efficient than a traditional out-of-order core when executing multi-threaded programs. It reduces execution time by exploiting TLP, and reduces energy consumption by turning off several power hungry structures (e.g., renaming logic, out-of-order scheduling, and the load queue) while in InOrder mode.

Our evaluation with 14 single-threaded and 14 multi-threaded workloads shows that MorphCore increases performance by 10% and reduces energy-delay-squared product by 22% over a typical 2-way SMT out-of-order core. We also compare MorphCore against three different core architectures optimized for different performance/energy design points, and against CoreFusion, a reconfigurable core architecture. We find that MorphCore performs best in terms of performance and energy-delay-squared product across a wide spectrum of single-threaded and multi-threaded workloads.

Contributions: This paper makes two contributions:

1. We present *MorphCore*, a new microarchitecture that combines out-of-order and highly-threaded in-order SMT execution within a single core. We comprehensively describe the microarchitecture needed to implement MorphCore, and the policy to switch between modes.
2. To the best of our knowledge, this is the first paper to quantitatively compare small, medium and large core architectures in

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.36

IEEE computer society

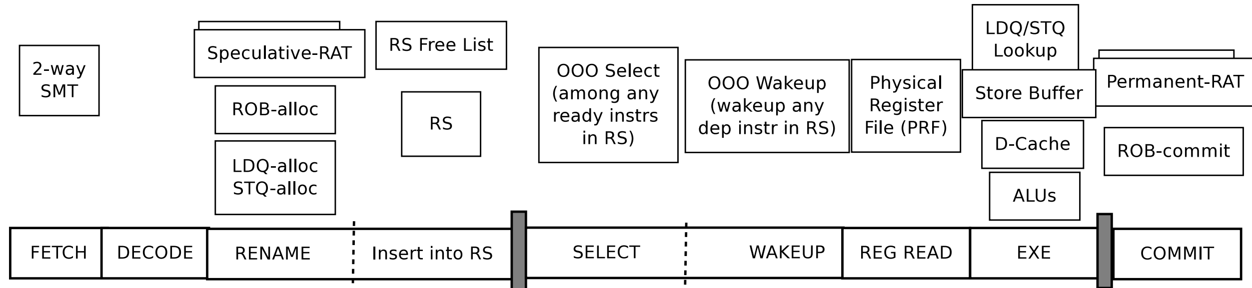


Figure 1: Out of Order core microarchitecture

terms of performance and energy-efficiency on single-threaded and multi-threaded workloads.

2. Background and Motivation

2.1. Out-of-order Execution

Out-of-order (OOO) cores provide better performance by executing instructions as soon as their operands become available, rather than executing them in program order. Figure 1 shows a high-level layout of a 2-way SMT OOO core pipeline. The top part shows major structures accessed and functionality performed in different stages of the pipeline. We describe a Pentium-4 like architecture [26], where the data, both speculative and architectural, is stored in the Physical Register File (PRF), and the per-thread Register Alias Table (RAT) entries point to PRF entries. The front-end Speculative-RAT points to the speculative state, and a back-end Permanent-RAT points to the architectural state. The front-end of the pipeline (from the Fetch stage until the Insert into Reservation Station (RS)) works in-order. Instructions are fetched, decoded, and then sent to the Rename Stage. The Rename stage renames (i.e. maps) the architectural source and destination register IDs into Physical Register File IDs by reading the Speculative-RAT of the thread for which instructions are being renamed, and inserts the instructions into the Reservation Station (also referred to as Issue Queue).

Instructions wait in the Reservation Station until they are selected for execution by the Select stage. The Select stage selects an instruction for execution once all of the source operands of the instruction are ready, and the instruction is the oldest among the ready instructions. When an instruction is selected for execution, it readies its dependent instructions via the Wakeup Logic block, reads its source operands from the PRF, and executes in a Functional Unit. After execution, an instruction's result is broadcast on the Bypass Network, so that any dependent instruction can use it immediately. The result is also written into the PRF, and the instruction updates its ROB status. The instruction retires once it reaches the head of the ROB, and updates the corresponding Permanent-RAT.

Problem With Wide Superscalar Out-of-order Execution. Unfortunately, the single-thread performance benefit of the large out-of-order (OOO) core comes with a power penalty. As we show in Section 6.2, a large OOO core consumes 92% more power than a medium OOO core, and 4.3x more than a small in-order core. This overhead exists to exploit instruction-level parallelism to increase core throughput, and is justified when the software has a single thread of execution. However, when multiple threads of execution exist, we propose that the core can be better utilized using *in-order* Simultaneous Multi-Threading (SMT).

2.2. Simultaneous Multi-Threading

Simultaneous Multi-Threading (SMT) [13, 35, 31] is a technique to improve the utilization of execution resources using multiple threads provided by the software. In SMT, a core executes instructions from multiple threads concurrently. Every cycle, the core picks a thread from potentially many ready threads, and fetches instructions from that thread. The instructions are then decoded and renamed in a regular pipelined fashion and inserted into a common (shared among all the threads) RS. Since instructions from multiple candidate threads are available in the RS, the possibility of finding ready instructions increases. Thus, SMT cores can achieve higher throughput provided that software exposes multiple threads to the hardware.

The Potential of In-Order SMT on a Wide Superscalar Core. The observation that a highly multi-threaded in-order core can achieve the instruction issue throughput similar to an OOO core was noted by Hily and Seznec [12]. We build on this insight to design a core that can achieve high-performance and low-energy consumption when software parallelism is available.

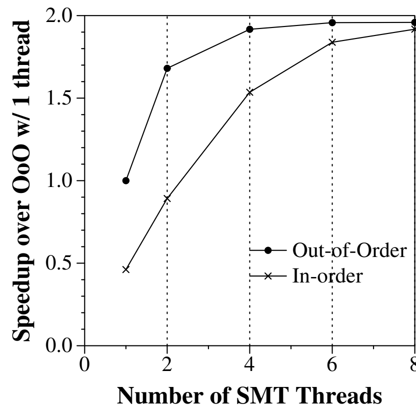


Figure 2: Performance of `black` with SMT

Figure 2 shows the performance of the workload `black` (Black-Scholes pricing [23]) on an out-of-order and an in-order core. For this experiment, both of the cores are similarly sized in terms of cache sizes, pipeline widths (4-wide superscalar) and depths (refer to Section 5 for experimental methodology). The performance of the in-order core is significantly less than the performance of the out-of-order core when both cores run only a single thread. As the

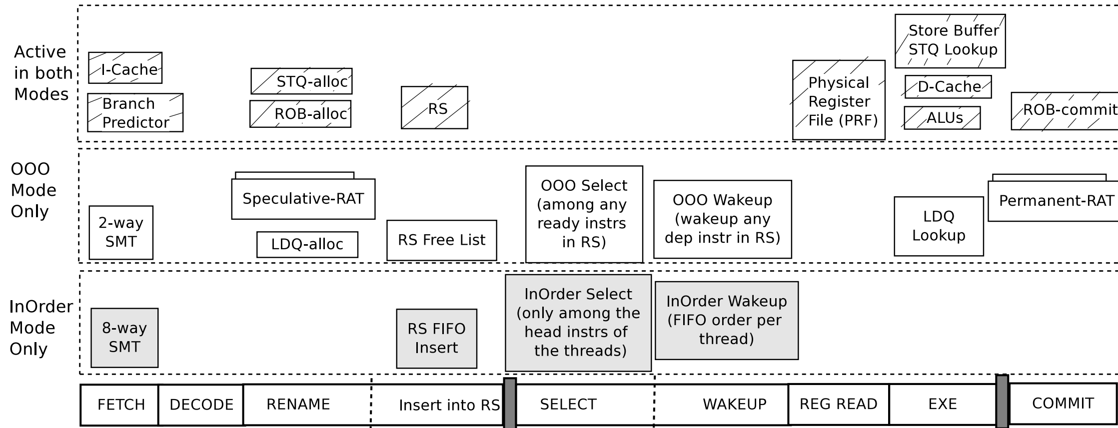


Figure 3: The MorphCore microarchitecture

number of SMT threads increases from 1 to 8, the performance of the out-of-order core increases significantly at 2 threads, but starts to saturate at 4 threads, because the performance is limited by the peak throughput of the core. In contrast, the performance of the in-order core continues to benefit from more threads (which allows it to better tolerate long latency operations and memory accesses). When the number of threads is equal to 8, the in-order core's performance begins to match the performance of the out-of-order core. This experiment shows that when high thread-level parallelism is available, high performance and low energy consumption can be achieved with *in-order SMT* execution, therefore the core need not be built with complex and power hungry structures needed for out-of-order SMT execution.

Summary. In spite of its high power cost, out-of-order execution is still desirable because it provides significant performance improvement over in-order execution. Thus, if we want high single-thread performance we need to keep support for out-of-order execution. However, when software parallelism is available, we can provide performance by using in-order SMT and not waste energy on out-of-order execution. To accomplish both, we propose the MorphCore architecture.

3. MorphCore Microarchitecture

3.1. Overview of the MorphCore Microarchitecture

The MorphCore microarchitecture is based on a traditional OOO core. Figure 3 shows the changes that are made to a baseline OOO core (shown in Figure 1) to build the MorphCore. It also shows the blocks that are active in both modes, and the blocks that are active only in one of the modes. In addition to out-of-order execution, MorphCore supports additional in-order SMT threads, and in-order scheduling, execution, and commit of simultaneously running threads. In OutofOrder mode, MorphCore works exactly like a traditional out-of-order core.

3.2. Fetch and Decode Stages.

The Fetch and Decode Stages of MorphCore work exactly like an SMT-enabled traditional OOO core. Figure 4 shows the Fetch Stage of the MorphCore. MorphCore adds 6 additional SMT contexts to the baseline core. Each context consists of a PC, a branch history register, and a Return Address Stack. In OutofOrder mode, only 2 of the SMT contexts are active. In InOrder mode, all 8 contexts are active. The branch predictor and the I-Cache are active in both modes.

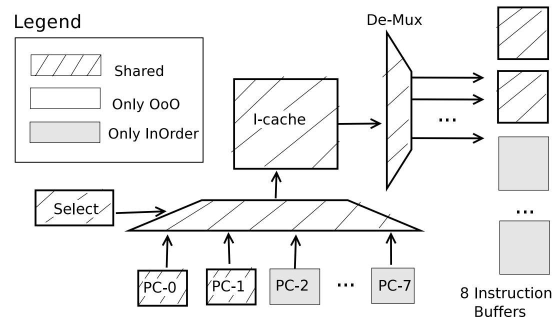


Figure 4: Microarchitecture of the Fetch stage

3.3. Rename Stage

Figure 5 shows the Rename Stage of the MorphCore. InOrder renaming is substantially simpler, and thus power-efficient, than OOO renaming. In InOrder mode, we use the Physical Register File (PRF) to store the architectural registers of the multiple in-order SMT threads: we logically divide the PRF into multiple fixed-size partitions where each partition stores the architectural register state of a thread (Figure 5(b)). Hence, the architectural register IDs can be mapped to the Physical Register IDs by simply concatenating the Thread ID with the architectural register ID. This approach limits the number of in-order SMT threads that the MorphCore can support to $num_physical_registers/num_architectural_registers$. However, the number of physical registers in today's cores is already large enough (and increasing) to support 8 in-order SMT threads which is sufficient to match the out-of-order core's performance. For the x86 ISA [15] that we model in our simulator, a FP-PRF partition of 24 entries and an INT-PRF partition of 16 entries per thread is enough to hold the architectural registers of a thread. The registers that are not renamed and are replicated 2-ways in the baseline OOO core need to be replicated 8-ways in MorphCore.

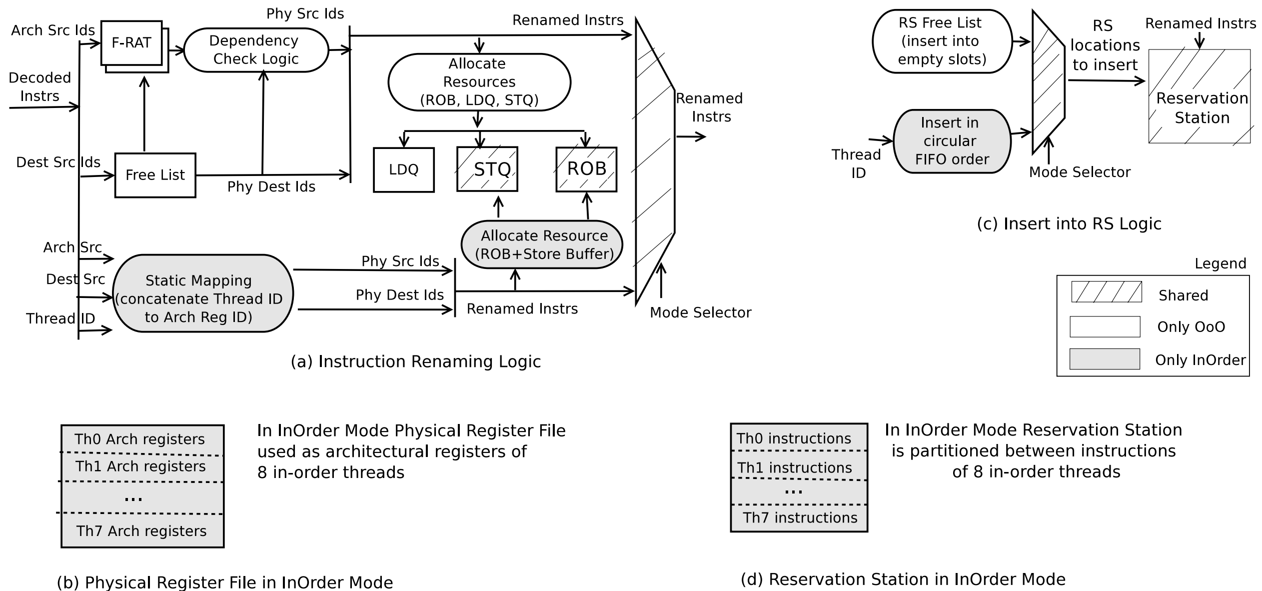


Figure 5: Microarchitecture of the Rename stage

Allocating/Updating the Resources. When the MorphCore is in OutofOrder mode, the instructions that are being renamed are allocated resources in the ROB and in the Load and Store Queues. In InOrder mode, MorphCore leverages the ROB to store the instruction information. We partition the ROB into multiple fixed-size chunks, one for each active thread. We do not allocate resources in the Load Queue in InOrder mode since memory instructions are not executed speculatively. Thus, the Load Queue is inactive. The Store Queue that holds the data from committed store instructions and the data that is not yet committed to the D-cache, is active in InOrder Mode as well, and is equally partitioned among the threads.

Insert into the Reservation Station (RS). Figure 5(c) shows the part of Rename stage that inserts renamed instructions into the RS. In OutofOrder mode, the RS is dynamically shared between multiple threads, and the RS entry that is allocated to an incoming renamed instruction is determined dynamically by consulting a Free List. In InOrder mode, the RS is divided among the multiple threads into fixed-size partitions (Figure 5(d)), and each partition operates as a circular FIFO. Instructions are inserted into consecutive RS entries pointed to by a per-thread RS-Insert-Ptr, and removed in-order after successful execution.

3.4. Select and Wakeup

MorphCore employs both OutofOrder and InOrder Wakeup and Select Logic. The Wakeup Logic makes instructions ready for execution, and the Select Logic selects the instructions to execute from the pool of ready instructions. Figure 6 shows these logic blocks.

OutofOrder Wakeup. OutofOrder Wakeup logic works exactly the same as a traditional out-of-order core. Figure 6 (unshaded) shows the structure of an RS entry [27]. An operand is marked ready (R-bit is set) when the corresponding MATCH bit has been set for the number of cycles specified in the DELAY field. When an instruction fires, it broadcasts its destination tag (power hungry), so that it can be compared against source tags of all instructions in the RS. If the destination tag matches the source tag of an operand, the MATCH bit is set and the DELAY field is set equal to the execution latency of the firing instruction (the latency of the instruction is stored in the RS entry allocated to the instruction). The DELAY field is also latched in the SHIFT field associated with the source tag. The SHIFT field is right shifted one-bit every cycle the MATCH bit is set. The R bit is set when the SHIFT field becomes zero. The RS-entry waits until both sources are ready, and then raises the Req OOO Exec line.

OutofOrder Select. The OutofOrder Select logic monitors *all* instructions in the RS (power hungry), and selects the oldest instruction(s) that have the Req OOO Exec lines set. The output of the Select Logic is a Grant bit vector, in which every bit corresponds to an RS entry indicating which instructions will fire next. When an instruction is fired, the SCHEDULED bit is set in the RS entry so that the RS entry stops requesting execution in subsequent cycles.

InOrder Wakeup. The InOrder mode executes/schedules instructions in-order, i.e., an instruction becomes ready after the previous instruction has either started execution or is ready and independent. We add 2 new bit-fields to each RS entry for in-order scheduling (Scheduled, and MATCH (M)). The new fields are shaded in Figure 6. The InOrder Wakeup Logic block also maintains the M/DELAY/SHIFT/R bit fields per architectural register, in order to track the availability of architectural registers. When an instruction fires, it sets the R, M, and DELAY bit fields corresponding to the destination register in the InOrder Wakeup Logic block as follows: resets the R bit, sets the MATCH (M) bit, and sets the DELAY field to the execution latency of the firing instruction (the DELAY/SHIFT mechanism works as explained above). Each cycle, for every thread, the InOrder Wakeup checks the availability of source registers of the two oldest instructions (R bit is set). If the sources are available, the Wakeup logic readies the instructions by setting the M bit in the RS entry to 1. The InOrder Wakeup is power-efficient since it avoids the broadcast and matching of the destination tag against the source

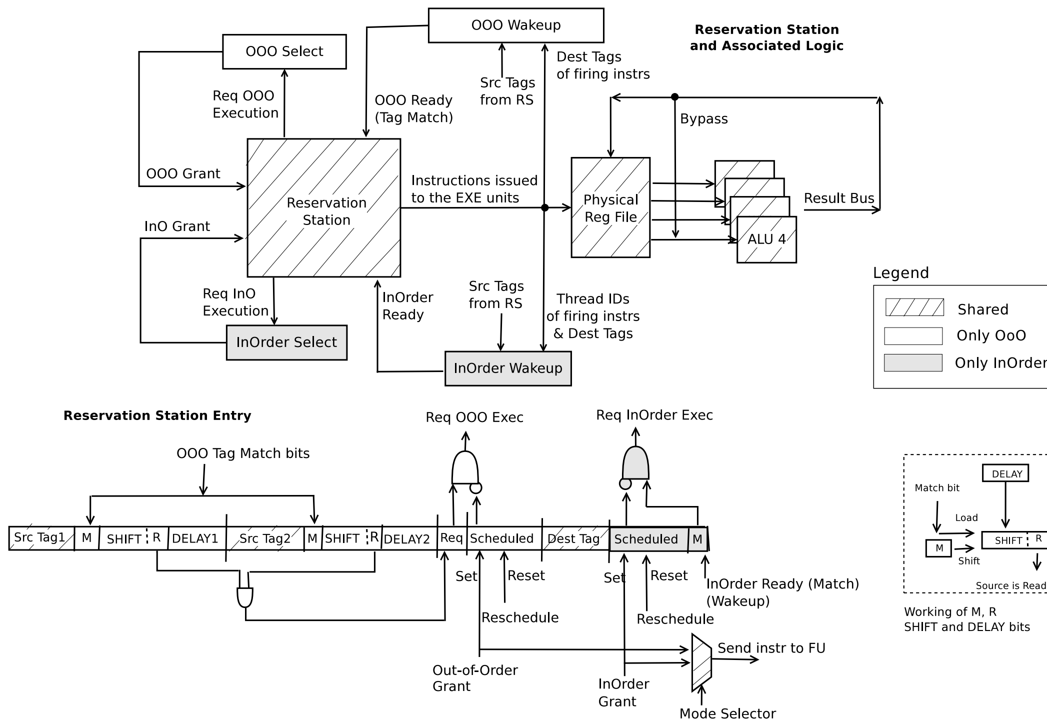


Figure 6: MorphCore Wakeup and Selection Logic

operands of all instructions in the RS.

InOrder Select. The InOrder Select Logic block works hierarchically in a complexity-effective (power-efficient) manner by maintaining eight InOrder select blocks (one per thread) and another block to select between the outcomes of these blocks. Furthermore, each in-order select logic only monitors the two oldest instructions in the thread's RS partition, rather than monitoring the entire RS as in OOO select. Note that only two instructions need monitoring in InOrder mode because instructions from each thread are inserted and scheduled/removed in a FIFO manner.

3.5. Execution and Commit

When an instruction is selected for execution, it reads its source operands from the PRF, executes in an ALU, and broadcasts its result on the bypass network as done in a traditional OOO core. In MorphCore, an additional PRF-bypass and data storage is active in InOrder mode. This bypass and buffering is provided in order to delay the write of younger instruction(s) in the PRF if an older longer latency instruction is in the execution pipeline. In such a case, younger instruction(s) write into a temporary small data buffer (4-entry per thread). The buffer adds an extra bypass in PRF-read stage. Instructions commit in traditional SMT fashion. For OutofOrder commit, the Permanent-RAT is updated as well. In InOrder mode, only the thread's ROB Head pointer needs to be updated.

3.6. Load/Store Unit

Figure 7 shows the Load/Store Unit. In OutofOrder mode, load/store instructions are executed speculatively and out of order (similar to a traditional OOO core). When a load fires, it updates its entry in

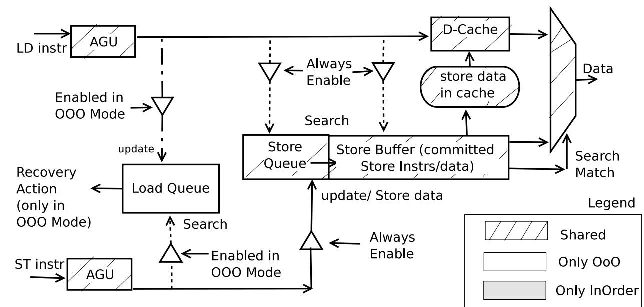


Figure 7: Load / Store unit

the Load Queue and searches the Store Queue to get the latest data. When a store fires, it updates and stores data in the Store Queue, and searches the Load Queue to detect store-to-load program order violations. In InOrder mode, since load/store instructions are not executed speculatively, no Load Queue CAM searches are done. However, loads still search the Store Queue that holds committed data. Store instructions also update the Store Queue.

3.7. Recovering from Branch Mispredictions

In OutofOrder mode, a branch misprediction triggers a recovery mechanism that recovers the F-RAT to the state prior to the renaming of the mispredicted branch instruction. In InOrder mode, a branch misprediction squashes the instructions in the RS partition, the ROB partition and the front-end pipeline from the thread, followed by redirection of the PC to the correct target.

4. MorphCore Discussion

4.1. Area and Power Overhead of MorphCore

First, MorphCore increases the number of SMT ways from 2 to 8. This adds hardware to the Fetch stage and other parts of the core, which is less than 0.5% area overhead as reported by our modified McPAT [20] tool. Note that it does not incur the two biggest overheads of adding SMT contexts in an OOO core –additional Rename tables and physical registers– because the SMT contexts being added are in-order. Second, MorphCore adds InOrder Wakeup and Select logic, which we assume adds an area overhead of less than 0.5% of core area, half the area of the OOO Wakeup and Select logic blocks. Third, adding extra bypass/buffering adds an area overhead of 0.5% of core. Thus, MorphCore adds an area overhead of 1.5%, and a power overhead of 1.5% in InOrder mode.

4.2. Timing/Frequency Impact of MorphCore

MorphCore requires only two key changes to the baseline OOO core:

1) InOrder renaming/scheduling/execution logic. MorphCore adds a multiplexer in the critical path of three stages: a) in the Rename stage to select between OutofOrder mode and InOrder mode renamed instructions, b) in the Instruction Scheduling stage to select between the OutofOrder mode and InOrder mode ready instructions, and c) in PRF-read stage because of additional bypassing in InOrder mode. In order to estimate the frequency impact of this overhead, we assume that a multiplexer introduces a delay of one transmission gate, which we assume to be half of an FO4 gate delay. Assuming 20 FO4 gate delays per pipeline stage [33, 7], we estimate that MorphCore runs 2.5% slower than the baseline OOO core.

2) More SMT contexts. Addition of in-order SMT contexts can lengthen the thread selection logic in MorphCore's front-end. This overhead is changing the multiplexer that selects one out of many ready threads from 2-to-1 to 8-to-1. We assume that running MorphCore 2.5% slower than the baseline OOO core hides this delay.

In addition to the above mentioned timing-critical changes to the baseline OOO core, MorphCore adds InOrder Wakeup and Select logic blocks. Because InOrder instruction scheduling is simpler than OutofOrder instruction scheduling, we assume that newly added blocks can be placed and routed such that they do not affect the critical path of other components of the baseline OOO core. Thus, we conclude that the frequency impact of MorphCore is only 2.5%.

4.3. Turning Off Structures in InOrder Mode

The structures that are inactive in InOrder Mode (OOO renaming logic, OOO scheduling, and load queue) are unit-level clock-gated. Thus, no dynamic energy is consumed, but static energy is still consumed. Unit-level power-gating could be applied to further cut-down static energy as well, but we chose not to do so, since according to our McPAT estimates, static energy consumed by these structures is very small, whereas the overhead incurred by unit-level power-gating is significant.

4.4. Interaction with OS

MorphCore does not require any changes to the operating system, and acts like a core with the number of hardware threads equal to the maximum number of threads supported in the InOrder Mode (8 in our implementation). Switching between the two modes is handled in hardware.

4.5. When does MorphCore Switch Modes?

In our current implementation of MorphCore, we switch between modes based on the number of active threads (other policies are part of our future work). A thread is active when it is not waiting on any synchronization event. The MorphCore starts running in OutofOrder mode when the number of active threads is less than a threshold (2 in our initial implementation). If the OS schedules more threads on MorphCore, and the number of active threads becomes greater than the threshold, the core switches to InOrder mode. While running in InOrder mode, the number of active threads can drop for two reasons: the OS can de-schedule some threads or the threads can become inactive waiting for synchronization. We assume that the threading library uses MONITOR/MWAIT [15] instructions such that MorphCore hardware can detect a thread becoming inactive, e.g., inactive at a barrier waiting for other threads to reach the barrier, or inactive at a lock-acquire waiting for another thread to release the lock. If the number of active threads becomes smaller than or equal to the threshold, the core switches back to OutofOrder mode until more threads are scheduled or become active (the hardware makes the thread active when a write to the cacheline being monitored is detected).

4.6. Changing Mode from OutofOrder to InOrder

Mode switching is handled by a micro-code routine that performs the following tasks:

1) Drains the core pipeline.

2) Spills the architectural registers of all threads. We spill these registers to a reserved memory region. To avoid cache misses on these writes, we use Full Cache Line Write instructions that do not read the cache line before the write [15].

3) Turns off the Renaming unit, OutofOrder Wakeup and Select Logic blocks, and Load Queue. Note that these units do not necessarily need to be power-gated (we assume that these units are clock-gated).

4) Fills the register values back into each thread's PRF partitions. This is done using special load micro-ops that directly address the PRF entries without going through renaming.

4.7. Changing Mode from InOrder to OutofOrder

Since MorphCore supports more threads in InOrder Mode than in OutofOrder Mode, when switched into OutofOrder mode, MorphCore cannot run all the threads simultaneously and out-of-order. Thus some of the threads need to be marked inactive or "not running" (unless they are already inactive, which is the case in our current implementation). The state of the inactive threads is stored in memory until they become active. To load the state of the active threads, the MorphCore stores pointers to the architectural state of the inactive threads in a structure called the Active Threads Table. The Active Threads Table is indexed using the Thread ID, and stores an 8-byte pointer for each thread. Mode switching is handled by a micro-code routine that performs the following tasks:

1) Drains the core pipeline.

2) Spills the architectural registers of all threads. Store pointers to the architectural state of the inactive threads in the Active Thread Table.

3) Turns on the Renaming unit, OutofOrder Wakeup and Select Logic blocks, and Load Queue.

4) Fills the architectural registers of only the active threads into pre-determined locations in PRF, and updates the Speculative-RAT and Permanent-RAT.

Table 1: Configuration of the simulated machine

Core Configurations	
OOO-2	**Core:** 3.4GHz, 4-wide issue OOO, 2-way SMT, 14-stage pipeline, 64-entry unified Reservation Station (Issue Queue), 192 ROB, 50 LDQ, 40 STQ, 192 INT/FP Physical Reg File, 1-cycle wakeup/select **Functional Units:** 2 ALU/AGUs, 2 ALU/MULs, 2 FP units. ALU latencies (cycles): int arith 1, int mul 4-pipelined, fp arith 4-pipelined, fp divide 8, loads/stores 1+2-cycle D-cache **L1 Caches:** 32KB I-cache, D-cache 32KB, 2 ports, 8-way, 2-cycle pipelined **SMT:** Stages select round-robin among ready threads. ROB, RS, and instr buffers shared as in Pentium 4 [18]
OOO-4	3.23GHz (5% slower than OOO-2), 4-wide issue OOO, 4-way SMT, Other parameters are same as OOO-2.
MED	**Core:** 3.4GHz, 2-wide issue OOO, 1 Thread, 10-stage, 48-entry ROB/PRF. **Functional Units:** Half of OOO-2. Latencies same as OOO-2. **L1 Caches:** 1 port Dcache, other same as OOO-2. **SMT:** N/A
SMALL	**Core:** 3.4GHz, 2-wide issue In-Order, 2-way SMT, 8-stage pipeline. **Functional Units:** Same as MED. **L1 Caches:** Same as MED. **SMT:** Round-Robin Fetch
MorphCore	**Core:** 3.315GHz (2.5% slower than OOO-2), Other parameters are same as OOO-2. **Functional Units and L1 Caches:** Same as OOO-2. **SMT and Mode switching:** 2-way SMT similar to OOO-2, 8-way in-order SMT (Round-Robin Fetch) in InOrder mode. RS and PRF partitioned in equal sizes among the in-order threads. InOrder mode when active threads > 2, otherwise, OutofOrder mode
Memory System Configuration	
Caches	**L2 Cache:** private L2 256KB, 8-way, 5 cycles. **L3 Cache:** 2MB write-back, 64B lines, 16-way, 10-cycle access
Memory	8 banks/channel, 2 channels, DDR3 1333MHz, bank conflicts, queuing delays modeled. 16KB row-buffe, 15 ns row-buffer hit latency

Table 2: Characteristics of Evaluated Architectures

Core	Type	Freq (Ghz)	Issue-width	Num of cores	SMT threads per core	Total Threads	Total Norm. Area	Peak ST throughput	Peak MT throughput
OOO-2	out-of-order	3.4	4	1	2	2	1	4 ops/cycle	4 ops/cycle
OOO-4	out-of-order	3.23	4	1	4	4	1.05	4 ops/cycle	4 ops/cycle
MED	out-of-order	3.4	2	3	1	3	1.18	2 ops/cycle	6 ops/cycle
SMALL	in-order	3.4	2	3	2	6	0.97	2 ops/cycle	6 ops/cycle
MorphCore	out-of-order or in-order	3.315	4	1	OutOfOrder: 2, or InOrder: 8	2 or 8	1.015	4 ops/cycle	4 ops/cycle

4.8. Overheads of Changing the Mode

The overhead of changing the mode is pipeline drain, which varies with the workload, and the spill or fill of architectural register state of the threads. The x86 ISA [15] specifies an architectural state of ∼780 bytes per thread (including the latest AVX extensions). The micro-code routine takes ∼30 cycles to spill or fill the architectural register state of each thread after the pipeline drain (a total of ∼6KB and ∼250 cycles for 8 threads) into reserved ways of the private L2 cache (assuming a 256 bit wide read/write port to the cache, and a cache bandwidth of 1 read/write per cycle). We have empirically observed no loss in performance by taking away ∼6KB from the private 256KB cache. Note that the overhead of changing the mode can be reduced significantly by overlapping the spilling or filling of the architectural state with the pipeline drain. It is our future work to explore such mechanisms.

5. Experimental Methodology

Table 1 shows the configurations of the cores and the memory subsystem simulated using our in-house cycle-level x86 simulator. The simulator faithfully models microarchitectural details of the core, cache hierarchy and memory subsystem, e.g., contention for shared resources, DRAM bank conflicts, banked caches, etc. To estimate the area and power/energy of different core architectures, we use a modified version of McPAT [20]. We modified McPAT to: 1) report finer-grain area and power data, 2) increase SMT ways without increasing the Rename (RAT) tables, 3) use the area/energy impact of InOrder scheduling (1/2 of OOO), 4) model extra bypass/buffering, and 5) model the impact of SMT more accurately. Note that all core configurations have the same memory subsystem (L2, L3 and main memory).

Table 2 summarizes the key characteristics of the compared architectures. We run the baseline OOO-2 core at 3.4GHz and scale the frequencies of the other cores to incorporate the effects of both increase in area and critical-path-delay. For example, OOO-4's frequency is 5% lower than OOO-2 because adding the 2 extra SMT threads significantly increases the area/complexity of the core: it adds two extra Rename tables (RATs), at least a multiplexer at the end of Rename stage, and also adds extra buffering at the start of Rename stage (to select between 4, rather than 2 rename tables) which we estimate (using McPAT) to be an additional 5% area and thus lower frequency by 5%. MorphCore's frequency is reduced by 2.5% because its critical path increased by 2.5% (as explained in Section 4.2). Since the OOO-2 core has the highest frequency and supports 4-wide superscalar OOO execution, we can expect it to have the highest single thread (ST) performance. Since the SMALL and MED cores have the highest aggregate ops/cycle, we can expect them to have the highest multi-threaded (MT) performance. We expect the MorphCore to perform close to best in both ST and MT workloads. In Section 7.1, we also compare MorphCore against CoreFusion [16], a representative of reconfigurable core architectures proposed to date.

5.1. Workloads

Table 3 shows the description and input-set for each application. We simulate 14 single-threaded SPEC 2006 applications and 14 multi-threaded applications from different domains. We limit the number of single-thread workloads to 14 to ensure that the number of single-thread and multi-thread workloads is equal, so that the single-thread results do not dominate the overall average performance data. We randomly choose the 14 SPEC workloads. Each SPEC benchmark

Table 3: Details of the simulated workloads

Workload	Problem description	Input set
Multi-Threaded Workloads		
web	web cache [29]	500K queries
qsort	Quicksort [8]	20K elements
tsp	Traveling salesman [19]	11 cities
OLTP-1	MySQL server [2]	OLTP-simple [3]
OLTP-2	MySQL server [2]	OLTP-complex [3]
OLTP-3	MySQL server [2]	OLTP-nontrx [3]
black	Black-Scholes [23]	1M options
barnes	SPLASH-2 [34]	2K particles
fft	SPLASH-2 [34]	16K points
lu (contig)	SPLASH-2 [34]	512x512 matrix
ocean (contig)	SPLASH-2 [34]	130x130 grid
radix	SPLASH-2 [34]	300000 keys
ray	SPLASH-2 [34]	teapot.env
water (spatial)	SPLASH-2 [34]	512 molecules
Single-Threaded Workloads		
SPEC 2006	7 INT and 7 FP benchmarks	200M instrs

is run for 200M instructions with ref input set, where the representative slice is chosen using a Simpoint-like methodology. We do so since SPEC workloads are substantially longer (billions of instructions), and easier to sample using existing techniques like SimPoint. Single-threaded workloads run on a single core with other cores turned off. In contrast, multi-threaded workloads run with the number of threads set equal to the number of available contexts, i.e., $number of cores \times number of SMT contexts$. We run all multi-threaded workloads to completion and count only useful instructions, excluding synchronization instructions. Statistics are collected only in the parallel region, and initialization phases are ignored. For reference, Figure 8 shows the percentage of execution time in multi-threaded workloads when a certain number of threads are active. A thread is active when it is not waiting on any synchronization event. We will refer to this data when presenting our results next.

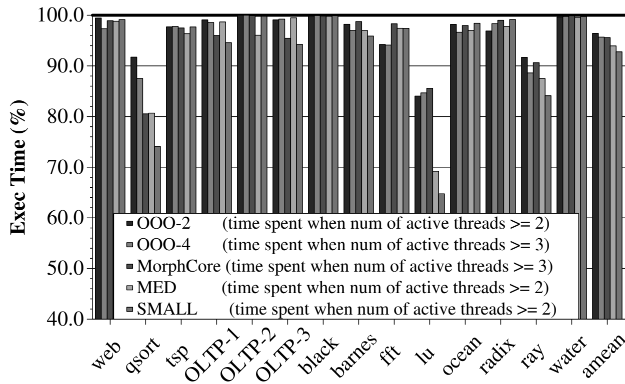


Figure 8: Percentage of execution time when a certain number of threads are active

6. Results

Since MorphCore attempts to improve performance and reduce energy, we compare our evaluated architectures on performance, energy-consumption, and a combined performance-energy metric, the energy-delay-squared product.

Table 4: Micro-op throughput (uops/cycle) on OOO-2

web	qsort	tsp	OLTP-1	OLTP-2	OLTP-3	black
3.47	1.95	2.78	2.29	2.23	2.35	3.39
`barnes`	`fft`	`lu`	`ocean`	`radix`	`ray`	`water`
3.31	2.68	3.11	2.17	1.94	3.06	3.51

6.1. Performance Results

Since design and performance trade-offs of ST and MT workloads are inherently different, we evaluate their performance separately. We will present a combined average across all ST and MT workloads in Section 6.1.3.

6.1.1. Single-Thread (ST) Performance Results. Figure 9a shows the speedup of each core normalized to OOO-2. As expected, OOO-2 achieves the highest performance on all workloads. The MorphCore is a close second. This is because MorphCore introduces minimal changes to a traditional out-of-order core. As a result of these changes, MorphCore runs at a 2.5% slower frequency than OOO-2, achieving 98.8% of the performance of OOO-2. The OOO-4 core provides slightly lower performance than MorphCore. This is because OOO-4 has a higher overhead when running in ST mode, a 5% frequency penalty, as it supports 4 OOO SMT threads. Note that the difference in performance among OOO-2, OOO-4, and MorphCore is the smallest for memory-bound workloads, e.g., `mcf`, `GemsFDTD`, and `lbm`. On the other hand, the cores optimized for multi-thread performance, MED and SMALL, have issue widths of 2 (as opposed to 4 for ST optimized cores) and either run in-order (SMALL) or out-of-order with a small window (MED). This results in significant performance loss in ST workloads: MED loses performance by 25% and SMALL by 59% as compared to OOO-2. The performance loss is more pronounced for FP workloads (right half of figure) as compared to INT workloads. In summary, MorphCore provides the second best performance (98.8% of OOO-2) on ST workloads.

6.1.2. Multi-Thread (MT) Performance Results. Multi-thread (MT) performance is affected by not only the performance potential of a single core, but the total number of cores and SMT threads on the cores. Figure 9b shows the speedup of each core normalized to OOO-2. As expected, the throughput optimized cores, MED and SMALL, provide the best MT performance (on average 30% and 33% performance improvement over OOO-2 respectively). This is because MED and SMALL cores have higher total peak throughput even though they take approximately the same area as OOO-2 (see Table 2).

More importantly, MorphCore provides a significant 22% performance improvement over OOO-2. MorphCore provides the highest performance improvement for workloads that have low micro-op execution throughput (uops/cycle) when run on the baseline OOO-2 core (Table 4). This is because MorphCore provides better latency tolerance and increases core throughput by executing up to 8 threads simultaneously. For example, `radix` gets the highest performance improvement of 84% over OOO-2 by increasing the uops/cycle from 1.94 to 3.58. In fact, MorphCore outperforms MED cores by 15% on `radix` because of its ability to run more SMT threads as compared

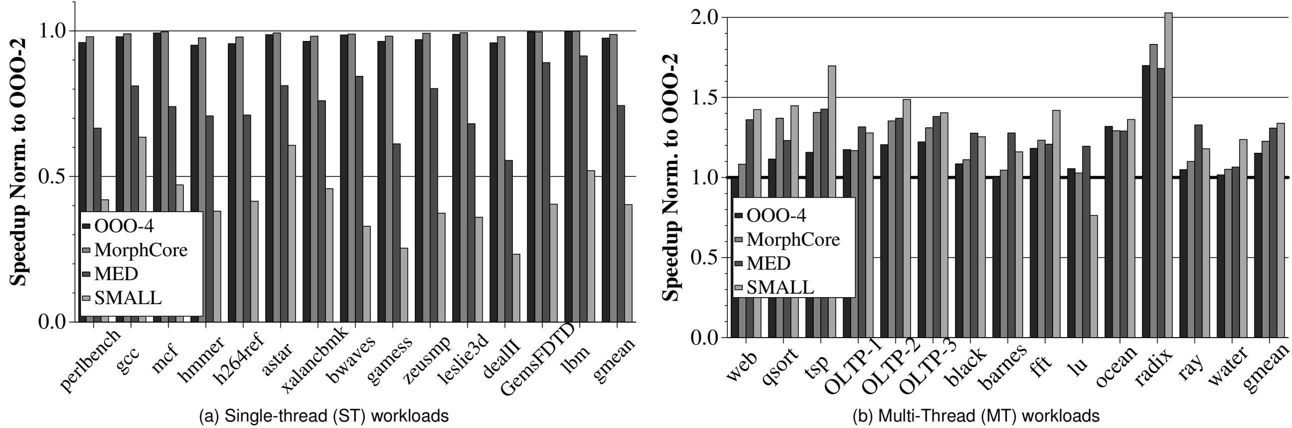


Figure 9: Performance results

to three MED cores. `qsort` is another workload with low uops/cycle (1.95), however MorphCore (similar to other throughput cores) does not provide as high a performance improvement as in the case of `radix`. This is because of two reasons: 1) when executing `qsort`, MorphCore does not spend a significant amount of time in InOrder mode (only 80% of execution time runs more than 2 threads active as shown in Figure 8), and 2) even when more than 2 threads are active, only 50% of the time are 6 or more threads active (data not shown in Figure 8). Thus, MorphCore does not get much opportunity to achieve higher throughput. Note that MorphCore still outperforms MED cores in `qsort` because of its ability to execute up to 8 threads.

Other workloads that have relatively high uops/cycle on OOO-2 (from 2.17 to 2.78) achieve relatively lower performance improvement with MorphCore over OOO-2 (from 23% for `fft` to 40% for `tsp`). The performance improvement of MorphCore is higher for `tsp` as compared to other workloads in this category even with a relatively high baseline uops/cycle of 2.78 (on OOO-2) because MorphCore ends up executing fewer number of total instructions (-10%) as compared to OOO-2 although doing the same algorithmic work. This is because `tsp` is a branch and bound algorithm, and the likelihood of quickly reaching the solution increases with more threads.

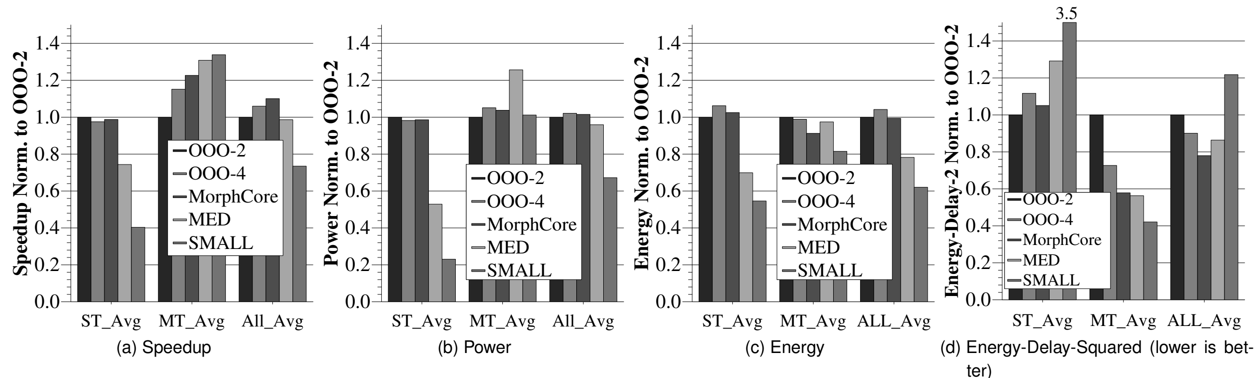
MorphCore provides the least performance improvement in workloads that can achieve a high uops/cycle (from 3.06 to 3.51) even when run with 2 threads on OOO-2 (`web`, `black`, `barnes`, `ray`, and `water`). These workloads have high per-thread ILP available, and thus do not benefit significantly from increasing the number of threads, because the performance improvement that can be achieved is limited by the peak throughput of MorphCore. However, as we later show, MorphCore is still beneficial because it is able to provide higher performance *at a lower energy consumption by executing SMT threads in-order*.

In general, MorphCore's performance improvement is lower than that of throughput optimized cores, MED and SMALL, over OOO-2 (on average 22% vs 30% and 33%) because of its lower peak MT throughput (Table 2). However, MorphCore outperforms MED cores in 3 workloads: `qsort`, `fft`, and `radix`. `qsort` and `radix` benefit from more threads as explained above. In `fft`, MED cores suffer from thread imbalance during the execution: 3 threads are active only for 72% of the execution time, and only 2 threads are active for 24% of execution time, and thus provide a slightly lower performance (-3%) than MorphCore. MorphCore also outperforms SMALL cores in `lu`. (In fact SMALL cores perform worse than OOO-2). This is because `lu`'s threads do not reach global barrier at the same time and have to wait for the lagging thread. Because SMALL cores have low single-thread performance, threads end up waiting for the lagging thread for a significant amount of time (only 1 thread is active for 35% of the execution time as shown in Figure 8), and thus execution time increases significantly. MorphCore does not suffer significantly from the problem of thread-imbalance-at-barrier because it switches into OutOfOrder mode when only 1 thread is active, therefore thread's waiting time is reduced.

MorphCore also outperforms OOO-4, a core architecture that has a higher area overhead and is significantly more complex than MorphCore (because OOO-4 supports 4 OOO SMT contexts), on average by 7% and up to 26% (for `qsort`). Although the peak throughput of both MorphCore and OOO-4 is the same (4, Table 2), MorphCore wins because it provides better latency tolerance by executing more threads than OOO-4. Thus, for workloads which have low uops/cycle and benefit from increasing the number of threads, MorphCore provides significantly higher MT performance compared to OOO-4.

6.1.3. Overall Performance Results. Figure 10a summarizes the average speedup of each architecture normalized to OOO-2. On single-thread (ST) workloads, MorphCore performs very close to OOO-2, the best ST-optimized architecture. On multi-thread (MT) workloads, MorphCore performs 22% higher than OOO-2, and achieves 2/3 of the performance potential of the best MT-optimized architectures (MED and SMALL). On average across all workloads (ST and MT), MorphCore outperforms all other architectures. We conclude that MorphCore is able to handle diverse ST and MT workloads efficiently.

6.1.4. Sensitivity of MorphCore's Results to Frequency Penalty. We find that for a MorphCore with an X% frequency penalty, performance of ST and MT workloads reduces by $\sim$ X/2% and X% respectively as compared to a MorphCore with no frequency penalty. This is because our ST workloads are core+memory bound while our MT workloads are primarily core bound.



(a) Speedup (b) Power (c) Energy (d) Energy-Delay-Squared (lower is better)

Figure 10: Speedup, Power, Energy, and Energy-Delay-Squared Results Summary

6.2. Energy-Efficiency Results

We first refer to Figure 10b that shows total power (static + dynamic) of each core configuration for ST and MT workloads. As expected, for ST workloads, the highest power configurations are large out-of-order cores (OOO-2, OOO-4, and MorphCore). A single MED or a single SMALL core takes area less than an out-of-order core, and thus toggle less capacitance, resulting in 47% and 77% lower power respectively. For MT workloads, all core configurations consume similar power (except MED cores), thus confirming that area-equivalent core comparisons result in power-equivalent core comparisons. The 3 MED cores take 18% more area (Table 2), and provide 30% higher performance (which translates into more dynamic power), resulting in 25% more power over OOO-2. Note that MorphCore consumes 2% less power than OOO-4 while providing 7% higher performance. This is because MorphCore does not waste energy on OOO renaming/scheduling, and instead, provides performance via highly-threaded in-order SMT execution.

Figure 10c shows the total (static + dynamic) energy consumed by each configuration (core includes L1 I and D caches but not L2 and L3 caches) normalized to OOO-2. As expected, SMALL cores are the most energy-efficient cores in both workload categories: for ST workloads, they have 59% lower performance for 77% lower power (an energy reduction of 46%), and for MT workloads they have 33% higher performance for 1% higher power (an energy reduction of 19%) than OOO-2. For MT workloads, MorphCore is the second best in energy-efficiency (after SMALL cores): MorphCore consumes 9%, 7%, and 6% less energy than OOO-2, OOO-4, and MED cores respectively.

MorphCore reduces energy consumption for two reasons: 1) MorphCore reduces execution time, thus keeping the core's structures active for shorter period of time, and 2) even when MorphCore is active, some of the structures that will be active in traditional out-of-order cores will be inactive in MorphCore's InOrder mode. These structures include the Rename logic, part of the instruction Scheduler, and the Load Queue. For reference, Table 5 shows the power for several key structures of OOO-2 core as a percentage of core power averaged across MT workloads. We find that 50% of the energy savings of MorphCore over OOO-2, and 75% of the energy savings of MorphCore over OOO-4 come from *reducing the activity of these structures*. MorphCore is also more energy-efficient than MED cores because even when it provides 8% lower performance, it does so at significantly (22%) lower power than MED cores, resulting an energy savings of 6% (70% of MorphCore's energy savings over MED cores happen because of reduced activity of MorphCore's structures in InOrder mode).

Table 5: Power of key structures of OOO-2

Structure	Power
Rename + RATs	4.9%
Scheduler	2.9%
Physical Register File	3.7%
Load + Store Queue	3.0%
ROB	2.1%

Figure 10d shows Energy-Delay-Squared (ED^2), a combined energy-performance efficiency metric [36, 21], of the five evaluated architectures. On average across all workloads, MorphCore provides the lowest ED^2: 22% lower than the baseline OOO-2, 13% lower than OOO-4, 9% lower than MED, and 44% lower than SMALL. We conclude that MorphCore provides a good balance between energy consumption and performance improvement in both ST and MT workloads.

7. Related Work

MorphCore is related to previous work in reconfigurable cores, heterogeneous chip multiprocessors, scalable cores, simultaneous multithreading, and power-efficient cores.

7.1. Reconfigurable Cores

Most closely related to our work are the numerous proposals that use reconfigurable cores to handle both latency- and throughput sensitive workloads [16, 5, 17, 25, 24, 32, 9, 10]. All these proposals share the *same fundamental idea*: build a chip with "simpler cores" and "combine" them using additional logic at runtime to form a high performance out-of-order core when high single thread performance is required. The cores operate independently in throughput mode.

TFlex [17], E2 dynamic multicore architecture [25], Bahurupi [24], and Core Genesis [10] require compiler analysis and/or ISA support for instruction steering to constituent cores to reduce the number of accesses to centralized structures. MorphCore does not require compiler/ISA support, and therefore can run legacy binaries without modification. Core Fusion [16], Federation Cores [5], Widget [32], and Forwardflow [9] provide scalability without any compiler/ISA support, similar to MorphCore.

Shortcomings. There are several shortcomings with the approach of combining simpler cores to form a large OOO core:
(1) Performance benefit of fusing the cores is limited because the constituent small cores operate in lock-step. Furthermore, fusing adds latencies among the pipeline stages of the fused core, and requires inter-core communication if dependent operations are steered to different cores.
(2) Switching modes incurs high overhead due to instruction cache flushes and data migration among the data caches of small cores.
(3) Core-Fusion-like proposals are not only in-efficient in "fused" mode, but also in their "non-fused" mode, because they use medium-size OOO cores as their base cores, which are power inefficient.

Comparison with CoreFusion. CoreFusion [16] fuses medium-sized OOO cores (2-wide, 48 entry OOO window) to form a large out-of-order core. Figure 11 shows the speedup of a single medium-sized OOO core (MED) and MorphCore normalized to CoreFusion for single-threaded workloads. In this experiment, CoreFusion combines three MED cores (see Table 2), and we use fusion latencies as described in [16] (7-cycle rename, 2-cycle extra branch misprediction, and 2-cycle inter-core communication penalties). We use the instruction steering heuristic described in the CoreFusion paper, and assume perfect LSQ bank prediction for steering loads/stores to cores. CoreFusion outperforms MED across all workloads except `mcf` (12% on average) because of its higher effective issue-width, window-size and L1 Dcache size. In `mcf`, these benefits are nullified by the overhead of inter-core communication introduced by CoreFusion. MorphCore outperforms both MED and CoreFusion across the board because unlike CoreFusion, it is a traditional aggressive out-of-order core without latency and communication overheads. On average MorphCore performs 17% better than CoreFusion.

Figure 12 shows the average speedup, power, energy, and ED^2 of MorphCore normalized to CoreFusion. On average, MorphCore provides 5% higher performance than CoreFusion. CoreFusion outperforms MorphCore in multi-threaded workloads (8% on average, per benchmark results are shown in Figure 9b) because it has a higher peak throughput as it consists of 3 medium-sized OOO cores. MorphCore reduces power (19%), energy (29%), and ED^2 (29%) when averaged across both single-threaded and multi-threaded workloads over CoreFusion because it uses less area (see Table 2), and thus, consumes less static power than CoreFusion's three MED cores while providing higher or comparable performance.

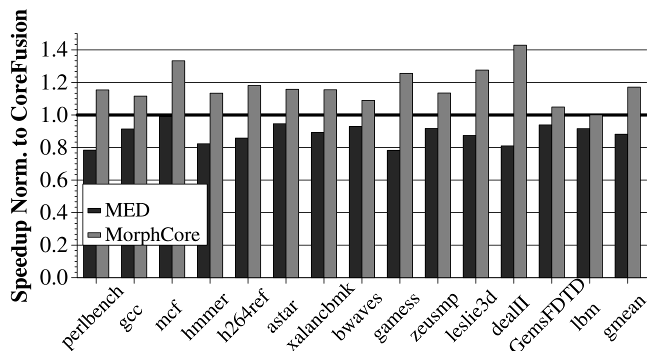


Figure 11: MorphCore's Single-Thread Performance versus CoreFusion

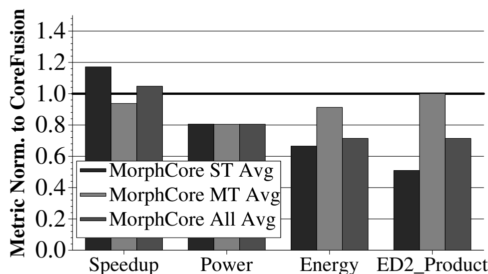


Figure 12: Average Speedup, Power, Energy, and ED^2 of MorphCore versus CoreFusion

7.2. Heterogeneous Chip-Multiprocessors

Heterogeneous (or Asymmetric) Chip Multiprocessors (ACMPs) [11, 22, 28] consist of one or a few large cores to accelerate single-threaded code, and many small cores to accelerate multi-threaded code. They have two limitations. First, the number of large and small cores is fixed at design time. In contrast, MorphCore can adapt the number of cores optimized for serial and parallel execution dynamically. Second, they incur a migration cost when execution is migrated between a small and a large core. Since MorphCore can accelerate threads "in-place," no migration overhead is incurred.

7.3. Scalable Cores

Scalable cores scale their performance and power consumption over a wide operating range. Dynamic Voltage and Frequency Scaling (DVFS) [14, 6] is a widely-used technique to scale a core's performance and power (e.g., Intel Turbo Boost [1]). However, increasing performance using DVFS costs significant increase in power consumption (power increases with the cube of frequency). Albonesi et al. [4] proposed dynamically tuning processor resources (e.g., cache size, register file size, issue queue entries etc.) in order to provide on-demand performance and energy savings. However, such techniques do not explore how these resources can be better used, and what other resources can be turned-off when TLP is available.

7.4. Simultaneous Multi-Threading

Simultaneous Multi-Threading (SMT) [13, 35, 31] was proposed to improve resource utilization by executing multiple threads on the same core. However, unlike MorphCore, previously proposed SMT techniques *do not reduce power consumption* when TLP is available. Furthermore, traditional SMT increases the area/complexity and power consumption of the core, whereas MorphCore leverages existing structures and does not increase area/complexity and power. Hily and Seznec observed in [12] that out-of-order execution becomes unnecessary when thread-level parallelism is available. In contrast, MorphCore saves energy and improves performance when executing multi-threaded workloads.

7.5. Power-Efficient Cores

Braids [30] provides OOO-like single-thread performance using in-order resources. However, Braids does not adapt to the software because it *requires* complicated compiler/software effort upfront. In contrast, MorphCore requires no software effort, and adapts to the software's needs.

8. Conclusion

We propose the MorphCore architecture which is designed from the ground-up to improve the performance and energy-efficiency of both single-threaded and multi-threaded programs. MorphCore operates as a high-performance out-of-order core when Thread-Level Parallelism is low, and as a high-performance low-energy, highly-threaded in-order SMT core when Thread-Level Parallelism is high. Our evaluation with 14 single-threaded and 14 multi-threaded workloads shows that MorphCore increases performance by 10% and reduces energy-delay-squared product (ED^2) by 22% over a typical 2-way SMT out-of-order core. We also show that MorphCore increases performance and reduces ED^2 when compared to an aggressive 4-way SMT out-of-order core, medium out-of-order cores, and small in-order cores. It also outperforms CoreFusion, a reconfigurable core architecture, in terms of performance (by 5%), and ED^2 (by 29%). We therefore suggest MorphCore as a promising direction for increasing performance, saving energy, and accommodating workload diversity while requiring minimal changes to a traditional out-of-order core. In the future we plan to further enhance MorphCore by exploring better policies for switching between in-order and out-of-order mode and by providing hardware mechanisms to support a low-power in-order single-thread mode.

Acknowledgments

We thank José Joao, other members of the HPS research group, Carlos Villavieja, our shepherd Scott Mahlke, and the anonymous reviewers for their comments and suggestions. Special thanks to Nikhil Patil, Doug Carmean, Rob Chappell, and Onur Mutlu for helpful technical discussions. We gratefully acknowledge the support of the Cockrell Foundation and Intel Corporation. Khubaib was supported by an Intel PhD Fellowship.

References

[1] "Intel Turbo Boost Technology," Intel Corporation, http://www.intel.com/technology/turboboost/index.htm.

[2] "MySQL database engine 5.0.1," http://www.mysql.com.

[3] "SysBench: a system performance benchmark v0.4.8," http://sysbench.sourceforge.net.

[4] D. H. Albonesi *et al.*, "Dynamically tuning processor resources with adaptive processing," *IEEE Computer*, 2003.

[5] M. Boyer, D. Tarjan, and K. Skadron, "Federation: Boosting per-thread performance of throughput-oriented manycore architectures," *ACM Trans. Archit. Code Optim. (TACO)*, 2010.

[6] T. Burd and R. Brodersen, "Energy efficient CMOS microprocessor design," in *Proceedings of the Twenty-Eighth Hawaii International Conference on System Sciences*, 1995.

[7] Z. Chishti and T. N. Vijaykumar, "Optimal power/performance pipeline depth for SMT in scaled technologies," *IEEE Trans. on Computers*, Jan. 2008.

[8] A. J. Dorta *et al.*, "The OpenMP source code repository," in *Euromicro*, 2005.

[9] D. Gibson and D. A. Wood, "Forwardflow: a scalable core for power-constrained CMPs," in *ISCA*, 2010.

[10] S. Gupta, S. Feng, A. Ansari, and S. Mahlke, "Erasing core boundaries for robust and configurable performance," in *MICRO*, 2010.

[11] M. D. Hill and M. R. Marty, "Amdahl's law in Multicore Era," Univ. of Wisconsin, Tech. Rep. CS-TR-2007-1593, 2007.

[12] S. Hily and A. Seznec, "Out-of-order execution may not be cost-effective on processors featuring simultaneous multithreading," in *HPCA*, 1999.

[13] H. Hirata *et al.*, "An elementary processor architecture with simultaneous instruction issuing from multiple threads," in *ISCA*, 1992.

[14] M. Horowitz *et al.*, "Low-power digital design," in *IEEE Symposium on Low Power Electronics*, 1994.

[15] Intel, "Intel 64 and IA-32 Architectures Software Dev. Manual, Vol-1," 2011.

[16] E. Ipek *et al.*, "Core fusion: accommodating software diversity in chip multiprocessors," in *ISCA-34*, 2007.

[17] C. Kim *et al.*, "Composable lightweight processors," in *MICRO-40*, 2007.

[18] D. Koufaty and D. Marr, "Hyperthreading technology in the Netburst microarchitecture," *IEEE Micro*, 2003.

[19] H. Kredel, "Source code for traveling salesman problem (tsp)," http://krum.rz.uni-mannheim.de/ba-pp-2007/java/index.html.

[20] S. Li *et al.*, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO 42*, 2009.

[21] A. J. Martin, *et al.*, *Power aware computing*. Kluwer Academic Publishers, 2002, ch. ET2: a metric for time and energy efficiency of computation.

[22] T. Y. Morad *et al.*, "Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors," 2006.

[23] NVIDIA Corporation, "CUDA SDK code samples," 2009.

[24] M. Pricopi and T. Mitra, "Bahurupi: A polymorphic heterogeneous multi-core architecture," *ACM TACO*, January 2012.

[25] A. Putnam *et al.*, "Dynamic vectorization in the E2 dynamic multicore architecture," *SIGARCH Comp. Arch. News*, 2011.

[26] D. Sager, D. P. Group, and I. Corp, "The microarchitecture of the pentium 4 processor," *Intel Technology Journal*, vol. 1, p. 2001, 2001.

[27] J. Stark *et al.*, "On pipelining dynamic instruction scheduling logic," in *MICRO-33*, 2000.

[28] M. A. Suleman *et al.*, "Accelerating critical section execution with asymmetric multi-core architectures," in *ASPLOS*, 2009.

[29] Tornado Web Server, "Source code," http://tornado.sourceforge.net/, 2008.

[30] F. Tseng and Y. N. Patt, "Achieving out-of-order performance with almost in-order complexity," in *ISCA*, 2008.

[31] D. M. Tullsen *et al.*, "Simultaneous multithreading: Maximizing on-chip parallelism," in *ISCA-22*, 1995.

[32] Y. Watanabe *et al.*, "Widget: Wisconsin decoupled grid execution tiles," in *ISCA*, 2010.

[33] C. Wilkerson *et al.*, "Trading off cache capacity for reliability to enable low voltage operation," in *ISCA*, 2008.

[34] S. C. Woo *et al.*, "The SPLASH-2 programs: Characterization and methodological considerations," in *ISCA-22*, 1995.

[35] W. Yamamoto *et al.*, "Performance estimation of multistreamed, superscalar processors," in *Hawaii Intl. Conf. on System Sciences*, 1994.

[36] V. Zyuban *et al.*, "Integrated analysis of power and performance for pipelined microprocessors," *IEEE Transactions on Computers*, 2004.