# Core Fusion: Accommodating Software Diversity in Chip Multiprocessors

Engin İpek, Meyrem Kırman, Nevin Kırman, and José F. Martínez

Computer Systems Laboratory
Cornell University
Ithaca, NY, USA

http://m3.csl.cornell.edu/

## ABSTRACT

This paper presents *core fusion*, a reconfigurable chip multiprocessor (CMP) architecture where groups of fundamentally independent cores can dynamically morph into a larger CPU, or they can be used as distinct processing elements, as needed at run time by applications. Core fusion gracefully accommodates software diversity and incremental parallelization in CMPs. It provides a single execution model across all configurations, requires no additional programming effort or specialized compiler support, maintains ISA compatibility, and leverages mature micro-architecture technology.

## Categories and Subject Descriptors

C.1.3 Computer Systems Organization [**Processor Architectures**]: Adaptable Architectures; C.1.4 Computer Systems Organization [**Processor Architectures**]: Parallel Architectures

## General Terms

Performance, Design

## Keywords

Chip Multiprocessors, Reconfigurable Architectures, Software Diversity

## 1. INTRODUCTION

Chip multiprocessors (CMPs) hold the prospect of translating Moore's Law into sustained performance growth by incorporating more and more cores on the die. In the short term, on-chip integration of a modest number of relatively powerful cores may yield high utilization when running multiple sequential workloads. However, although sequential codes are likely to remain important, they alone are not sufficient to sustain long-term performance scalability. Consequently, harnessing the full potential of CMPs in the long term makes the widespread adoption of parallel programming inevitable.


*ISCA'07,* June 9–13, 2007, San Diego, California, USA.


Unfortunately, code parallelization constitutes a tedious, time-consuming, and error-prone effort. Historically, programmers have parallelized code incrementally to amortize programming effort over time. Typically, the most promising loops or regions in a sequential execution of the program are identified through profiling. A subset of these regions is then parallelized. Over time, more effort is spent on the remaining code. Already popular programming models (e.g., OpenMP [12]) are designed to facilitate incremental parallelization.

As CMPs become ubiquitous, we envision a dynamic and diverse landscape of software products of very different characteristics and in different stages of development: from purely sequential, to highly parallel, and everything in between. Moreover, as a result of incremental parallelization, applications will exert very different demands on the hardware across phases of the same run (e.g., sequential vs. highly parallel code sections within the same program). This diversity is fundamentally at odds with most CMP designs, whose composition is "set in stone" by the time they are fabricated.

Asymmetric chip multiprocessors (ACMPs) [3, 23, 24] comprise cores of varying sizes and computational capabilities. The hope is to match the demands of a variety of sequential and parallel software. Still, the particular die composition is set at design time. Ultimately, this may constitute a hurdle to high performance. For example, Balakrishnan et al. [3] find that asymmetry generally hurts parallel application scalability, and renders the applications' performance less predictable, unless relatively sophisticated software changes are introduced. Hence, for example, while an ACMP may deliver increased performance on sequential codes by placing one large core on the die, it may do so at the expense of parallel performance or programmability.

Instead, we would like a CMP to provide the flexibility to dynamically "synthesize" the right composition, based on software demands. In this paper, we investigate a novel reconfigurable hardware mechanism that we call *core fusion*. It is an architectural technique that empowers groups of relatively simple and fundamentally independent CMP cores with the ability to "fuse" into one large CPU on demand. We envision a core fusion CMP as a homogeneous substrate with conventional memory coherence/consistency support that is optimized for parallel execution, but where groups of up to four adjacent cores and their i- and d-caches can be fused at run-time into CPUs that have up to four times the fetch, issue, and commit width, and up to four times the i-cache, d-cache, branch predictor, and BTB size.

Core fusion has the potential to provide a number of highly desirable benefits to CMP design and functionality. Among them:

- *Support for software diversity.* CMPs may be configured for fine-grain parallelism (by providing many lean cores), coarse-grain parallelism (by fusing many cores

into fewer, but more powerful CPUs), sequential code (by executing on one fused group), and different levels of multiprogramming (by providing as many fused groups as needed, up to capacity). In contrast, for example, ACMPs are "stuck" with the mix chosen at design time, which may compromise performance for parallel codes and/or mismatched multiprogrammed workloads.

- *Support for smoother software evolution.* Core fusion would naturally support incremental parallelization, by *dynamically* providing the optimal configuration for sequential and parallel regions of a particular code, e.g., one large fused group during sequential regions, and many small independent cores during parallel regions.
- *Single-design solution.* A fusion group is essentially a modular structure comprising four identical cores, plus the core fusion fabric. Core fusion CMPs can be designed by tiling as many such groups as desired. In contrast, for example, ACMPs require the adoption of at least two processor core designs.
- *Optimized for parallel code.* Core fusion comprises relatively small and fundamentally independent cores. This provides good isolation across threads in parallel runs, both internally (branch predictor, i- and d-TLB, physical registers, etc.) and at the L1 cache level (i- and d-cache). The core fusion support allows cores to work co-operatively when needed (albeit probably at somewhat lower performance than a large, monolithic processor). In contrast, techniques like simultaneous multithreading (SMT) take the opposite approach: A large wide-issue core that is optimized for sequential execution, augmented with support for multiple threads to increase utilization. When executing parallel applications, cross-thread interference in SMT designs is an obstacle to high performance. In a software landscape where parallel code is expected to be increasingly more prevalent, a "bottom-up" approach like core fusion may be preferable. (Moreover, SMT support can be added to core fusion's base cores.)
- *Design-bug and hard-fault resilience.* A design bug or hard fault in the core fusion hardware need not disable an entire four-core fusion group, as each core may still be able to operate independently. Similarly, a hard fault in one core still allows independent operation of the three fault-free cores, and even two-way fusion on the other two cores in the fusion group. Bug/hard fault isolation may be significantly more challenging in designs based on large cores. (The mechanisms that would be needed for detection, isolation, and recovery are out of the scope of this paper.)

At the same time, providing CMPs with the ability to "fuse" cores on demand presents significant design challenges. Among them:

- Core fusion should not increase software complexity significantly. Specifically, cores should be able to execute programs co-operatively without changing the execution model, and without resorting to custom ISAs or specialized compiler support. This alone would set core fusion apart from other proposed reconfigurable architectures, such as TRIPS [37] or Smart Memories [28], and from speculative architectures such as Multiscalar [38]. (Section 6 conducts a review of this and other related work.)
- Core fusion hardware should work around the fundamentally independent nature of the base cores. This means providing complexity-effective solutions to collective fetch, rename, execution, cache access and commit, by leveraging each core's existing structures without unduly overprovisioning or significantly restructuring the base cores.
- Dynamic reconfiguration should be efficient, and each core's hardware structures should work fundamentally the same way regardless of the configuration.

This paper presents, for the first time, a detailed description of a complete hardware solution to support adaptive core fusion in CMPs. In the course of formulating our core fusion solution, this paper makes the following additional contributions over prior art:

- A reconfigurable, distributed front-end and instruction cache organization that can leverage individual cores' front-end structures to feed an aggressive fused back-end, with minimal over-provisioning of individual front-ends.
- A complexity-effective remote wake-up mechanism that allows operand communication across cores without requiring additional register file ports, wake-up buses, bypass paths, or issue queue ports.
- A reconfigurable, distributed load/store queue and data cache organization that *(a)* leverages the individual cores' data caches and load/store queues in all configurations; *(b)* does not cause thread interference in L1 caches when cores run independently; *(c)* supports conventional coherence when running parallel code, generates zero coherence traffic within the fusion group when running sequential code in fused mode, and requires minimal changes to each core's CMP subsystem; *(d)* guarantees correctness without requiring data cache flushes upon runtime configuration changes; and *(e)* enforces memory consistency in both modes.
- A reconfigurable, distributed ROB organization that can fully leverage individual cores' ROBs to seamlessly support fusion, without overprovisioning or unnecessarily replicating core ROB structures.
- A quantitative assessment of the incremental parallelization process on CMPs.

Our evaluation pits core fusion against more traditional CMP architectures, such as fine- and coarse-grain homogeneous cores, as well as ACMPs, and shows that core fusion's flexibility and run-time reconfigurability make it an attractive CMP architecture to support a diverse, evolving software landscape.

# 2. ARCHITECTURE

Core fusion builds on top of a substrate comprising identical, relatively efficient two-issue out-of-order cores. A bus connects private L1 i- and d-caches and provides data coherence. On-chip L2 cache and memory controller reside on the other side of this bus. Cores can execute fully independently if desired. It is also possible to fuse groups of two or four cores to constitute larger cores. Figure 1 is an illustrative example of a CMP comprising eight two-issue cores with core fusion capability. The figure shows an (arbitrarily chosen) asymmetric configuration comprising one eight-issue, one four-issue, and two two-issue processors.

We now describe in detail the core fusion support. In the discussion, we assume four-way fusion.

## 2.1 Front-end

### 2.1.1 Collective Fetch

A small co-ordinating unit called the *fetch management unit* (FMU) facilitates collective fetch. The FMU receives and re-sends relevant fetch information across cores. The total latency from a core into the FMU and out to any other core is two cycles (Section 4).

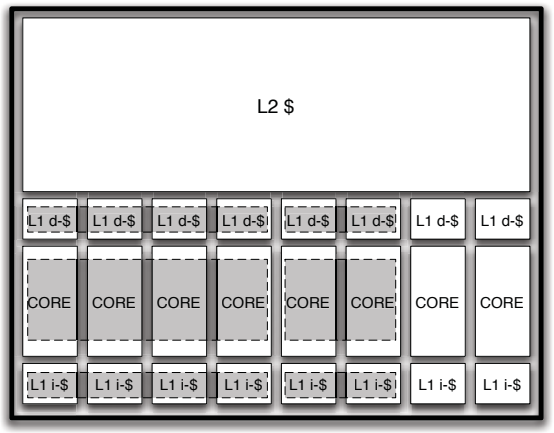


**Figure 1:** **Conceptual floorplan of an eight-core CMP with core fusion capability. The figure shows a configuration example comprising two independent cores, a two-core fused group, and a four-core fused group. The figure is not meant to represent an actual floorplan.**

## *Fetch Mechanism and Instruction Cache*

Each core fetches two instructions from its own i-cache every cycle, for a total of eight instructions. Fetch is aligned, with core zero generally responsible for the oldest two instructions. On a taken branch (or misprediction recovery), however, the target may not be aligned with core zero. In that case, lower-order cores skip fetch, and core-zero-aligned fetch resumes on the next cycle.

On an i-cache miss, an eight-word block is *(a)* delivered to the requesting core if it is operating independently, or *(b)* distributed across all four cores in a fused configuration to permit collective fetch. To support these two options, we make i-caches reconfigurable along the lines of earlier works [28]. Each i-cache has enough tags to organize its data in two-word subblocks. When running independently, four such subblocks and one tag make up a cache block. When fused, cache blocks span all four i-caches, with each i-cache holding one subblock and a replica of the cache block's tag. (How to dynamically switch from one i-cache mode to the other is explained later in Section 3.) Figure 2 shows an example of i-cache organization in a fusion group.

During collective fetch, it makes sense to replicate the i-TLB across all cores in a fused configuration. Notice that this would be accomplished "naturally" as cores miss on their i-TLBs, however taking multiple i-TLB misses for a single eight-instruction block is unnecessary, since the FMU can be used to refill all i-TLBs upon a first i-TLB miss by a core. The FMU is used to gang-invalidate i-TLB entries.

## *Branches and Subroutine Calls*

**Prediction.** During collective fetch, each core accesses its own branch predictor and BTB. Because collective fetch is aligned, each branch instruction always accesses the same branch predictor and BTB. Consequently, the effective branch predictor and BTB capacity is four times as large. To accomplish maximum utilization while retaining simplicity, branch predictor and BTB are indexed as shown in Figure 4 regardless of the configuration. We empirically observe no loss in prediction accuracy when using this "configuration-oblivious" indexing scheme. Notice that branch predictor and BTB entries remain meaningful across configurations as a result of this indexing scheme.

Each core can handle up to one branch prediction per cycle. PC redirection (predict-taken, mispredictions) is enabled by the FMU. Each cycle, every core that predicts a taken branch, as well as every core that detects a branch

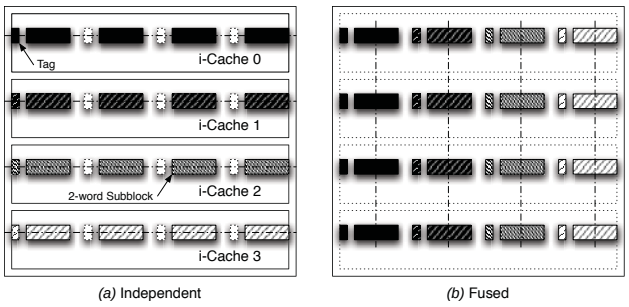


**Figure 2:** **Illustrative example of four i-caches organized *(a)* independently or *(b)* fused. In independent mode, four subblocks and one tag within each i-cache constitute a cache block. In fused mode, a cache block spans four i-caches, each i-cache being responsible for a subblock and a tag replica.**

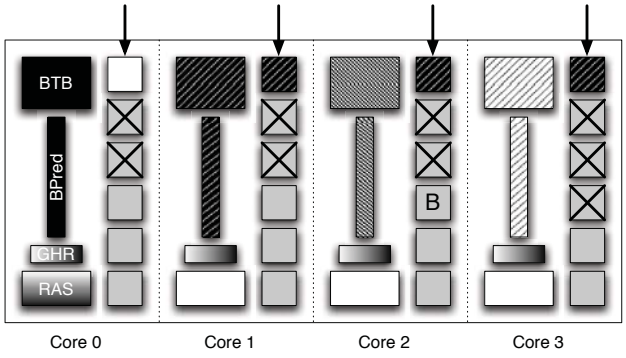


**Figure 3:** **Example of aligned fetch in fused mode. In the figure, cores squash overfetched instructions as they receive a predict-taken notice from Core 2 with a two-cycle delay. The new target starts at Core 1, and thus Core 0 skips the first fetch cycle. Notice the banked branch predictor and BTB, the replicated GHR, and the Core-0-managed RAS.**

misprediction, sends the new target PC to the FMU. The FMU selects the correct PC by giving priority to the oldest misprediction-redirect PC first, and the youngest branch-prediction PC last, and sends the selected PC to all fetch units. Once the transfer of the new PC is complete, cores use it to fetch from their own i-cache as explained above.

Naturally, on a misprediction, misspeculated instructions are squashed in all cores. This is also the case for instructions "overfetched" along the not-taken path on a taken branch, since the target PC will arrive with a delay of a few cycles. In Figure 3, Core 2 predicts branch B to be taken. After two cycles, all cores receive this prediction. They squash overfetched instructions, and adjust their PC. In the example, the target lands on Core 1, which makes Core 0 skip the initial fetch cycle.

**Global History.** Because each core is responsible for a subset of the branches in the program, having independent and unco-ordinated history registers on each core may make it impossible for the branch predictor to learn of their correlation. To avert this situation, the GHR can be simply replicated across all cores, and updates be co-ordinated through the FMU. Specifically, upon every branch prediction, each core communicates its prediction–whether taken or not taken–to the FMU. Additionally, as discussed, the FMU receives nonspeculative updates from every back-end upon branch mispredictions. The FMU communicates such events to each core, which in turn update their GHR. Upon nonspeculative updates, earlier (checkpointed) GHR contents are recovered on each core. The fix-up mechanism employed to checkpoint and recover GHR contents can be along

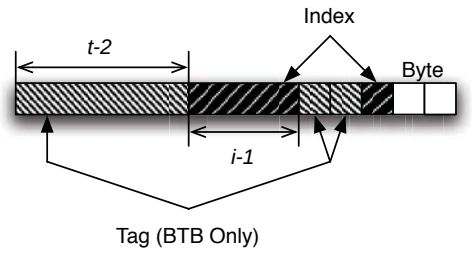


**Figure 4: Configuration-oblivious indexing utilized in branch prediction and BTB. In the figure, $i$ bits are used for indexing and $t$ for tagging (tagging only meaningful in the BTB). Of course, $i$ and $t$ are generally not the same for branch predictor and BTB. Because of aligned fetch, the two tag bits sandwiched between index bits match the core number in the fused configuration.**

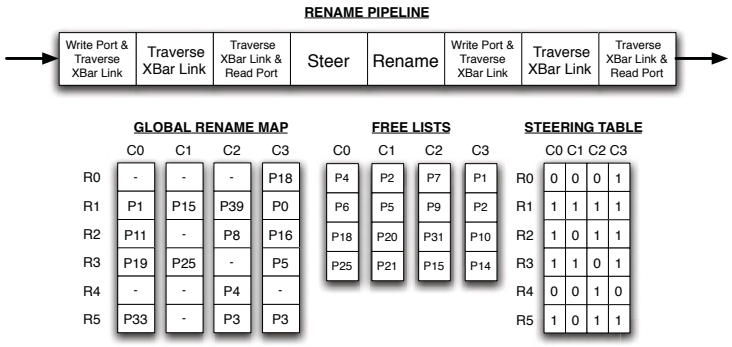


**Figure 5: Rename pipeline (top) and illustrative example of SMU organization (bottom). R0 has a valid mapping in core three, whereas R1 has four valid mappings (one in each core). Only six architectural registers are shown.**

the lines of the outstanding branch queue (OBQ) mechanism in the Alpha 21264 microprocessor [21].

**Return Address Stack.** As the target PC of a subroutine call is sent to all cores by the FMU (which flags the fact that it is a subroutine call), core zero pushes the return address into its RAS. When a return instruction is encountered (possibly by a different core from the one that fetched the subroutine call) and communicated to the FMU, core zero pops its RAS and communicates the return address back through the FMU. Notice that, since all RAS operations are processed by core zero, the effective RAS size does not increase when cores are fused. This is reasonable, however, as call depth is a program property that is independent of whether execution is taking place on an independent core or on a fused configuration.

### *Handling Fetch Stalls*

On a fetch stall by one core (e.g., i-cache miss, i-TLB miss, fetching two branches), all fetch engines must also stall so that correct fetch alignment is preserved. To accomplish this, cores communicate stalls to the FMU, which in turn informs the other cores. Because of the latency through the FMU, it is possible that the other cores may overfetch, for example if *(a)* on an i-cache or i-TLB miss, one of the other cores does hit in its i-cache or i-TLB (unlikely in practice, given how fused cores fetch), or *(b)* generally in the case of two back-to-back branches fetched by the same core that contend for the predictor (itself exceedingly unlikely). Fortunately, the FMU latency is deterministic: Once all cores have been informed (including the delinquent core) they all discard at the same time any overfetched instruction (similarly to the handling of a taken branch before) and resume fetching in sync from the right PC—as if all fetch engines had synchronized through a "fetch barrier."

### *2.1.2 Collective Decode/Rename*

After fetch, each core pre-decodes its instructions independently. Subsequently, all instructions in the fetch group need to be renamed and steered. (As in clustered architectures, steering consumers to the same core as their producers can improve performance by eliminating communication delays.) Renaming and steering is achieved through a *steering management unit* (SMU). The SMU consists of: a global *steering table* to track the mapping of architectural registers to any core; four free-lists for register allocation (one for each core); four rename maps; and steering/renaming logic (Figure 5). The steering table and the four rename maps together allow up to four valid mappings of each architectural register, and enable operands to be replicated across multiple cores. Cores still retain their individual renaming structures, but these are bypassed when cores are fused.

Figure 5 depicts the high level organization of the rename pipeline. After pre-decode, each core sends up to two instructions to the SMU through a set of links. In our evaluation, we assume a three-cycle communication delay to the SMU over a repeated link (Section 4). Three cycles after pre-decode, the SMU receives up to two instructions and six architectural register specifiers (three per instruction) from each core. After renaming and steering, it uses a second set of links to dispatch no more than six physical register specifiers, two program instructions, and two copy instructions to each core. (Copy instructions have a separate, dedicated queue in each core (Section 2.2.1).) Restricting the SMU dispatch bandwidth in this way keeps the wiring overhead manageable, lowers the number of required rename map ports, and also helps achieve load balancing. In our evaluation (Section 5), we accurately model the latency of the eight-stage rename pipeline when running in fused mode, as well as the SMU dispatch bandwidth restrictions.

The SMU uses the incoming architectural register specifiers and the steering table to steer up to eight instructions every pipeline cycle. Each instruction is assigned to one of the cores via a modified version of dependence based steering [32] that guarantees that each core is assigned no more than two instructions. Copy instructions are also created in this cycle.

In the next cycle, instructions are renamed. Since each core receives no more than two instructions and two copy instructions, each rename map has only six read and six write ports. The steering table requires sixteen read and sixteen write ports (note that each steering table entry contains only a single bit, and thus the overhead of multi-porting this small table is relatively low). If a copy instruction cannot be sent to a core due to bandwidth restrictions, renaming stops at the offending instruction that cycle, and starts with the same instruction next cycle, thereby draining crossbar links and guaranteeing forward progress.

As in existing microprocessors, at commit time, any instruction that renames an architectural register releases the physical register holding the prior value (now obsolete). This is accomplished in core fusion easily, by having each ROB send the register specifiers of committing instructions to the SMU. Register replicas, on the other hand, can be disposed of more aggressively, provided there is no pending consumer instruction in the same core. (Notice that the "true" copy is readily available in another core.) We employ a well-known mechanism based on pending consumer counts [29, 30]. Naturally, the counters must be backed up on every branch prediction. Luckily, in core fusion these are small: four bits suffice to cover a core's entire instruction window (16 entries in our evaluation).

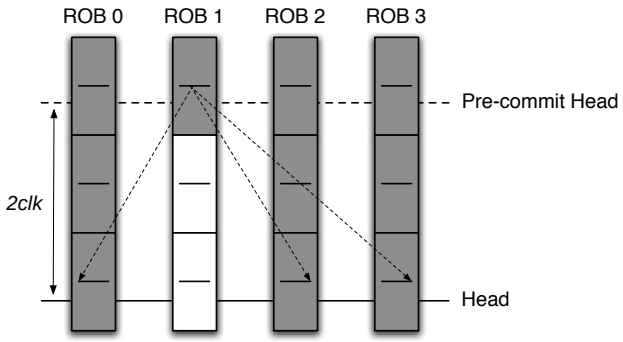


**Figure 6: Simplified diagram of core fusion's distributed ROB. In the figure, ROB 1's head instruction pair is not ready to commit, which is communicated to the other ROBs. Pre-commit and conventional heads are spaced so that the message arrives just in time (2 clock cycles in the example). Upon completion of ROB 1's head instruction pair, a similar message is propagated, again arriving just in time to retire all four head instruction pairs in sync.**

## 2.2 Back-end

Each core's back-end is essentially quite typical: separate floating-point and integer issue queues, a physical register file, functional units, load/store queues, and a ROB. Each core has a private L1 d-cache. L1 d-caches are connected via a split-transaction bus and are kept coherent via a MESI-based protocol. When cores get fused, back-end structures are co-ordinated to form a large virtual back-end capable of consuming instructions at a rate of eight instructions per cycle.

### 2.2.1 Collective Execution

#### Operand Crossbar

To support operand communication, a copy-out and a copy-in queue are added to each core. Copy instructions wait in the copy-out queue for their operands to become available, and once issued, they transfer their source operand and destination physical register specifier to a remote core. The operand crossbar is capable of supporting two copy instructions per core, per cycle. In addition to copy instructions, loads use the operand crossbar to deliver values to their destination register (Section 2.2.2). In our evaluation (Section 5), we accurately model latency and contention for the operand crossbar, and quantify its impact on performance.

#### Wake-up and Selection

When copy instructions reach the consumer core, they are placed in a FIFO copy-in queue. Each cycle, the scheduler considers the two copy instructions at the head, along with the instructions in the conventional issue queue. Once issued, copies wake up their dependent instructions and update the physical register file, just as regular instructions do.

#### Reorder Buffer and Commit Support

Fused in-order retirement requires co-ordinating four ROBs to commit in lockstep up to eight instructions per cycle. Instructions allocate ROB entries locally at the end of fetch. If the fetch group contains less than eight instructions, NOPs are allocated at the appropriate cores to guarantee alignment (Section 5.1.1 quantifies the impact that these "ROB bubbles" have on performance). Of course, on a pipeline bubble, no ROB entries are allocated.

When commit is not blocked, each core commits two instructions from the oldest fetch group every cycle. When one of the ROBs is blocked, all other cores must also stop committing on time to ensure that fetch blocks are committed atomically in order. This is accomplished by exchanging stall/resume signals across ROBs. To accommodate the inevitable (but deterministic) communication delay, each ROB is extended with a *pre-commit head pointer* in addition to the conventional head and tail pointers (Figure 6). Instructions always pass through the pre-commit head before they reach the actual ROB head and commit. Instructions that are not ready to commit by the time they reach the pre-commit head stall immediately, and send a "stall" signal to all other cores. Later, as they become ready, they move past the pre-commit head, and send a "resume" signal to the other cores. The number of ROB entries between the pre-commit head pointer and the actual head pointer is enough to cover the communication latency across cores. This guarantees that ROB stall/resume always take effect in a timely manner, enabling lockstep in-order commit. In our experiments (Section 5), we set the communication latency to two cycles, and consequently the actual head is separated from the pre-commit head by four instruction slots on each core at all times.

### 2.2.2 Load/Store Queue Organization

Our scheme for handling loads and stores is conceptually similar to clustered architectures [4, 10, 19, 26, 41]. However, while most proposals in clustered architectures choose a centralized L1 data cache or distribute it based on bank assignment, we keep the private nature of L1 caches, requiring only minimal modifications to the CMP cache subsystem.

Instead, in fused mode, we adopt a banked-by-address load-store queue (LSQ) implementation. This allows us to keep data coherent without requiring cache flushes after dynamic reconfiguration, and to support elegantly store forwarding and speculative loads. The core that issues each load/store to the memory system is determined based on effective addresses. The two bits that follow the block offset in the effective address are used as the LSQ bank-ID to select one of the four cores, and enough index bits to cover the L1 cache are allocated from the remaining bits. The rest of the effective address and the bank-ID are stored as a tag. Making the bank-ID bits part of the tag is important to properly disambiguate cache lines regardless of the configuration.

Effective addresses for loads and stores are generally not known at the time they are renamed. This raises a problem, since at rename time memory operations need to allocate LSQ entries from the core that will eventually issue them to the memory system. We attack this problem through LSQ bank prediction [4, 6]. Upon pre-decoding loads and stores, each core accesses its bank predictor by using the lower bits of the load/store PC. Bank predictions are sent to the SMU, and the SMU steers each load and store to the predicted core. Each core allocates load queue entries for the loads it receives. On stores, the SMU also signals all cores to allocate dummy store queue entries regardless of the bank prediction. Dummy store queue entries guarantee in-order commit for store instructions by reserving place-holders across all banks for store bank mispredictions. Upon effective address calculation, remote cores with superfluous store queue dummies are signaled to discard their entries (recycling these entries requires a collapsing LSQ implementation). If a bank misprediction is detected, the store is sent to the correct queue. Of course, these messages incur delays, which we model accurately in our experiments.

In the case of loads, if a bank misprediction is detected, the load queue entry is recycled (LSQ collapse) and the load is sent to the correct core. There, it allocates a load queue entry and resolves its memory dependences locally. Notice that, as a consequence of bank mispredictions, loads can allocate entries in the load queues out of program order. Fortunately, this is not a problem, because load queue entries are typically tagged by instruction age. However, there is a danger of deadlock in cases where the mispredicted load is older than all other loads in its (correct) bank and the load queue is full at the time the load arrives at the consumer

core. To prevent this situation, loads search the load queue for older instructions when they cannot allocate entries. If no such entry is found, a replay trap is taken, and the load is steered to the right core. Otherwise, the load is buffered until a free load queue entry becomes available.

Address banking of the LSQ also facilitates load speculation and store forwarding. Since any load instruction is free of bank mispredictions at the time it issues to the memory system, loads and stores to the same address are guaranteed to be processed by the same core.

Moreover, because fetch is aligned in all cases, we can easily leverage per-core load wait tables (LWT) [21] along the lines of the Alpha 21264. At the time a load is fetched, if the load's LWT entry bit is set, the load will be forced to wait until all older stores in its (final) core have executed (and all older dummy store queue entries in that core have been dealt with).[1]

When running parallel applications, memory consistency must be enforced regardless of the configuration. We assume relaxed consistency models where special primitives like memory fences (weak consistency) or acquire/release operations (release consistency) enforce ordering constraints on ordinary memory operations. Without loss of generality, we discuss the operation of memory fences below. Acquire and release operations are handled similarly.

For the correct functioning of synchronization primitives in fused mode, fences must be made visible to all load/store queues. We achieve this by dispatching these operations to all the queues, but having only the copy in the correct queue perform the actual synchronization operation. The fence is considered complete once each one of the local fences completes locally and all memory operations preceding each fence commit. Local fence completion is signaled to all cores through a one-bit interface in the portion of the operand crossbar that connects the load-store queues.

# 3. DYNAMIC RECONFIGURATION

Our discussion thus far explains the operation of the cores in a static fashion. This alone may improve performance significantly, by choosing the CMP configuration most suitable for a particular workload. However, support for dynamic reconfiguration to respond to software changes (e.g., dynamic multiprogrammed environments or serial/parallel regions in a partially parallelized application) can greatly improve versatility, and thus performance.

In general, we envision run-time reconfiguration enabled through a simple application interface. The application requests core fusion/split actions through a pair of *FUSE* and *SPLIT* ISA instructions, respectively. In most cases, these requests can be readily encapsulated in conventional parallelizing macros or directives. FUSE and SPLIT instructions are executed conditionally by hardware, based on the value of an OS-visible control register that indicates which cores within a fusion group are eligible for fusion. To enable core fusion, the OS allocates either two or four of the cores in a fusion group to the application when the application is context-switched in, and annotates the group's control register. If, at the time of a FUSE request, fusion is not possible (e.g., in cases where another application is running on the other cores), the request is simply ignored. This is possible because core fusion provides the same execution model regardless of the configuration.

We now explain FUSE and SPLIT operations in the context of alternating serial/parallel regions of a partially parallelized application that follows a fork/join model (typical of OpenMP). Other uses of these or other primitives (possibly involving OS scheduling decisions) are left for future work.

**FUSE operation.** After completion of a parallel region, the application may request cores to be fused to execute the upcoming sequential region. (Cores need not get fused on every parallel-to-sequential region boundary: if the sequential region is not long enough to amortize the cost of fusion, execution can continue without reconfiguration on one of the small cores.) If fusion is not allowed at this time, the FUSE instruction is turned into a NOP, and execution continues uninterrupted. Otherwise, all instructions following the FUSE instruction are flushed; the i-caches are flushed; the FMU, SMU, and the i-caches are reconfigured; and the rename map on the core that commits the FUSE instruction is transferred to the SMU. Data caches do not need any special actions to be taken upon reconfigurations: the coherence protocol naturally ensures correctness across configuration changes. Finally, the FMU signals the i-caches to start fetching in fused mode from the instruction that follows the FUSE instruction in program order.

**SPLIT operation.** The application advises the fused group of an upcoming parallel region using a SPLIT instruction. When the SPLIT instruction commits, in-flight instructions are allowed to drain, and enough copy instructions are generated to gather the architectural state into core zero's physical register file. When the transfer is complete, the FMU and SMU are reconfigured, and core zero starts fetching from the instruction that follows the SPLIT in program order. The other cores remain available to the application (although the OS may re-allocate them at any time after this point).

| Two-issue Processor Parameters | |
|---|---|
| **Frequency** | 4.0 GHz |
| **Fetch/issue/commit** | 2/2/2 |
| **Int/FP/AGU/Br Units** | 1/1/1/1 |
| **Int/FP Multipliers** | 1/1 |
| **Int/FP issue queues** | 16/16 |
| **Copy-In/Copy-Out queues** | 16/16 |
| **ROB entries** | 48 |
| **Int/FP registers** | 32+40 / 32+40 (Arch.+Ren.) |
| **Ld/St queue entries** | 12/12 |
| **Bank predictor** | 2048 entries |
| **Max. br. pred. rate** | 1 taken/cycle |
| **Max. unresolved br.** | 12 |
| **Br. penalty** | 7 cycles min. (14 when fused) |
| **Br. predictor** | Alpha 21264 |
| **RAS entries** | 32 |
| **BTB size** | 512 entries, 8-way |
| **iL1/dL1 size** | 16 kB |
| **iL1/dL1 block size** | 32B/32B |
| **iL1/dL1 round-trip** | 2/3 cycles (uncontended) |
| **iL1/dL1 ports** | 1 / 2 |
| **iL1/dL1 MSHR entries** | 8 |
| **iL1/dL1 associativity** | DM/4-way |
| **Coherence protocol** | MESI |
| **Memory Disambiguation** | Perfect |
| **Consistency model** | Release consistency |

**Table 1: Two-issue Processor Parameters.**

| Shared Memory Subsystem | |
|---|---|
| System bus transfer rate | 32GB/s |
| Shared L2 | 4MB, 64B block size |
| Shared L2 associativity | 8-way |
| Shared L2 banks | 16 |
| L2 MSHR entries | 16/bank |
| L2 round-trip | 32 cycles (uncontended) |
| Memory access latency | 328 cycles (uncontended) |

**Table 2: Parameters of the shared memory subsystem.**

[1] We prefer LWT's simplicity over a store set predictor solution [9, 14]. Nevertheless, load speculation in core fusion can also be implemented using store set predictors [14], with a few changes that we briefly outline here: (1) The predictor's smaller table (the *LFST [14]*) resides in the SMU; the significantly larger [14] per-core SSITs are effectively "merged," simply by virtue of aligned fetch. (2) Memory operations predicted dependent on a store are initially steered to the same core as that store, overriding the bank predictor. To accomplish this, LFST entries provide the bank ID to which the predicted-dependent instruction should be steered. (3) On a bank misprediction by a store, the LFST entry's bank ID (assuming it still contains the store's *inum* [14]) is updated appropriately. A few cycles after sending the update to the LFST (to allow for loads in flight from the SMU to arrive), the store "liberates" any memory operation that was flagged as dependent and steered to the same core.

| CMP Configuration | Composition (Cores) |
|---|---|
| CoreFusion | 8x2-issue |
| FineGrain-2i | 9x2-issue |
| CoarseGrain-4i | 4x4-issue |
| CoarseGrain-6i | 2x6-issue |
| Asymmetric-4i | 1x4-issue + 6x2-issue |
| Asymmetric-6i | 1x6-issue + 4x2-issue |

**Table 3: Composition of the evaluated CMP architectures.**

# 4. EXPERIMENTAL SETUP

## 4.1 Architecture

We evaluate the performance potential of core fusion by comparing it against five static homogeneous and asymmetric CMP architectures. As building blocks for these systems, we use two-, four-, and six-issue out-of-order cores. Table 1 shows the microarchitectural configuration of the two-issue cores in our experiments. Four- and six-issue cores have two and three times the amount of resources as each one of the two-issue cores, respectively, except that first level caches, branch predictor, and BTB are four times as large in the six-issue core (the sizes of these structures are typically powers of two). Across different configurations, we always maintain the same parameters for the shared portion of the memory subsystem (system bus and lower levels of the memory hierarchy). All configurations are clocked at the same speed. We model wake-up and selection delays in the two-issue core to be one cycle each, and extrapolate delays for four- and six-issue cores to be 2-2 and 3-2, respectively, using trends presented in the literature [32]. Our experiments are conducted using a detailed, heavily modified version of the SESC [35] simulator. Contention and latency are modeled at all levels. In fused mode, this includes two-cycle wire delays for cross-core communication across fetch, operand and commit wiring, the additional latency due to the eight-stage rename pipeline, and contention for SMU dispatch ports. (We explain later how we derive cross-core communication latencies.) For dynamic reconfiguration, we model a 400-cycle reconfiguration overhead in addition to draining the pipelines.

Since we explore an inherently area-constrained design space, choosing the right number of large and small cores requires estimating their relative areas. Prior work [24, 23, 31, 32] shows that the area overheads of key microarchitectural resources scale superlinearly with respect to issue width in monolithic cores. Burns et al. [8] estimate the area requirements of out-of-order processors by inspecting layout from the MIPS R10000 and from custom layout blocks, finding that four- and six-issue cores require roughly 1.9 and 3.5 times the area of a two-issue core, respectively, even when assuming clustered register files, issue queues, and rename maps, which greatly reduce the area penalty of implementing large SRAM arrays.[2] Recall also that our six-issue baseline's first level caches and branch predictor are four times as large as those of a two-issue core. Consequently, we model the area requirements of our four- and six-issue baselines to be two and four times higher than a two-issue core, respectively.[3]

We estimate the area overhead of core fusion additions conservatively, assuming that no logic is laid out under the metal layer for cross-core wiring. Specifically, we use the wiring area estimation methodology described in [25], assuming a 65nm technology and Metal-4 wiring with a 280nm wire pitch [17]. Accordingly, we find the area for fetch wiring (92 bits/link) to be 0.32mm$^2$, the area for rename wiring (250 bits/link) to be 0.99mm$^2$, and the area for the operand crossbar (80 bits / link) to be 1.90mm$^2$. The area of the commit wiring is negligible, as it is two bits wide. This yields a total area overhead of 3.21mm$^2$ for fusing a group of four cores, or 6.42mm$^2$ for our eight-core CMP. Using CACTI 3.2, we also estimate the total area overhead of the SMU, the extra i-cache tags, copy-in/copy-out queues, and bank predictors (four bank predictors, one per core) to be 0.48, 0.25, 0.15, and 0.23mm$^2$ per fusion group, respectively, for a total of 2.22mm$^2$ for the entire chip. Adding these to the wiring estimates, we find the total area overhead of core fusion to be 8.64mm$^2$. Even for a relatively small, non-reticle-limited, 200mm$^2$ die that devotes half of the area to the implementation of the cores, this overhead represents a little over two thirds of the area of one core. Hence, we conservatively assume the area overhead to be equal to one core.

We estimate the latency of our cross-core wiring additions conservatively, assuming that cores are laid out in a worst-case organization that maximizes cross-core communication delays. We assume that each group of four cores in our eight-core CMP must communicate over a distance equal to one half of the chip edge length. Assuming a 65nm technology, a 4GHz clock, and 50ps/mm Metal-4 wire delay [17], we find that it is possible to propagate signals over a distance of 5mm in one cycle. Even for a relatively large, reticle-limited, 400mm$^2$ die with a worst-case floorplan, this represents a two-cycle cross-core communication latency. While these delays are likely to be lower for a carefully organized floorplan [25] or for smaller dice, we conservatively model fetch, operand, and commit communication latencies to be equal to two cycles, and due to its wider links, we set the latency of the rename communication to three cycles (which makes the rename pipeline add up to eight cycles).

Table 3 details the number and type of cores used in our studies for all architectures we model. Our core-fusion-enabled CMP consists of eight two-issue cores. Two groups of four cores can each be fused to synthesize two large cores on demand. For our coarse-grain CMP baselines, we experiment with a CMP consisting of two six-issue cores (CoarseGrain-6i) and another coarse-grain CMP consisting of four four-issue cores (CoarseGrain-4i). We also model an asymmetric CMP with one six-issue and four two-issue cores (Asymmetric-6i), and another asymmetric CMP with one four-issue and six two-issue cores (Asymmetric-4i). Finally, we model a fine-grain CMP with *nine* two-issue cores (FineGrain-2i). The ninth core is added to compensate for any optimism in the area estimates for six- and four-issue cores, and for the area overhead of core fusion. We have verified that all the parallel applications in the paper use this ninth core effectively.

| Data Mining | Description | Problem size |
|---|---|---|
| BSOM | Self-organizing map | 2,048 rec., 100 epochs |
| BLAST | Protein matching | 12.3k sequences |
| KMEANS | K-means clustering | 18k pts., 18 attributes |
| SCALPARC | Decision Tree | 125k pts., 32 attributes |
| FIMI | Itemset Mining | 1M trans., 1.3% support |
| **SPEC OpenMP** | | |
| SWIM-OMP | Shallow water model | MinneSpec-Large |
| EQUAKE-OMP | Earthquake model | MinneSpec-Large |
| **NAS OpenMP** | | |
| MG | Multigrid Solver | Class A |
| CG | Conjugate Gradient | Class A |
| **Splash2** | | |
| BARNES | Evolution of galaxies | 16k particles |
| FMM | N-body problem | 16k particles |
| RAYTRACE | 3D ray tracing | car |

**Table 4: Simulated parallel applications and their input sizes.**

## 4.2 Applications

We evaluate our proposal by conducting simulations on parallel, evolving parallel, and sequential workloads. Our

[2]Note that, when all resources are scaled linearly, monolithic register files grow as O($w^3$), where $w$ is the issue width. This is due to the increase in the number of bit lines and word lines per SRAM cell, times the increase in physical register count.

[3]We also experimented with an eight-issue clustered core (optimistically assumed to be area-equivalent to the six-issue core), but found its performance to be inferior. Consequently, we chose the six-issue monolithic core as our baseline.

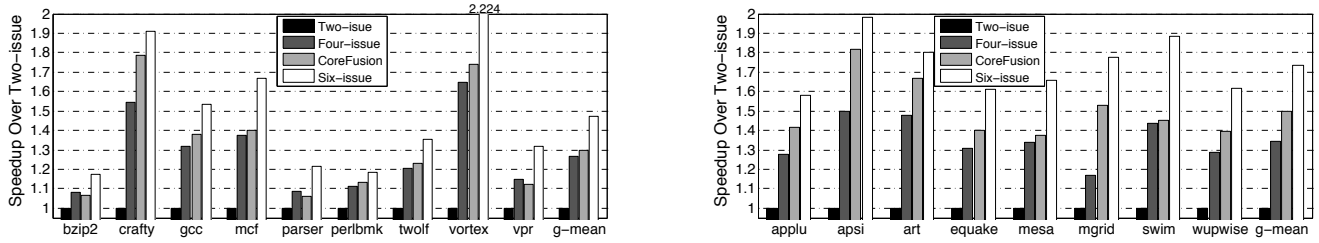


**Figure 7: Speedup over FineGrain-2i when executing SPECINT (left) and SPECFP (right) benchmarks.**

parallel workloads represent a mix of scalable scientific applications (three applications from the Splash-2 suite [39], two applications from the SPEC OpenMP suite [2], plus two parallel NAS benchmarks), and five parallelized data mining applications [1, 27, 34]. The input sets we use are listed in Table 4.

Our sequential workloads comprise nine integer and eight floating point applications from the SPEC2000 suite [20]. We use the MinneSpec reduced input sets [22]. In all cases, we skip the initialization parts and then simulate the applications to completion.[4]

We derive our evolving workloads from existing applications by following a methodology that aims at mimicking an actual incremental parallelization process. Specifically, we use Swim-OMP and Equake-OMP from the SPEC OpenMP suite, and MG from the OpenMP version of the NAS benchmarks to synthesize our evolving workloads. These applications contain multiple parallel regions that exploit loop-level parallelism [2]. We emulate the incremental parallelization process by gradually transforming sequential regions into parallel regions, obtaining more mature versions of the code at each turn. To do this, we first run each application in single-threaded mode and profile the run times of all regions in the program. We then create an initial version of the application by turning on the parallelization for the most significant region while keeping all other regions sequential. We repeat this process until we reach the fully parallelized version, turning on the parallelization of the next significant region at each step along the process.

# 5. EVALUATION

## 5.1 Sequential Application Performance

Figure 7 shows speedups with respect to FineGrain-2i on SPEC 2000 applications. As expected, the results indicate that wide-issue cores have significant performance advantages on sequential codes. Configurations with a six-issue monolithic core obtain average speedups of 73% and 47% on floating-point and integer benchmarks, respectively. (Speedups on floating-point benchmarks are typically higher due to higher levels of ILP present in these applications.) Configurations that employ a four-issue core observe average speedups of 35% and 27% on floating-point and integer-benchmarks, respectively. Core fusion improves performance over the fine-grain CMP by up to 81% on floating-point applications, with an average of 50%. On integer applications, up to 79% speedup improvements are obtained, with an average speedup of 30%.

In summary, when running sequential applications, the monolithic six-issue core performs best, and is followed by CoreFusion's fused core. FineGrain-2i is the worst architecture for this class of workloads. While core fusion enjoys a high core count to extract TLP, it can aggressively exploit ILP on single-threaded applications by adopting a fused configuration.

[4]Our simulation infrastructure currently does not support the other SPEC benchmarks.

### 5.1.1 Performance Analysis

In this section, we analyze and quantify the performance overhead of cross-core communication delays. We also investigate the efficacy of our distributed ROB and LSQ implementations.

**Distributed Fetch.** Our fused front-end communicates taken branches across the FMU. Consequently, while a monolithic core could redirect fetch in the cycle following a predicted-taken branch, core fusion takes two additional cycles. Figure 8 shows the speedups obtained when the fused front-end is idealized by setting the FMU communication latency to zero. The performance impact of the FMU delay is less than 3% on all benchmarks except vpr, indicating that there is significant slack between the fused front- and back-ends. Figure 9 illustrates this point by showing a breakdown of front-end activity for realistic (R) and idealized (I) FMU delays, as well as our six-issue monolithic baseline (6i). On memory-intensive floating-point applications, the fused front-end spends 33-92% of its time waiting for the back-end to catch up, and 10% of its time stalling for FMU communication on average. On integer codes, 10-29% of the front-end time is spent stalling for FMU communication, but removing this delay does not necessarily help performance: once the FMU delay is removed, the idealized front-end simply spends a commensurately higher portion of its total time waiting for the fused back-end. Overall, performance is relatively insensitive to the FMU delay.

**SMU and the Rename Pipeline.** Figure 8 shows the speedups obtained when pipeline depth and the SMU are idealized (by reducing the eight-stage rename pipe to a single stage, and allowing the SMU to dispatch an arbitrary number of instructions to each core, respectively). Depending on the application, the longer rename pipeline results in performance losses under 5%, with an average of less than 1%. While fusion increases the branch misprediction penalty from seven to fourteen cycles, both the branch predictor and the BTB are four times as large in fused mode, decreasing misprediction rates and lowering sensitivity to pipe depth. The performance impact of restricted SMU bandwidth is more pronounced, and ranges from 0-7%, with an average of 3%. However, considering the wiring overheads involved, and the impact on the two-issue base cores, these performance improvements do not warrant an implementation with higher dispatch bandwidth.

**Operand Crossbar.** Figure 8 shows the speedups achieved by an idealized operand crossbar with zero-cycle latency. Unlike communication delays incurred in the front-end of the machine, the latency of the operand crossbar affects performance noticably, resulting in up to 18% performance losses, with averages of 13% and 9% on integer and floating point applications, respectively. Sensitivity is higher on integer codes compared to floating-point codes: the latter are typically characterized by high levels of ILP, which helps hide the latency of operand communication by executing instructions from different dependence chains.

**Distributed ROB and LSQ.** Inevitably, core fusion's distributed ROB and LSQ organizations suffer from inefficien-

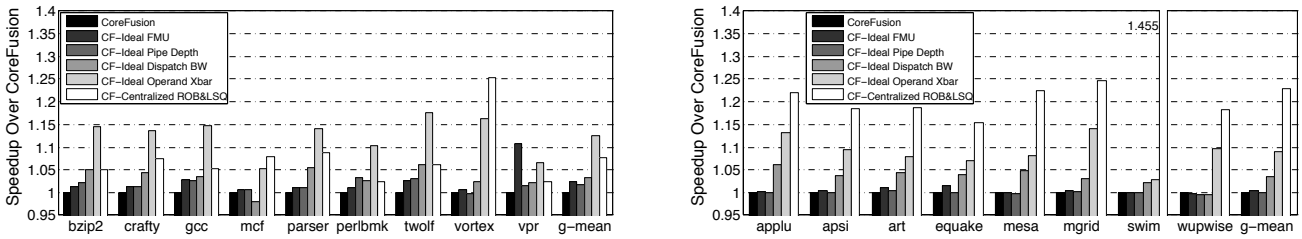


**Figure 8: Speedups obtained when the FMU latency, rename pipeline depth, SMU dispatch bandwidth, operand crossbar delay, or the distributed ROB/LSQ are idealized.**

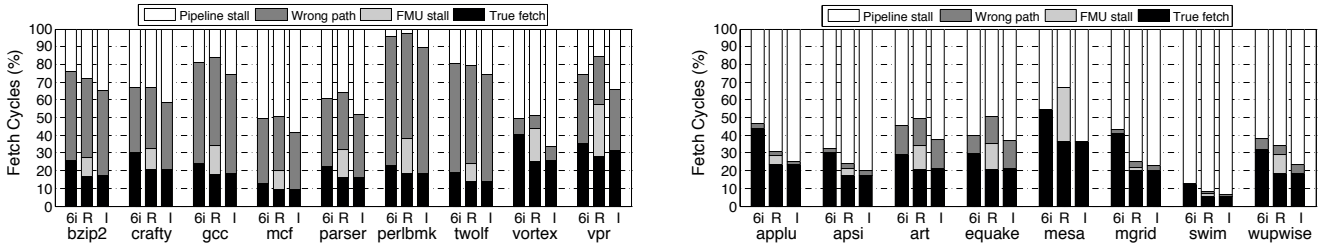


**Figure 9: Distribution of fetch cycles on SPECINT (left) and SPECFP (right) benchmarks. 6i, R, and I denote our six-issue monolithic baseline, a realistic fused front-end with a two-cycle FMU delay, and an idealized fused front-end with no FMU delay, respectively.**

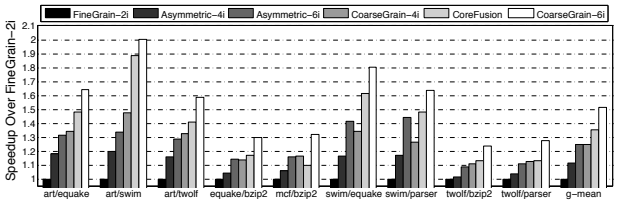


**Figure 10: Speedup over FineGrain-2i when executing two sequential applications simultaneously.**

cies that would be absent from a monolithic implementation (e.g., NOP insertion for aligned ROB allocation, and dummy entry allocation in the LSQ). Figure 8 show that eliminating these inefficiencies improves performance by 7 and 23% over core fusion on integer and floating point codes, respectively. Along with the latency of the operand communication, this reduction in effective LSQ and ROB sizes has the highest impact on core fusion's performance.

### 5.1.2 Desktop Workload Performance

One potential shortcoming of ACMP designs is that they may not accommodate well more sequential codes than the number of large cores on the die. For example: In a desktop environment, Asymmetric-4i and -6i will readily accommodate *one* sequential program on the large core. However, if the desktop runs *two* such programs, Asymmetric-4i and -6i will necessarily have to allocate a weak core to one of the programs. In contrast, CoreFusion will be able to synthesize two large cores to accommodate both programs in this environment. (In environments with a relatively large number of concurrent programs, we expect the relative performance of the different CMP designs to be along the lines of the results for parallel software (Section 5.2)).

We would like to assess how Asymmetric-4i and -6i would stack up against CoreFusion's hardware flexibility in this environment. Intuitively, we expect Asymmetric-6i to perform competitively, since the monolithic 6-issue core generally outperforms CoreFusion's largest core configuration (Section 5.1).

We derive our desktop workloads from the SPEC2000 suite [20]. We classify applications as high- and low-ILP benchmarks based on how much speedup they obtain in going from a two-issue core to four- and six-issue cores. We then use these classifications to guide our workload construction process. We set the degree of multiprogramming to two applications, and we form a total of nine workloads with different ILP characteristics: high-ILP workloads, low-ILP workloads, and mixed (both high and low ILP) workloads. We conduct preliminary experiments using a static oracle scheduler, as well as a published dynamic scheduler [24], and find the static scheduler to perform equally well or better for all our desktop workloads. Thus, we use the oracle scheduler in our evaluation.

Figure 10 shows the results. Once again, CoreFusion is closest in performance to the optimum static CMP configuration for this type of workload (CoarseGrain-6i). And indeed, Asymmetric-6i's performance is closely behind CoreFusion's. We will see shortly, however, that CoreFusion significantly outperforms Asymmetric-6i in the experiments with parallel and evolving software (Section 5.2 and 5.3, respectively). This is indicative of CoreFusion's overall superior flexibility across a diverse set of workload environments. Finally, the results for Asymmetric-4i indicate that the ACMP is clearly inferior to CoreFusion's ability to synthesize two large eight-issue cores to accommodate both programs.

## 5.2 Parallel Application Performance

Figure 11 compares the performance of core fusion against our baseline CMP configurations on parallel workloads. Results are normalized to the performance of single-threaded runs on FineGrain-2i. As expected, on scalable parallel applications, maximizing the number of cores leads to significant performance improvements. The fine-grain CMP performs best on this class of applications due to its higher number of cores that allows it to aggressively harness TLP. FineGrain-2i is followed closely by CoreFusion, which has one fewer core due to its area overheads. Coarse-grain and asymmetric designs sacrifice parallel application performance significantly to improve single-thread performance. These architectures are forced to *trade off* TLP for ILP by their static nature, while CoreFusion aims to *synthesize* the right ILP/TLP balance based on workload needs.

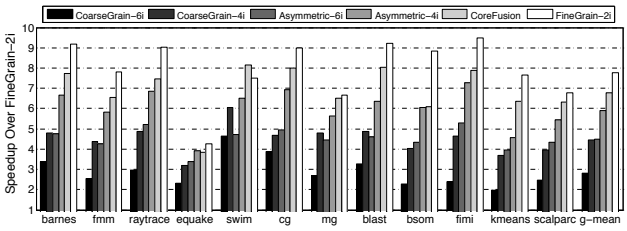


**Figure 11: Speedup over single-thread run on FineGrain-2i when executing parallel applications.**

## 5.3 Evolving Application Performance

Figure 12 compares the performance of all six CMP configurations on our evolving workloads. Each graph shows the speedups obtained by each architecture as applications evolve from sequential (stage zero) to highly parallel (last stage). Results are normalized to a sequential run of the application on FineGrain-2i. When running on the asymmetric CMPs, we schedule the master thread on the large core so that sequential regions are sped up. Parallel regions are executed on all cores.[5] We evaluate our proposal by applying dynamic core fusion to fuse/split cores when running sequential/parallel regions, respectively.

When applications are not parallelized (stage zero), exploiting ILP is crucial to obtaining high performance. As a result, coarse-grain CMPs, asymmetric CMPs and CoreFusion all enjoy speedups over the fine-grain CMP. In this regime, performance is strictly a function of the largest core on the chip. CoreFusion outperforms all but the six-issue configurations, due to its ability to exploit high levels of ILP.

In the intermediate stages, significant portions of the applications are still sequential, and exploiting ILP is still crucial for getting optimum performance. Asymmetric-6i's monolithic core marginally outperforms CoreFusion's fused core, but as a result of dynamic fusion and fission, CoreFusion enjoys a higher core count on parallel regions, thereby exploiting higher levels of TLP. Asymmetric-4i has two more cores than Asymmetric-6i, but the application does not yet support enough TLP to cover the performance hit with respect to Asymmetric-6i's six-issue core on sequential regions. Because of the scarcity of TLP in this evolutionary stage, FineGrain-2i performs worst among all architectures.

Eventually, enough effort is expended in parallelization to convert each program into a highly parallel application. In MG, performance is determined strictly by core count. FineGrain-2i obtains the best speedup (6.7), followed immediately by CoreFusion (6.5). Architectures that invest in ILP (Asymmetric-6i and CoarseGrain-6i) take a significant performance hit (speedups of 4.5 and 2.7, respectively). In Swim-OMP and Equake-OMP, CoreFusion still performs the best, followed closely by the fine-grain CMP. This is because these applications, even at this parallelization stage, have sequential regions, on which CoreFusion outperforms FineGrain-2i through dynamic fusion. Note, however, that statically allocating a large core to obtain speedup on these regions does not pay off, as evidenced by the lower performance of Asymmetric-4i and -6i compared to CoreFusion. Attempting to exploit ILP in these regions is worthwhile only if it does not adversely affect the exploitation of TLP.

In summary, performance differences between the best and the worst architectures at any parallelization stage are high, and moreover, the best architecture at one end of the evolutionary spectrum performs worst at the other end. As applications evolve through the incremental parallelization process, performance improves on all applications. Throughout this evolution, CoreFusion is the only architecture that consistently performs the best or rides close to the best configuration. While all static architectures get "stuck" at some (different) point along the incremental parallelization process, core fusion adapts to the changing demands of the evolving application and obtains significantly higher overall performance.

[5]We also experimented with running parallel regions on small cores only, but found that the results were inferior.

# 6. RELATED WORK

## 6.1 Reconfigurable Architectures

Smart memories [28] is a reconfigurable architecture capable of merging in-order RISC cores to form a VLIW machine. The two configurations are not ISA-compatible, and the VLIW configuration requires specialized compiler support. In contrast, core fusion merges out-of-order cores while remaining transparent to the ISA, and it does not require specialized compiler support. Voltron [40] is a multicore architecture that can exploit hybrid forms of parallelism by organizing its cores as a wide VLIW machine. Voltron's VLIW configuration relies on specialized compiler support.

TRIPS [37] is a pioneer reconfigurable computing paradigm that aims to meet the demands of a diverse set of applications by splitting ultra-large cores. TRIPS and core fusion represent two very different visions toward achieving a similar goal. In particular, TRIPS opts to implement a custom ISA and microarchitecture, and relies heavily on compiler support for scheduling instructions to extract ILP. Core fusion, on the other hand, favors leveraging mature microarchitecture technology and existing ISAs, and does not require specialized compiler support.

## 6.2 Clustered Architectures

Core fusion borrows from some of the mechanisms developed in the context of clustered architectures [4, 5, 7, 10, 11, 13, 18, 32, 41]. Our proposal is closest to the recent thrust in clustered multithreaded processors (CMT) [15, 16, 26]. In this section, we give an overview of the designs that are most relevant to our work, and highlight the limitations that preclude these earlier proposals from supporting workload diversity effectively. Table 5 provides an outline of our discussion.

El-Moursy et al. [16] consider several alternatives for partitioning multithreaded processors. Among them, the closest one to our proposal is a CMP that comprises multiple clustered multithreaded cores (CMP-CMT). The authors evaluate this design with both shared and private L1 data cache banks, finding that restricting the sharing of banks is critical for obtaining high performance with multiple independent threads. However, the memory system is not reconfigurable; in particular, there is no mechanism for merging independent cache banks when running sequential code. Consequently, sequential regions/applications can exploit only a fraction of the L1 data cache and load/store queues on a given core. Similarly, each thread is assigned its own ROB, and these ROBs cannot be merged. Finally, neither coherence nor memory consistency issues are considered.

Latorre et al. [26] propose a CMT design with multiple front- and back-ends, where the number of back-ends assigned to each front-end can be changed at runtime. Each front-end can fetch from only a single thread, and front-ends cannot be merged or reconfigured. When running a single thread, only one of these front-ends is active. As a result, each front-end has to be large enough to support multiple (potentially all) back-ends, and this replication results in significant area overheads (*each* front-end supports four-wide fetch, has a 512-entry ROB, a 32k-entry branch predictor, a 1k-entry i-TLB and a trace cache with 32k micro-ops). Stores allocate entries on all back-ends, and these entries are not recycled. This requires the store queue in each back-end to be large enough to accommodate *all* of the thread's uncommitted stores.

Collins et al. [15] explore four alternatives for clustering SMT processors. Among them, the most relevant to

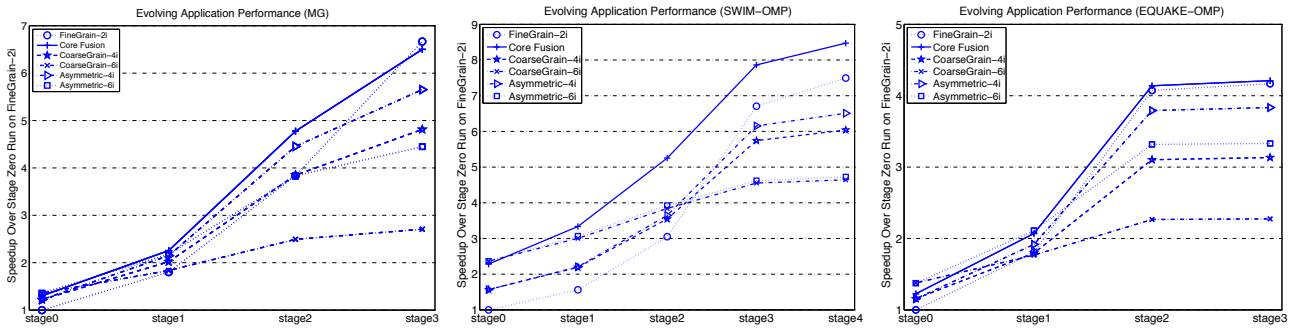


**Figure 12: Speedup over stage zero run on FineGrain-2i.**

| *Architecture* | *Performance Potential* | | *Throughput Potential* | *Modularity* | | | *Reconfigurability* | | |
|---|---|---|---|---|---|---|---|---|---|
| | Sequential | Parallel | | FE | BE | Caches | FE | BE | Caches |
| *Collins et al. [15]* | Low | High | High | Yes[1] | Yes | No | No | No | No |
| *El-Moursy et al. [16]* (Shared Banks) | High | Low | Low | Partial[1] | Yes | Yes | No | Yes | No |
| *El-Moursy et al. [16]* (Private Banks) | Low | Not Supported | High | Partial[1] | Yes | Yes | No | Yes | No |
| *Latorre et al. [26]* (Fewer, large FEs) | High | Low | Low | Partial[1] | Yes | Yes | No | Yes | Yes |
| *Latorre et al. [26]* (More, small FEs) | Low | High | High | Yes[1] | Yes | Yes | No | Yes | Yes |
| *Parcerisa [33]* | High | Not Supported | Not Supported | Yes[2] | Yes | No | No | Yes | No |
| ***Core Fusion*** | **High** | **High** | **High** | **Yes** | **Yes** | **Yes** | **Yes** | **Yes** | **Yes** |

[1]Modules cannot collectively support one thread
[2]Modules do not support more than one thread

**Table 5: Comparison to recent proposals for clustered processors. FE and BE stand for front- and back-end, respectively.**

our work is a processor with clustered front-ends, execution units, and register files. Each front-end is capable of fetching from multiple threads, but the front-ends are not reconfigurable, and multiple front-ends cannot be merged when running a single thread. As the authors explain, the reduced fetch/rename bandwidth of each front-end can severely affect single-thread performance. There is no direct communication between the register files, and sequential codes can utilize only a single cluster's register file at any point in time.

Parcerisa [33] partitions the front-end of a conventional clustered architecture to improve clock frequency. The front-end is designed to fetch from a single thread: parallel, evolving, or multiprogrammed workloads are not discussed and reconfiguration is not considered. The branch predictor is interleaved on high-order bits, which may result in under-utilized space. Mechanisms for keeping consistent global history across different branch predictor banks are not discussed.

Chaparro et al. [13] distribute the rename map and the ROB to obtain temperature reductions. Fetch and steering are centralized. Their distributed ROB expands each entry with a pointer to the ROB entry (possibly remote) of the next dynamic instruction in program order. Committing involves pointer chasing across multiple ROBs. In core fusion, we also fully distribute our ROB, but without requiring expensive pointer chasing mechanisms across cores.

## 6.3 Other Related Work

Trace Processors [36] overcome the complexity limitations of monolithic processors by distributing instructions to processing units at the granularity of traces. The goal is the complexity-effective exploitation of ILP in sequential applications. Other types of workloads (e.g., parallel codes) are not supported. MultiScalar processors [38] rely on compiler support to exploit ILP with distributed processing elements. The involvement of the compiler is prevalent in this approach (e.g., for register communication, task extraction, and marking potential successors of a task). On the contrary, core fusion does not require specialized compiler support. Neither multiscalar nor trace processors address the issue of accommodating software diversity in CMPs or facilitating incremental software parallelization, which is a key focus of our work.

# 7. CONCLUSIONS

In this paper, we have introduced a novel reconfigurable CMP architecture that we call *core fusion*, which allows relatively simple CMP cores to dynamically fuse into larger, more powerful processors. The goal is to accommodate software diversity gracefully, and to dynamically adapt to changing demands by workloads. We have presented a complete hardware solution to support core fusion. In particular, we have described complexity-effective solutions for collective fetch, rename, execution, cache access, and commit, that respect the fundamentally independent nature of the base cores. The result is a flexible CMP architecture that can adapt to a diverse collection of software, and that rewards incremental parallelization with higher performance along the development curve. It does so without requiring higher software complexity, a customized ISA, or specialized compiler support.

Through detailed simulations, we have identified and quantified the degree to which core fusion's major components impact performance. Specifically, we have observed that the cross-core operand communication cost and core fusion's ROB/LSQ allocation inefficiencies have the most impact on performance. We have also pitted core fusion against several static CMP designs, and confirmed that core fusion's versatility across a diverse software spectrum makes it a promising design approach for future CMPs.

# ACKNOWLEDGMENTS

This work was funded in part by NSF awards CCF-0429922, CNS-0509404, CAREER Award CCF-0545995, and an IBM Faculty Award. Meyrem Kırman and Nevin Kırman were supported in part by two Intel graduate fellowships.

# 8. REFERENCES

[1] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, pages 403–410, 1990.

[2] V. Aslot and R. Eigenmann. Quantitative performance analysis of the SPEC OMPM2001 benchmarks. *Scientific Programming*, 11(2):105–124, 2003.

[3] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The impact of performance asymmetry in emerging multicore architectures. In *Intl. Symp. on Computer Architecture*, pages 506–517, Madison, Wisconsin, June 2005.

[4] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi. Dynamically managing the communication-parallelism trade-off in future clustered processors. In *Intl. Symp. on Computer Architecture*, pages 275–287, San Diego, CA, June 2003.

[5] A. Baniasadi and A. Moshovos. Instruction distribution heuristics for quad-cluster, dynamically-scheduled, superscalar processors. In *Intl. Symp. on Microarchitecture*, pages 337–347, Monterey, CA, December 2000.

[6] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz, and U. Weiser. Correlated load-address predictors. In *Intl. Symp. on Computer Architecture*, pages 54–63, Atlanta, GA, May 1999.

[7] R. Bhargava and L. K. John. Improving dynamic cluster assignment for clustered trace cache processors. In *Intl. Symp. on Computer Architecture*, pages 264–274, San Diego, CA, June 2003.

[8] J. Burns and J.-L. Gaudiot. Area and system clock effects on SMT/CMP processors. In *Intl. Conf. on Parallel Architectures and Compilation Techniques*, page 211, Barcelona, Spain, September 2001.

[9] B. Calder and G. Reinman. A comparative survey of load speculation architectures. *Journal of Instruction-Level Parallelism*, 2, May 2000.

[10] R. Canal, J.-M. Parcerisa, and A. González. A cost-effective clustered architecture. In *Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 160–168, Newport Beach, CA, October 1999.

[11] R. Canal, J.-M. Parcerisa, and A. González. Dynamic cluster assignment mechanisms. In *Intl. Symp. on High-Performance Computer Architecture*, pages 132–142, Toulouse, France, January 2000.

[12] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann, San Francisco, CA, 2001.

[13] P. Chaparro, G. Magklis, J. González, and A. González. Distributing the frontend for temperature reduction. In *Intl. Symp. on High-Performance Computer Architecture*, pages 61–70, San Francisco, CA, February 2005.

[14] G. Chrysos and J. Emer. Memory dependence prediction using store sets. In *Intl. Symp. on Computer Architecture*, pages 142–153, Barcelona, Spain, June–July 1998.

[15] J. D. Collins and D. M. Tullsen. Clustered multithreaded architectures - pursuing both ipc and cycle time. In *Intl. Parallel and Distributed Processing Symp.*, Santa Fe, New Mexico, April 2004.

[16] A. E.-Moursy, R. Garg, D. H. Albonesi, and S. Dwarkadas. Partitioning multi-threaded processors with a large number of threads. In *Intl. Symp. on Performance Analysis of Systems and Software*, pages 112–123, Austin, TX, March 2005.

[17] P. Bai et al. A 65nm logic technology featuring 35nm gate length, enhanced channel strain, 8 cu interconnect layers, low-k ILD and $0.57\mu m^2$ SRAM cell. In *IEEE Intl. Electron Devices Meeting*, Washington, DC, December 2005.

[18] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. The Multicluster architecture: Reducing cycle time through partitioning. In *Intl. Symp. on Microarchitecture*, pages 149–159, Research Triangle Park, NC, December 1997.

[19] J. González, F. Latorre, and A. González. Cache organizations for clustered microarchitectures. In *Workshop on Memory Performance Issues*, pages 46–55, Munich, Germany, June 2004.

[20] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, 33(7):28–35, July 2000.

[21] R. E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 9(2):24–36, March 1999.

[22] A. KleinOsowski and D. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1, June 2002.

[23] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Intl. Symp. on Microarchitecture*, pages 81–92, San Diego, CA, December 2003.

[24] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Intl. Symp. on Computer Architecture*, pages 64–75, München, Germany, June 2004.

[25] R. Kumar, V. Zyuban, and D. M. Tullsen. Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling. In *Intl. Symp. on Computer Architecture*, pages 408–419, Madison, Wisconsin, June 2005.

[26] F. Latorre, J. González, and A. González. Back-end assignment schemes for clustered multithreaded processors. In *Intl. Conf. on Supercomputing*, pages 316–325, Malo, France, June–July 2004.

[27] R. Lawrence, G. Almasi, and H. Rushmeier. A scalable parallel algorithm for self-organizing maps with applications to sparse data mining problems. Technical report, IBM, January 1998.

[28] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz. Smart Memories: a modular reconfigurable architecture. In *Intl. Symp. on Computer Architecture*, pages 161–171, Vancouver, Canada, June 2000.

[29] J. F. Martínez, J. Renau, M. C. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *Intl. Symp. on Microarchitecture*, Istanbul, Turkey, November 2002.

[30] M. Moudgill, K. Pingali, and S. Vassiliadis. Register renaming and dynamic speculation: An alternative approach. In *Intl. Symp. on Microarchitecture*, pages 202–213, Austin, TX, December 1993.

[31] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, Cambridge, MA, October 1996.

[32] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Intl. Symp. on Computer Architecture*, pages 206–218, Denver, CO, June 1997.

[33] J.-M. Parcesira. *Design of Clustered Superscalar Microarchitectures*. Ph.D. dissertation, Univ. Politècnica de Catalunya, April 2004.

[34] J. Pisharath, Y. Liu, W.-K. Liao, A. Choudhary, G. Memik, and J. Parhi. NU-MineBench 2.0. Technical Report CUCIS-2005-08-01, Center for Ultra-Scale Computing and Information Security, Northwestern University, August 2005.

[35] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. `http://sesc.sourceforge.net`.

[36] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. E. Smith. Trace processors. In *Intl. Symp. on Microarchitecture*, pages 138–148, Research Triangle Park, NC, December 1997.

[37] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Intl. Symp. on Computer Architecture*, pages 422–433, San Diego, CA, June 2003.

[38] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Intl. Symp. on Computer Architecture*, pages 414–425, Santa Margherita Ligure, Italy, June 1995.

[39] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Intl. Symp. on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.

[40] H. Zhong, S. A. Lieberman, and S. A. Mahlke. Extending multicore architectures to exploit hybrid parallelism in single-thread applications. In *Intl. Symp. on High-Performance Computer Architecture*, Phoenix, Arizona, February 2007.

[41] V. V. Zyuban and P. M. Kogge. Inherently lower-power high-performance superscalar architectures. *IEEE Transactions on Computers*, 50(3):268–285, March 2001.