A Reconfigurable Chip Multiprocessor Architecture to Accommodate Software Diversity

Engin İpek Meyrem Kırman Nevin Kırman José F. Martínez

Computer Systems Laboratory
Cornell University
Ithaca, NY 14853 USA

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

ABSTRACT

We present *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.

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.

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. 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. 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.

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, 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 accommodate software diversity better: 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). 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.

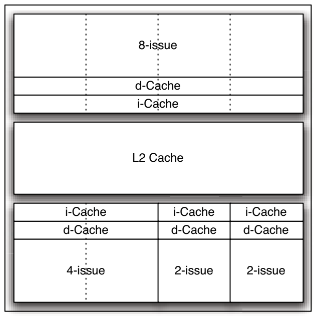


Figure 1: Conceptual floorplan of an eight-core CMP with core fusion capability. The figure shows a configuration example comprising two independent cores, a two-core fused group, and a four-core fused group. The figure is not meant to represent an actual floorplan.

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 CPUs.

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 latency from a core into the FMU and out to any other core is two cycles (Section 4).

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

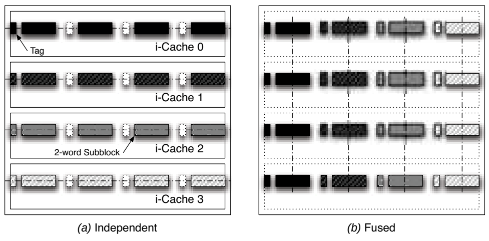


Figure 2: Illustrative example of four i-caches organized *(a)* independently or *(b)* fused. In independent mode, four subblocks and one tag within each i-cache constitute a cache block. In fused mode, a cache block spans four i-caches, each i-cache being responsible for a subblock and a tag replica.

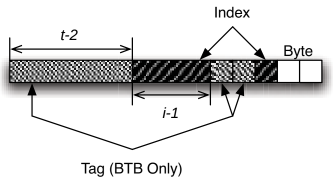


Figure 3: Configuration-oblivious indexing utilized in branch prediction and BTB. In the figure, i bits are used for indexing and t for tagging (tagging only meaningful in the BTB). Of course, i and t are generally not the same for branch predictor and BTB. Because of aligned fetch, the two tag bits sandwiched between index bits match the core number in the fused configuration.

(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 delivered *(a)* 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 [11]. 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 3 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 misprediction, sends the new target PC

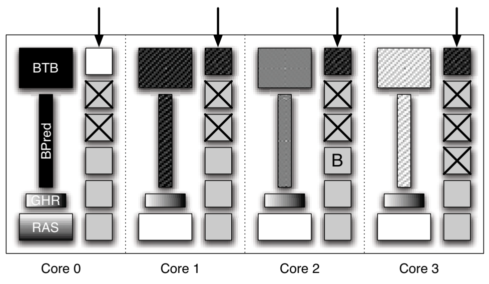


Figure 4: Example of aligned fetch in fused mode. In the figure, cores squash overfetched instructions as they receive a predict-taken notice from Core 2 with a two-cycle delay. The new target starts at Core 1, and thus Core 0 skips the first fetch cycle. Notice the banked branch predictor and BTB, the replicated GHR, and the Core-0-managed RAS.

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.

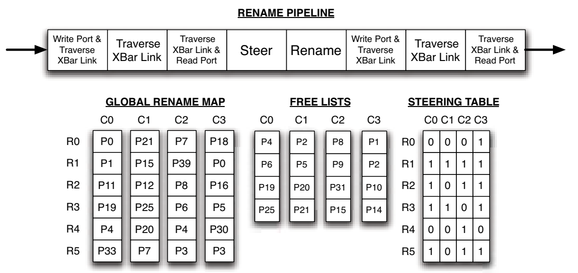
Naturally, on a misprediction, misspeculated instructions are squashed in all cores. This is also the case for instructions "overfetched" along the not-taken path on a taken branch, since the target PC will arrive with a delay of a few cycles. In Figure 4, Core 2 predicts branch B to be taken. After 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 the lines of the outstanding branch queue (OBQ) mechanism in the Alpha 21264 microprocessor [9].

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


RENAME PIPELINE
Write Port & Traverse XBar Link
Traverse XBar Link
Traverse XBar Link & Read Port
Steer
Rename
Write Port & Traverse XBar Link
Traverse XBar Link
Traverse XBar Link & Read Port
GLOBAL RENAME MAP
FREE LISTS
STEERING TABLE

GLOBAL RENAME MAP

	C0	C1	C2	C3
R0	P0	P21	P7	P18
R1	P1	P15	P39	P0
R2	P11	P12	P8	P16
R3	P19	P25	P6	P5
R4	P4	P20	P4	P30
R5	P33	P7	P3	P3

FREE LISTS

C0	C1	C2	C3
P4	P2	P8	P1
P6	P5	P9	P2
P19	P20	P31	P10
P25	P21	P15	P14

STEERING TABLE

	C0	C1	C2	C3
R0	0	0	0	1
R1	1	1	1	1
R2	1	0	1	1
R3	1	1	0	1
R4	0	0	1	0
R5	1	0	1	1

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.

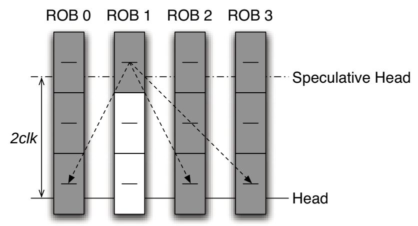


Figure 6: Simplified diagram of core fusion's distributed ROB. In the figure, ROB 1's head instruction pair is not ready to commit, which is communicated to the other ROBs. Speculative and conventional heads are spaced so that the message arrives just in time (2 clock cycles in the example). Upon completion 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.

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 cross-core communication 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 four free lists 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 [14] 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 eight write ports. The steering table requires eight 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 [12, 13]. 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).

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 **??** 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 *speculative head pointer* in addition to the conventional head and tail pointers 6. Instructions always pass through the speculative ROB head before they reach the actual ROB head and commit. Instructions that are not ready to commit by the time they reach the speculative ROB head stall immediately, and send a "stall" signal to all other cores. Later, as they become ready, they move past the speculative ROB head, and send a "resume" signal to the other cores. The number of ROB entries between the speculative 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 speculative 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 [2, 5, 7, 10, 15]. 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 [2, 3]. 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 placeholders 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) [9] 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 FMU, SMU, and the i-caches are configured; 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 fetch-

[1]We prefer LWT's simplicity over a store set predictor solution [4, 6]. Nevertheless, load speculation in core fusion can also be implemented using store set predictors [6], with a few changes that we describe in our WCED '06 paper [8].

Processor	
Frequency	4.0 GHz
Fetch/issue/commit	2/2/2
Int/FP issue queues	16/16
ROB entries	48
Integer FUs	1×ALU 1×AGU 1×Br 1×Mul/Div
Floating-point FUs	1×ALU 1×Mul/Div
Int/FP registers	32+40 / 32+40 (Architectural+Rename)
Max. br. pred. rate	1 taken/cycle
Br. predictor	Alpha 21264
Br. penalty	7 cycles minimum (14 cycles when fused)
Max. unresolved br.	12
BTB size	512 entries, direct mapped
RAS size	32 entries
Ld/St queue entries	12/12
Bank predictor	2K-entries
Memory Disambiguation	Perfect
iL1/dL1 size	16 kB
iL1/dL1 block size	32B/32B
iL1/dL1 associativity	DM/4-way
iL1/dL1 ports	1 / 2
iL1/dL1 round-trip	2/3 cycles (uncontended)
iL1/dL1 MSHR entries	8
Memory System	
Coherence protocol	MESI
Consistency model	Release consistency
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 round-trip	320 cycles (uncontended)

Table 1: Baseline two-issue core and memory system parameters.

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 2: Composition of the evaluated CMP architectures.

ing 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).

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 (this mainly favors the wide-issue cores). We conservatively model core fusion's 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). The details regarding core fusion latencies can be found in our WCED '06 paper [8].

SPEC OpenMP	**Description**	**Input set**
SWIM-OMP	Shallow water model	MinneSpec-Large
EQUAKE-OMP	Earthquake model	MinneSpec-Large
NAS OpenMP		
MG	Multigrid Solver	Class A

Table 3: Simulated parallel applications and their input sizes.

Since we explore an inherently area-constrained design space, choosing the right number of large and small cores requires estimating their relative areas. Details can be found in our WCED '06 paper [8]. Table 2 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 (Section 4.2) use this ninth core effectively.

4.2 Applications

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 [1]. 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

Figure 7 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). 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.[2] 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

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

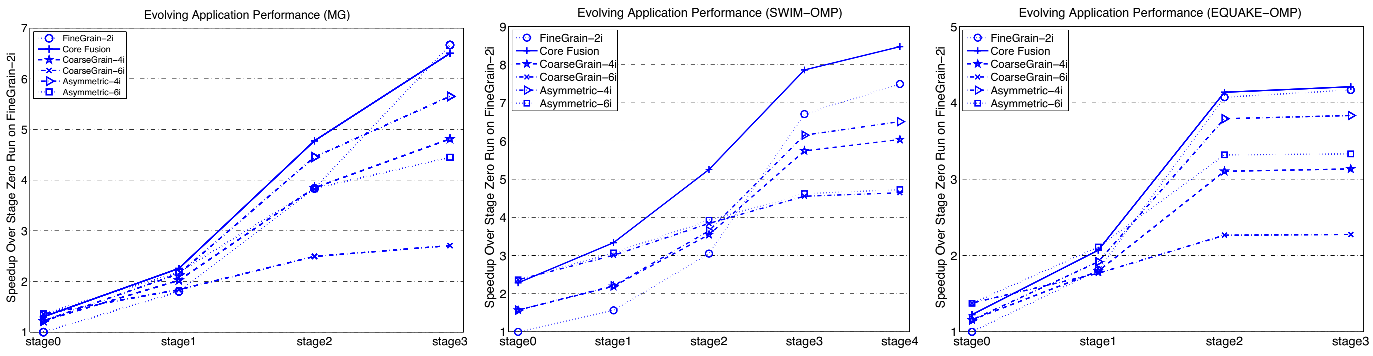


Figure 7: Speedup over stage zero run on FineGrain-2i.

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.

6 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.

REFERENCES

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

[2] 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.

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] Engin İpek, Meyrem Kırman, Nevin Kırman, and José F. Martínez. Accommodating workload diversity in chip multiprocessors via adaptive core fusion. In *Workshop on Complexity-effective Design (WCED), conc. with ISCA*, Boston, MA, June 2006.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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