HIGH PERFORMANCE SEQUENTIAL EXECUTION IN FINE-GRAIN MULTICORE PROCESSORS VIA CORE AGGREGATION

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Meyrem Kırman

February 2010

HIGH PERFORMANCE SEQUENTIAL EXECUTION IN FINE-GRAIN MULTICORE PROCESSORS VIA CORE AGGREGATION

Meyrem Kırman, Ph.D.

Cornell University 2010

This dissertation 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 improves sequential-code performance and thus gracefully accommodates software diversity in future's highly-parallel 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.

We first present an effective approach to dynamically fuse multiple narrow-issue out-of-order cores into a more powerful out-of-order execution engine. The use of out-of-order base cores provides the design with valuable opportunities for latency hiding.

Next, we present a second set of mechanisms to dynamically fuse multiple in-order cores into a more powerful out-of-order execution engine. In-order cores are extremely power-efficient and simple, and they help maximize core count, which is ideal for exploiting thread-level parallelism (TLP). However, sequential-code performance is significantly degraded. Enabling core fusion on such substrates proves to be very effective in boosting performance, and only with relatively small hardware overhead.

BIOGRAPHICAL SKETCH

Meyrem Kırman graduated with B.S. degrees in Control and Computer Engineering and Electronics and Communication Engineering in 2002 and 2003, respectively, from Istanbul Technical University (ITU), Turkey. In Fall 2003, she started her M.S./Ph.D. study in Electrical and Computer Engineering at Cornell University. Along the way, she earned her M.S. degree in 2007. Her research interests included checkpointed processor architectures, reconfigurable processor architectures, on-chip optical interconnects and memory-system design for chip multiprocessors.

To my dearest family

ACKNOWLEDGEMENTS

I would like to express my gratitude to Prof. José F. Martínez, my adviser, for all his contributions to my academic and personal development, and for providing a great working environment. His continuous guiding and support, always constructive critiques, and high standards made my graduate study a valuable and enriching one. I thank him for his patient and devoted efforts for developing our presentation, writing and communication skills. Discussions with him have always been very inspiring and instructive. I also thank him for creating a working environment which I feel great pleasure to work in, as well as providing numerous opportunities to us for attending conferences and other professional activities.

I am greatly indebted to my sister Nevin Kırman. Without her support, it would be difficult to successfully complete this work. She is continuous source of motivation and joy. I deeply thank her for always being with me in my good and difficult times.

I would like to thank Prof. Rajit Manohar and Prof. David Albonesi for being a part of my committee, their useful feedback, and their contributions to the Computer Systems Laboratory (CSL) and its warm environment.

I also thank my group members and other CSL members for fruitful collaborations, numerous discussions, and the good time spent together. I feel lucky to be a part of such friendly and stimulating research environment.

Finally, I am deeply thankful to my family for their continuous support and motivation, comfort and encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 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, sustaining long-term performance scalability calls for power- and performance-efficient, small-footprint core (micro)architectures that maximize raw performance per power and overall throughput. Consequently, harnessing the full potential of future CMPs makes the widespread adoption of parallel programming inevitable. Unfortunately, code parallelization constitutes a tedious, time-consuming, and error-prone effort. Significant amount of code, therefore, is anticipated to remain sequential in future's parallel computing substrates.

Beside the existing legacy sequential applications, difficulties of parallel programming is likely to result in a dynamic and diverse landscape of software of very different characteristics and in different stages of development: from purely sequential, to highly parallel, and everything in between. Improving regions of sequential code, even in highly-parallel applications, is crucial to maintain scalability, as also stressed by an article revisiting the Amdahl's Law in the multicore era [23]. In some cases, the problem itself is sequential or hard-to-parallelize in nature. As a result, future highly-parallel CMP substrates have to address sequential codes whose performance will be confined to a small core.

*

Asymmetric chip multiprocessors (ACMPs) [3, 31, 32] 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 thesis, 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 to execute sequential code at higher performance. We envision a core fusion CMP as a homogeneous substrate with conventional memory coherence/consistency support, where groups of cores and their i- and d-caches can be fused at run-time into CPUs that have aggregated fetch, issue, and commit width, and aggregated i-cache, d-cache, branch predictors. Our approach promotes single-thread performance in many core CMPs of homogenous substrate of simpler and smaller cores by aggressively extracting ILP, improving memory-latency tolerance, and increase memory-level parallelism.

The core-fusion concept 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, Asymmetric CMPs (ACMPs) [3, 31] 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 several 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 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 other fault-free cores, and even smaller-way fusion on the other cores in the fusion group. Bug/hard fault isolation may be significantly more challenging in designs based on large cores.

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. 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 [52] or Smart Memories [36], and from speculative architectures such as Multiscalar [53].
- 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.

In this thesis, we look at both narrow out-of-order and in-order core substrates to build upon when constructing a core-fusion architecture. The choice of base core presents a trade-off to chip designers. An out-of-order core achieves higher sequential-code performance, while an in-order core provides higher power and cost efficiency, and maximizes core count and throughput on CMPs.

In the first part of our thesis, we construct a core-fusion architecture based on dual-issue out-of-order cores. In the second part of the thesis, we construct a core-fusion architecture based on single-issue in-order cores. In both cases, we respect the capabilities of the base cores and try to devise techniques that leverage existing resources and mechanisms.

CHAPTER 2

CORE FUSION BASED ON OUT-OF-ORDER CORES*

In this chapter, we present a detailed description of a complete hardware solution to support dynamic core fusion in CMPs of narrow out-of-order cores. In particular, we describe 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. It does so without requiring higher software complexity, a customized ISA, or specialized compiler support.

In the course of formulating our core-fusion solution, we make 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 coher-

*© ACM, 2007. Reprinted, with permission, from "E. Ipek, M. Kırman, N. Kırman, J. F. Martínez. Core Fusion: Accommodating software diversity in chip multiprocessors. In *Intl. Symp. on Computer Architecture (ISCA)*, pages 186-197, San Diego, CA, Jun. 2007. http://doi.acm.org/10.1145/1250662.1250686"

ence 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.

Our evaluation pits core fusion against more traditional CMP architectures, such as fine- and coarse-grain homogeneous cores.

2.1 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 2.1 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.

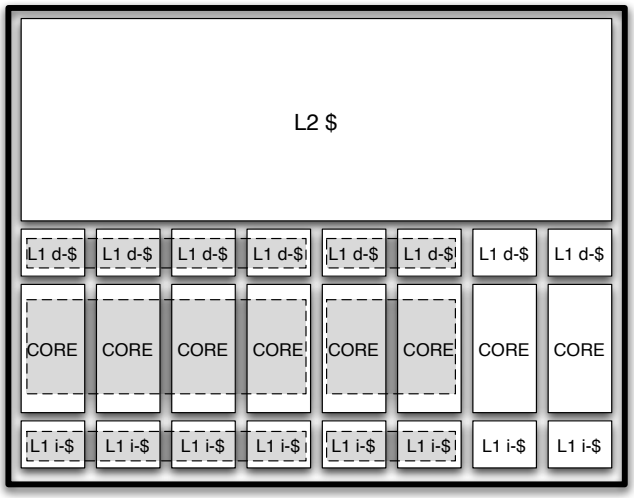

Figure 2.1: Conceptual floorplan of an eight-core CMP with core fusion capability. The figure shows a configuration example comprising two independent cores, a two-core fused group, and a four-core fused group. The figure is not meant to represent an actual floorplan.

2.1.1 Front-end

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 3.1).

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 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 [36]. Each i-cache has enough tags to organize its data in two-word subblocks in fused mode.

During collective fetch, i-TLBs are "naturally" replicated as cores miss on their i-TLBs. 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. 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, these structures are indexed as shown in Figure 2.2 regardless of the configuration. We empirically observe no loss in prediction accuracy when using this "configuration-oblivious" indexing scheme.

Each core can handle up to one branch prediction per cycle. PC redirection (predict-taken, mispredictions) is enabled through the FMU.

Naturally, on a misprediction, misspeculated instructions are squashed in all cores. This is also the case for instructions "overfetched" along the not-taken

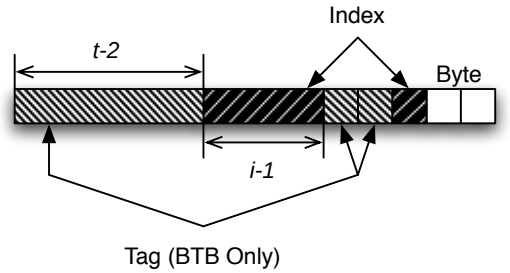

Figure 2.2: Configuration-oblivious indexing utilized in branch prediction and BTB. In the figure, i bits are used for indexing and t for tagging (tagging only meaningful in the BTB). Of course, i and t are generally not the same for branch predictor and BTB. Because of aligned fetch, the two tag bits sandwiched between index bits match the core number in the fused configuration.

path on a taken branch, since the target PC will arrive with a delay of a few cycles.

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 is simply replicated across all cores, and predictions and non-speculative updates are co-ordinated through the FMU.

Return Address Stack. All RAS operations are processed by core zero. Subroutine calls and function returns are communicated to core zero 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."

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 2.3). The steering table and the four rename maps together allow up to four valid mappings of each architectural

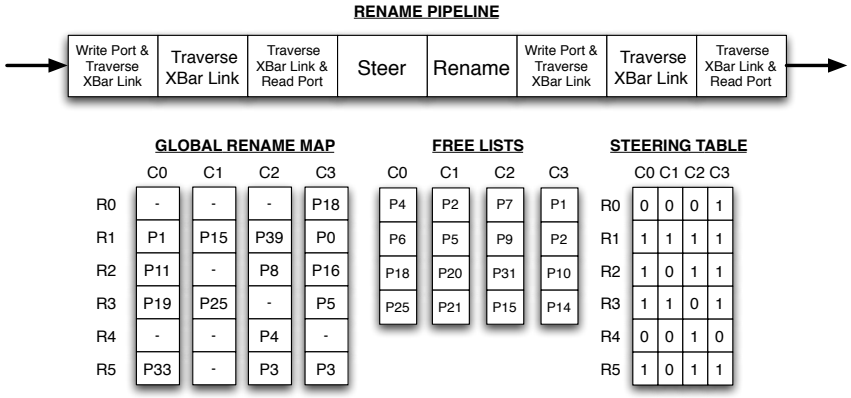

Figure 2.3: Rename pipeline (top) and illustrative example of SMU organization (bottom). R0 has a valid mapping in core three, whereas R1 has four valid mappings (one in each core). Only six architectural registers are shown.

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 2.3 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 3.1). 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 2.1.2).) 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 3), 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 [45] 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 [37, 39]. 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).

2.1.2 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 eight instructions per cycle.

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 2.1.2). In our evaluation (Section 3), we accurately model latency and contention for the operand crossbar, and quantify its impact on performance.

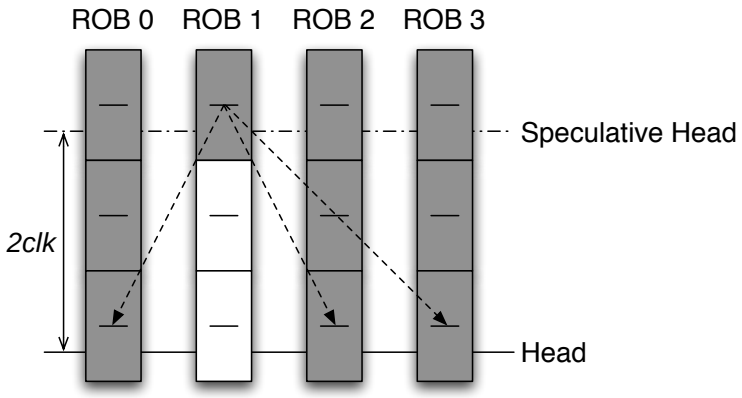

Figure 2.4: Simplified diagram of core fusion's distributed ROB. In the figure, ROB 1's head instruction pair is not ready to commit, which is communicated to the other ROBs. Speculative and conventional heads are spaced so that the message arrives just in time (2 clock cycles in the example). Upon completion 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.

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 lock-step 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 3.4 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 (Figure 2.4). 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 3), 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.

Load/Store Queue Organization

Our scheme for handling loads and stores is conceptually similar to clustered architectures [4, 12, 20, 34, 59]. 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. 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. We attack this problem through LSQ bank prediction [4, 6]. 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. In case the load queue is full at the time the load arrives, it searches the load queue for older instructions. 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.

Fence synchronization operation is dispatched to all the queues. The fence is considered complete once each one of the local fences completes locally.

2.2 Dynamic Reconfiguration

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 via 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. 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.

CHAPTER 3

EVALUATING FUSION OF OUT-OF-ORDER CORES*

3.1 Experimental Setup

We evaluate the performance of 4-way core fusion and compare it against three monolithic configurations: two-, four-, and six-issue out-of-order cores. Table 3.1 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) (Table 3.2). All configurations are clocked at the same speed (this mainly favors the wide-issue cores). Our experiments are conducted using a detailed, heavily modified version of the SESC [43] 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.)

*

Table 3.1: Two-Issue processor parameters.

Two-Issue Processor Parameters	
Frequency	4.0 GHz
Fetch/issue/commit	2/2/2
Int/FP/Ld/St/Br Units	1/1/1/1/1
Int/FP Multipliers	1/1
Int/FP issue queues	16/16
Copy-In/Copy-Out queues	16/16
ROB entries	48
Int/FP registers (Arch.+Ren.)	32+40 / 32+40
Ld/St queue entries	12/12
Bank predictor	2K-entries
Max. br. pred. rate	1 taken/cycle
Max. unresolved br.	12
Br. penalty (min.)	7 cycles (14 when fused)
Br. predictor	Alpha 21264
BTB size / RAS entries	512 entries/ 32
iL1/dL1 size	16 kB
iL1/dL1 block size	32B/32B
iL1/dL1 round-trip	2/3 cycles
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 3.2: Parameters of the shared-memory subsystem.

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 round-trip	320 cycles (uncontended)

Applications

We conduct the simulations on sequential workloads that comprise nine integer and eight floating point applications from the SPEC2000 suite [22]. We use the

MinneSpec reduced input sets [30]. In all cases, we skip the initialization parts and then simulate the applications to completion.[1]

3.2 Hardware Overhead

We compare the areas of all configurations.

Prior work [32, 31, 44, 45] shows that the area overheads of key microarchitectural resources scale superlinearly with respect to issue width in monolithic cores. Burns et al. [9] 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 out-of-order 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 [33],

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

[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.

assuming a 65nm technology and Metal-4 wiring with a 280nm wire pitch [17]. Accordingly, we find the area for fetch wiring (74 bits/link) to be 0.30mm^2, the area for rename wiring (244 bits/link) to be 1.56mm^2, and the area for the operand crossbar (76 bits / link) to be 1.46mm^2. The area of the commit wiring is negligible, as it is two bits wide. This yields a total area overhead of 3.32mm^2 for fusing a group of four cores, or 6.64mm^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 0.68, 0.25, 0.26, 0.23mm^2 per fusion group, respectively, for a total of 2.84mm^2 for the entire chip. Adding these to the wiring estimates, we find the total area overhead of core fusion to be 9.48mm^2. Even for a non-reticle-limited, 200mm^2 die that devotes half of the area to the implementation of the cores, this overhead represents roughly three quarters of the area of one two-issue out-of-order 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 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 [33] 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

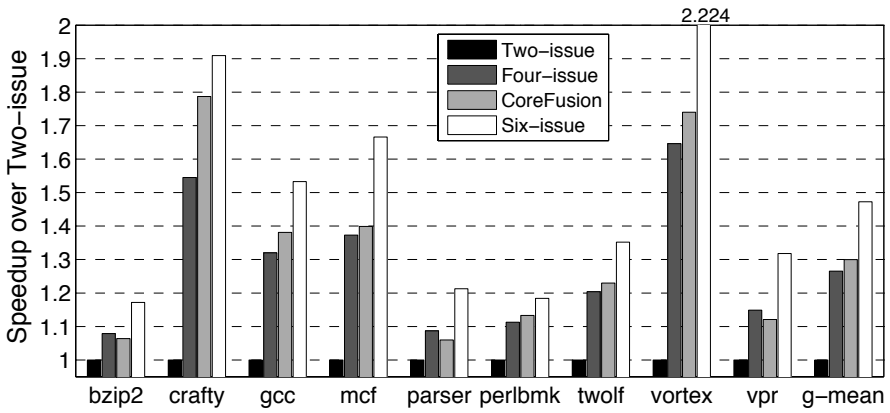

Figure 3.1: Speedup over base core for SPECINT benchmarks.

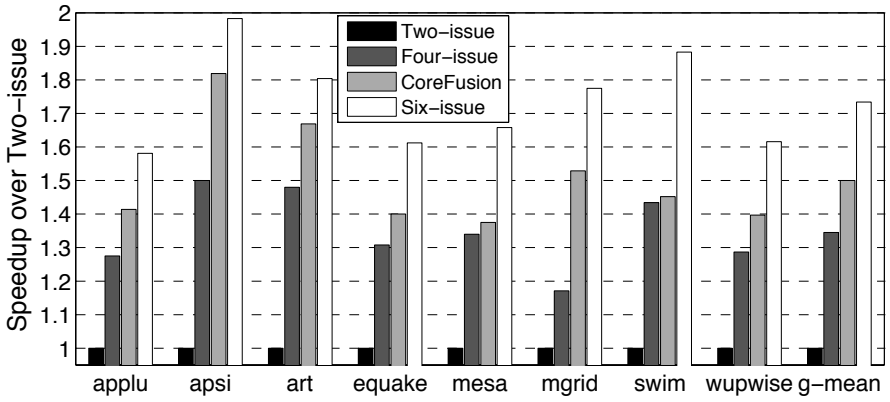

Figure 3.2: Speedup over base core for SPECFP benchmarks.

latency of the rename communication to three cycles (which makes the rename pipeline add up to eight cycles).

3.3 Core-Fusion Performance

Figures 3.1 and 3.2 show speedups with respect to two-issue out-of-order core on SPEC 2000 applications. As expected, the results indicate that wide-issue cores have significant performance advantages on sequential codes. Configura-

tions with a six-issue monolithic core obtain average speedups of 73% and 47% on floating-point and integer benchmarks. (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 two-issue out-of-order core 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, the monolithic six-issue core performs best, followed by Core-Fusion's fused core. 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.

3.4 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. Figures 3.5 and 3.5 show 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, indi-

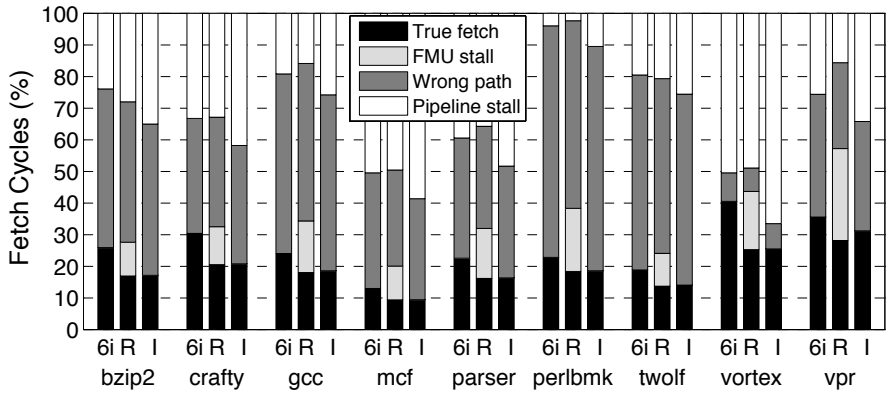

Figure 3.3: Distribution of fetch cycles on SPECINT benchmarks. 6i, R, and I denote our six-issue monolithic baseline, a realistic fused front-end with a two-cycle FMU delay, and an idealized fused front-end with no FMU delay, respectively.

cating that there is significant slack between the fused front- and back-ends. Figures 3.3 and 3.4 illustrate 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 35-95% of its time waiting for the back-end to catch up, and less than 5% of its time communicating through the FMU. On integer codes, 10-60% of the front-end time is spent communicating through the FMU, 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. Figures 3.5 and 3.6 show 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

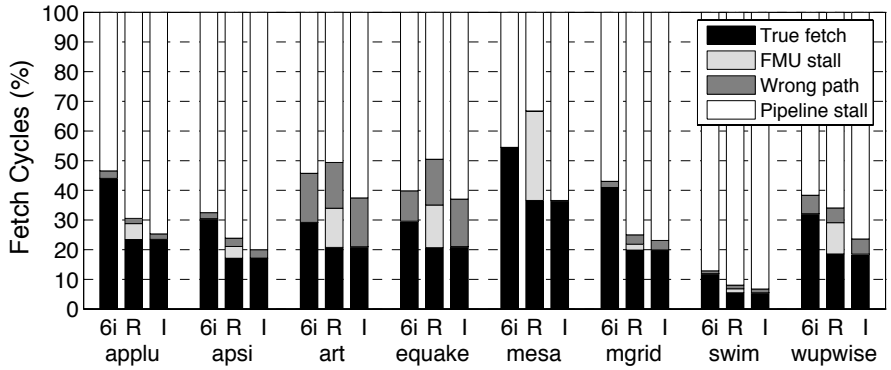

Figure 3.4: Distribution of fetch cycles on SPECFP benchmarks. 6i, R, and I denote our six-issue monolithic baseline, a realistic fused front-end with a two-cycle FMU delay, and an idealized fused front-end with no FMU delay, respectively.

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. Figures 3.5 and 3.6 show 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 lat-

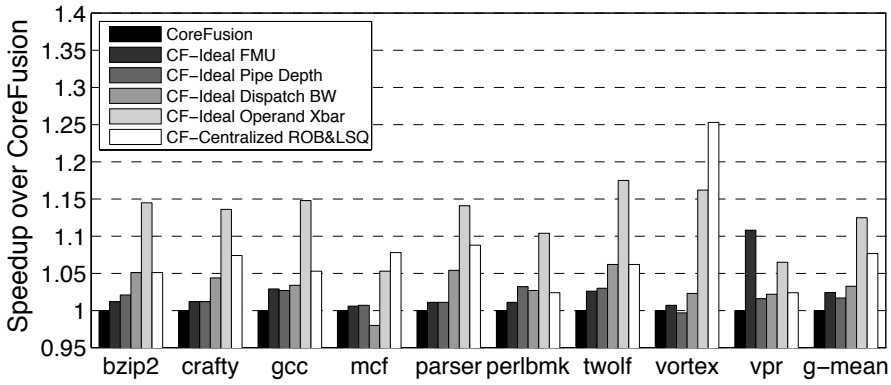

Figure 3.5: Speedups on SPECINT benchmarks when the FMU latency, rename pipeline depth, SMU dispatch bandwidth, operand crossbar delay, or the distributed ROB/LSQ are idealized.

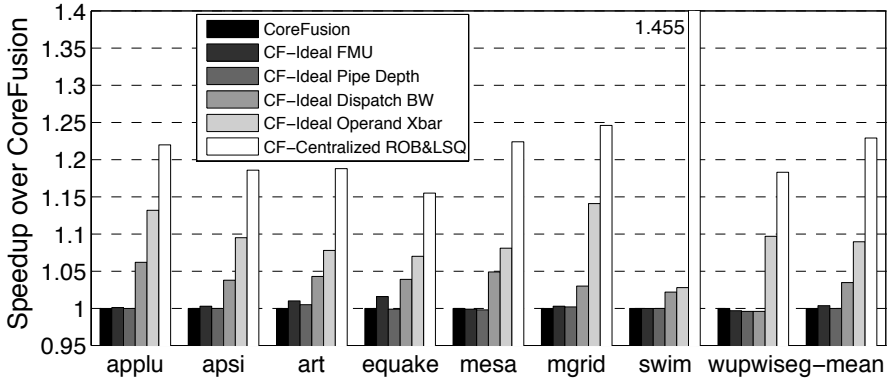

Figure 3.6: Speedups on SPECFP benchmarks when the FMU latency, rename pipeline depth, SMU dispatch bandwidth, operand crossbar delay, or the distributed ROB/LSQ are idealized.

ter 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 inefficiencies that would be absent from a monolithic implementation (e.g., NOP insertion for aligned ROB allocation, and dummy

entry allocation in the LSQ). Figures 3.5 and 3.6 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.

CHAPTER 4

CORE FUSION BASED ON IN-ORDER CORES

In the landscape of multicore chips, in-order cores are quickly gaining relevance–they are extremely power-efficient and simple, and they help maximize core count, which is ideal for exploiting throughput oriented or highly parallel codes. In-order cores already have a significant presence in the server domain, where there is ample thread-level parallelism.

In this work, we propose mechanisms to facilitate dynamic aggregation of small, fundamentally-independent in-order cores to execute sequential code fast. Our hardware-based approach does not change the ISA, and does not require compiler support. Unlike the case of out-of-order core fusion, where the base cores provide the design with valuable opportunities for latency hiding, minimally-provisioned in-order base cores leave little margin for inefficiencies.

In our design, loosely coupled cores process instructions at their own pace, and maintain the program state in a distributed manner. The modular and distributed mechanisms require minimal central processing and functional replication. In the course of formulating our solution, we devise: (1) A distributed checkpoint-based mechanism for bookkeeping of the program state; (2) a distributed renaming mechanism; (3) a lightweight approach for distributed memory disambiguation, and (4) a lightweight lookahead execution within each base core to partially hide cross-core communication and other latencies latencies. Our evaluation shows that a four-way fused configuration delivers a 45% performance gain over a single core, at the cost of 15% hardware overhead per core.

We describe the proposed core-fusion architecture in Section 4.2 after review-

ing the base in-order core in the next section. We quantify the area overhead of core fusion and present its performance evaluation in Chapter 5.

4.1 Base In-order Core

We assume a chip multiprocessor targeted for 32nm technology node that comprises single-issue, in-order, 8-stage-pipeline cores with 16KB private i- and d-caches[1]. Cores are grouped in four, and cores within each group share a unified L2 cache.

Figure 4.1 shows the base core's pipeline structure. Most instructions finish execution in one cycle, while integer multiply/divide, floating-point, and load instructions finish execution in three cycles[2]. The single write-back stage (WB) for all instructions guarantees in-order completion and update of the architectural state (except for load misses). It also simplifies architectural register file (ARF), by requiring a single write port (load misses may require another depending on implementation). All exceptions are caught at WB stage.

Direct jumps are processed at decode, while conditional branches and indirect jumps are resolved at EXE1. Each core has a branch predictor, a branch target buffer (BTB), and a return address stack (RAS). On a misprediction, all newer instructions are flushed, and fetch resumes from the correct target address next cycle.

[1]Using Cacti5.3 [56], we estimate two-cycle access latency for the i- and d-caches, hence the two stages for fetch and d-cache access.

[2]A double-precision floating-point unit in a SPE element of the IBM CELL processor consists of a 9-cycle pipeline and has a cycle time of 11FO4 [40]. Assuming a latching delay of 2FO4, the latency of computation is 81 FO4. Accordingly, in our three-cycle pipeline, we must fit 27 FO4. Using ITRS [26] projections, we estimate that at 32 nm technology and at 4 GHz clock frequency, this amount of computation can easily fit in one cycle.

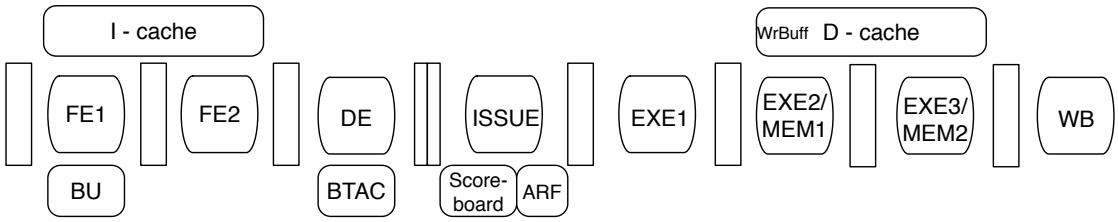

Figure 4.1: Pipeline of the base in-order core.

We assume a unified ARF with a split-cycle write/read implementation [21] (the write-back stage writes during the first half of a cycle, and the issue stage reads operands during the second half). This avoids structural hazards, and it allows consumer instructions to read operands just updated by producer instructions in the first half cycle. The pipeline has all the needed bypass paths to EXE1 from later stages, in order to achieve back-to-back dependent instruction execution.

Effective addresses are calculated in EXE1. Stores are inserted into a FIFO write buffer (no store merging) in MEM1, and are issued to the cache from the buffer when they reach the WB stage and it is known that it is safe to perform the memory update. For loads, cache tag and data array accesses, as well as the check against the write buffer, are performed in parallel in MEM1. Data forwarding from the write buffer, if any, happens in MEM2; also in MEM2 the data array access is completed. Cache misses skip the write-back phase. They generally do not block subsequent L1 accesses; the cache has miss-status handling registers (MSHR).

The core supports weak memory consistency, by ensuring that when a fence operation is at the head of the write buffer, it can complete only when all previous stores have become globally visible. Also, loads cannot issue when there is at least one issued but incomplete prior fence in the pipeline.

The issue stage has a simple scoreboard to track data hazards due to multi-cycle operations. A Ready bit per register is reset when a multi-cycle instruction issues, and set in a timely manner for bypassing. On a load miss, the Ready bit is set back assuming a hit. The RAW and WAW hazards on destination registers for load misses are handled via an Unknown-WB-Time bit per register. When a load issues, this bit is set, conservatively assuming a miss. It is reset either on a hit (on time to enable bypassing), or when a miss is eventually resolved and the result is written back to the ARF. A set Unknown-WB-Time bit for any of the source (RAW) and destination (WAW) registers of an instruction prevents it from issuing.

On a pipeline flush, instructions in the front-end including the issue stage are flushed. If the flush is due to an exception, caught at WB stage, then instructions in the execution stages are allowed to drain, properly setting back the scoreboard bits, but without writing to the ARF.

4.2 Architecture

In this section, we describe the mechanisms to distribute execution of a single instruction stream over a group of four in-order cores, effectively constituting a four-issue processor. We strive for a modular approach that does not replicate functionality or employ complex structures typically seen in out-of-order processors. The following sections detail how we accomplish this. We explain our extensions extensively in detail, however the overall complexity and associated area overhead remain relatively low (Section 5.2) Figure 4.2 highlights the components we introduce on top of a base core.

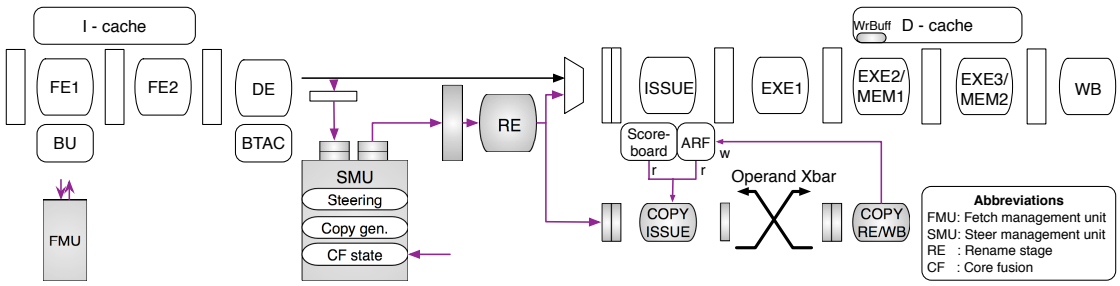

Figure 4.2: Core-fusion support on top of an in-order pipeline.

4.2.1 Distributed Program State

A base core modifies the program state in the architectural register file (ARF) in order at WB stage. A distributed execution, on the other hand, disperses the program state across the ARFs of all four cores. Retrieving a precise program state upon mispredictions or exceptions may not be possible, unless the design includes, for example, a global re-order buffer (ROB), as well as a global ARF or commit-time rename table. Adding such hardware support is not only costly, we believe it is unnecessary.

Checkpoint-based execution, where a processor periodically takes snapshots of the program state, has been proposed as an alternative to ROB-based execution. It enables large instruction windows with relatively limited resources, and has been successfully used to enable speculative execution mechanisms [2, 41, 37, 54, 29].

We leverage checkpointing as a means to enable cost-effective bookkeeping of the program state across cores. It avoids fine-grain global commit of instructions. On any event that requires program-state recovery (e.g. on misspeculation or exception), the last committed checkpoint is restored.

Our design retains a checkpointed program state in the ARF registers holding the program-state values at the time of checkpointing. This pins those registers during the lifetime of the checkpoint.[3]The distributed nature of our checkpoints necessitates a lightweight mechanism to coordinate global checkpoint allocation and recovery. We review the checkpointing support in Section 4.2.4, after we detail the architecture.

4.2.2 Distributed Fetch

We mostly borrow the approach for out-of-order core fusion to achieve collective fetch across cores. Each core fetches one instruction, for up to four consecutive instructions per cycle. Fetch is aligned such that the two lowest bits in the instruction address determine the fetching core. These two bits are not used in indexing predictor tables and caches, but are used in the tag, if any. For out-of-order core fusion, we add extra tags to reconfigure the i-cache into smaller cache blocks in fused mode, to utilize otherwise unused regions. These effectively build wider fetch, larger i-cache and branch predictor from smaller individual core components. In in-order core fusion, however, single instruction granularity requires eight cache-tag arrays if we were to use the same i-cache reconfiguration approach. Our preliminary estimations revealed that this would significantly increase the area overhead relative to a base in-order core. We avoid this by leaving 3/4 of each i-cache block unused in fused mode, effectively having the same aggregate amount of i-cache as in a base core. This simplification is reflected in our hardware-overhead estimation (Section 5.2) and

[3]Because the architectural state is distributed over four ARFs, which provide an aggregate space four times the number of architectural registers, the cores may store checkpoints in place and still have free registers to work with.

evaluation (Section 5.4).

A *Fetch Management Unit* (FMU) coordinates fetch alignment and global branch-history update across cores, which results in one cycle fetch bubble on taken branches and two-cycle bubble on function returns through return-address stack in Core 0.

Due to checkpointed execution, branch predictions are assigned a confidence for checkpoint allocation on low-confidence branches. We use saturated counters of four bits, and confidence threshold of fifteen. Confidence award is one, and penalty is seven. We record branch predictions and their confidence for each checkpoint interval, and reuse them after checkpoint recovery. Resolved branches correct their prediction record and set their confidence to high.

Each core independently decodes its instruction and sends it to a centralized hardware structure that is responsible for instruction steering, named *Steering Management Unit (SMU)*. We assume one cycle link delay to the SMU (Section 5.2). The SMU is the only structure that observes the complete program flow. Therefore, it plays a critical role in satisfying the program order, specifically the register and memory dependences. It coordinates global checkpoint allocation and recovery, and steers instructions to cores. In the following sections, we describe these mechanisms.

4.2.3 Satisfying Register Dependences

We introduce register renaming to allow preserving the checkpointed values in the ARF registers, efficiently utilize registers, and correctly track dependences.

We propose a distributed renaming technique to reduce coupling between SMU and cores, and simplify the SMU and rename logic. Each core independently and locally maps logical source and destination registers to its ARF registers; the SMU does not manage register allocation and release, nor does it store any logical-to-physical mappings. We do not increase the size of each ARF–the distribution of the architectural state over the four cores automatically creates space in each ARF for renaming.

Distributed register renaming

Each core is augmented with a rename stage that comes prior to the issue stage (Figure 4.2). It renames instructions coming one at a time from the SMU, and inserts them into the regular issue queue for execution.

A Local Rename Table (LRT) in the rename stage maps logical registers to the local ARF registers. Different from typical rename tables, some LRT entries might be invalid, and some valid mappings might be stale if a logical register is more recently mapped in another core. Only the SMU knows and keeps track of the core that has the most recent mapping for each logical register in a table.

Only local source operands locate their ARF register via the LRT. A remote source operand (appropriately tagged by the SMU) allocates a new local ARF register from the free-register list without updating the LRT. The remote value will be brought into the register by a *copy instruction* injected by the SMU into the sourcing core. The ARF register identifier is recorded in a small table called Remote Operand Table (ROT).The ROT is a RAM array, indexed by a *remote-operand id* that is assigned by the SMU at the time of steering. The id is also at-

tached to the corresponding copy instruction, so that the two instructions match on a common ROT entry. When the copy instruction arrives at the destination core, it checks the ROT entry to learn the ARF register to copy over.

It is possible that the copy instruction arrives before the regular instruction is renamed. In this case, which is detected through invalid ROT entry, the copy instruction allocates an ARF register and records it in the ROT. The regular instruction finds the entry valid and uses that register to rename its remote source operand. When the ROT entry is consumed by the second instruction, the ROT entry is invalidated.

The size of the ROT is large enough to accommodate the maximum number of remote operands in the issue queue.[4] SMU generates remote-operand ids per core using a wrap-around counter. Free-ROT-entry assignment is guaranteed by (1) securing a free issue-queue entry before dispatching an instruction and its copy instruction(s), and (2) releasing issue queue entries in order, already being the case.

Finally, renaming a destination register simply allocates a free ARF register and updates the LRT.

Because of renaming, an ARF register's Ready bit in the scoreboard is reset at the time the register is allocated from the free-register list. For locally produced registers, it is set in the main pipeline as in the base core. For remote-operand registers, it is set by the copy instruction upon value delivery.

Because the SMU does not know a priori whether there are free registers in the cores, it is possible that an instruction may be unable to secure an ARF regis-

[4] In an alternative implementation, the SMU can constrain the number of remote operands per core.

ter during renaming. In this case, the instruction is marked as having raised an exception. At WB stage, the failure will be communicated to the SMU as a *Rename Failure*, and it will cause the architecture to roll back to the last committed checkpoint. When a rename failure occurs on copy-induced register allocation, the failure is propagated to the consumer instruction through a flag in the ROT entry. It is easy to guarantee forward progress on rename failures, by making cores send a *Free-Register Threshold* signal to the SMU when the number of free ARF registers drops below a predetermined threshold. Upon receiving such a signal by any core, the SMU initiates checkpoint allocation, which guarantees absence of rename failures up to that point (baring mispredictions or exceptions). We dynamically adjust this threshold between an aggressive and a conservative one based on the occurrence of rename failures, to minimize rollback events. The SMU also takes measure to prevent mapping too many registers to a core (Section 4.2.8).

Remote-operand transfer flow

This section details the copy-instruction flow (Figure 4.3). At a high level, the SMU dispatches a copy instruction to the source core where the remote operand value currently resides. Upon reading the value, the copy instruction delivers it to the core of the instruction with the remote operand, over a cross-core full crossbar.

Copy instructions share the same path with the regular instructions from the SMU up to and including the rename stage in the source core. This ensures properly ordered renaming. Because their source operand is always local, they use the LRT to locate the ARF register. Afterwards, they are placed in a FIFO

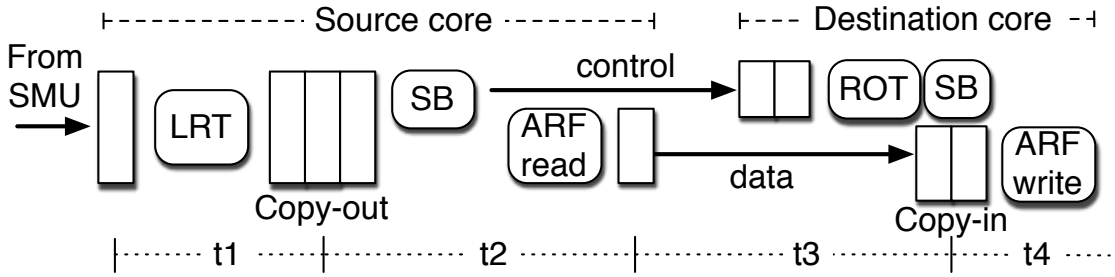

Figure 4.3: Copy-instruction flow through the source and destination cores. SB is short for scoreboard.

RAM *copy-out* queue (separate from the issue queue) for issue. The copy instruction at the head of the copy-out queue checks the scoreboard for operand readiness, in the same way regular instructions do. When it issues, it reads the ARF[5], and goes to its destination core where it is placed in a *copy-in* queue. Processing copy instructions at the source core, thus, requires an additional read port to both the scoreboard and the ARF.

We assume that the remote-operand id precedes the data transfer, to make room for (1) multiplexing among copy instructions from other cores, (2) locating the ARF register through ROT (Section 4.2.3), and (3) setting the Ready bit of that register so that a consumer instruction can issue earliest next cycle, all in the destination core.[6] When issued from the copy-in queue, the copy instruction updates the ARF register and terminates. The aforementioned operations require one extra write port to the Ready bit array. Copy-in queue accesses use the extra write port originally reserved for load misses.

Renaming and proper register release (Section 4.2.5), together ensure that no WAW and WAR hazards occur between instructions in the copy queues and

[5]We assume proper bypassing from the main pipeline.

[6]In rare cases, the Unknown-WB-Time flag of the register is set. This cancels the write-back of the copy instruction. Eventually, when the load miss resets the flag, the copy instruction resumes. This requires one read port to the Unknown-WB-Time array.

regular issue queue. (Recall that, checking the scoreboard for operand readiness handles the RAW hazards.)

4.2.4 Checkpoint Allocation, Commit, and Recovery

Our preliminary studies showed that a lightweight checkpoint management that does not block instruction flow is critical to performance. This led us to implement a novel distributed and nonblocking checkpointing approach.

Checkpoint allocation and commit

To allocate a checkpoint, the SMU generates a special *Checkpoint* instruction, which is dispatched to all cores. These are interleaved with ordinary instructions cutting them in waves. A *Checkpoint* passes through the main pipeline in a core. It checkpoints the LRT at the rename stage. At this time, it is guaranteed that all instructions prior to the checkpoint have updated the LRT. Only valid and non-stale mappings need to be checkpointed, which are conveyed by the SMU along with the *Checkpoint* instruction. The checkpointed registers are pinned and not allowed to be released/overwritten until the checkpoint is released. On successful completion at WB stage, it notifies the SMU in order, so that the SMU properly orders multiple in-flight checkpoint allocations and determine the final action for each. The SMU deems a checkpoint committed when it receives the acknowledgements from all cores. Then, it sends a *Commit* signal to all cores. Each core commits the oldest non-committed checkpoint, and releases the last committed one. Because checkpoints are handled at WB, they are committed in order.

Checkpoint recovery

A mispredicted or excepting instruction in a core communicates a global flush request to the SMU at WB stage. This orders flush requests with respect to *Checkpoint* completions. Upon receiving a flush request, the SMU first waits all prior checkpoints to commit (baring any flush request prior in program order, necessarily from another core). At this point, the SMU cancels more recent checkpoints and flushes the cores. A worst case delay allows any in-flight copy instruction in the operand crossbar to be flushed as well. On receiving a complete set of flush acknowledgements, the SMU signals *Checkpoint Recovery* to cores, which instructs each core to restore the last committed checkpoint, clear the ROT, and properly initialize the free-register list. The SMU also restores its logical register-to-core mapping from its own checkpoint. Note that fetching the new instruction stream overlaps with checkpoint recovery.

Committing speculative memory updates

We keep speculative memory updates off the caches and memory. Stores remain in the write buffer and are not issued to the memory system until they are committed. A write buffer maintains a speculation level (SL) (initially 1 in fused mode), whose value is attached to a write-buffer entry upon allocation. A *Checkpoint* instruction increments the SL as it passes through the pipeline. On a checkpoint commit, the SL as well as the non-zero SL fields in the write-buffer entries are decremented. Entries whose SL becomes 0 are effectively committed. This commits all speculative stores prior to the commit point. On a flush, all speculative entries are invalidated.

When write-buffer occupancy reaches a threshold, the core sends a commit request to the SMU. If a store finds the write buffer full with speculative stores, and no checkpoint commit is expected (speculation level is still 1), the store is let to issue and go through the pipeline to request a flush at WB stage.

4.2.5 Register Recycling

In our distributed register management, each core recycles its ARF registers independently using only local information. The SMU is not directly involved in register recycling.

A register can be released when (1) all consumers have read it, and (2) it is not part of any checkpoint.

First, we identify after which local instruction there will be no more consumers in program order to an ARF register. An instruction whose destination rename mapping overwrites a valid LRT mapping safely indicates the end of consumers for the ARF register in this previous mapping. In addition, an instruction with a remote operand is the only consumer of the register allocated for it.[7] (This does not apply if the register is made usable to subsequent instructions through a register replication optimization (Section 4.2.9).) Also, after a local instruction that identifies a stale logical-to-ARF register mapping, there cannot be subsequent consumers of that ARF register, since any subsequent instruction will necessarily use the more recently mapped register in another core. There are two occasions for identifying a stale mapping. First, when an instruction renames its remote source operand, the current local mapping for

[7] Recall that the mapping is not present in the LRT.

the register in the LRT, if valid, must be stale. Second, when a *Checkpoint* instruction presents the list of logical registers to be checkpointed at rename stage (Section 4.2.4), the remaining valid mappings in the LRT must be stale as well. (Once identified, stale mappings are invalidated.) As a result, at rename stage, each instruction identifies the registers it can recycle later.

The in-order pipeline guarantees that, by the time an instruction gets to the WB stage, its recyclable registers have been read by all the local consumers that went through the main pipeline. However, a preceding consumer copy instruction may still be outstanding in the copy-out queue in the core. Therefore, we track the outstanding copy instructions using a consumer counter per ARF register (incremented when a copy instruction is renamed, and decremented when a copy instruction issues from the copy-out queue).

As a result, a register is recycled when (1) its recycling instruction reaches the WB stage, (2) its copy-consumer counter is zero, and (3) it is not a part of any checkpoint.[8] Because an instruction can recycle multiple registers, we simplify tracking (1) by allowing one or multiple instructions to accumulate their recyclable registers on a *to-be-recycled* bit vector at rename stage, and letting the last of these instructions to mark these registers at WB stage into a final *ready-to-recycle* bit vector where they wait other recycling conditions to be satisfied. The subsequent instructions must accumulate on a different *to-be-recycled* bit vector. Hence, we provide multiple (at least two) of these vectors. We put charge on an instruction whenever there is a free vector.

[8]In cases where a load-miss result register has no consumers, and it is recycled and reallocated before the result is actually written back, the Unknown-WB-time bit in the scoreboard will prevent any data hazard on this register. For this reason, copy instructions in copy-in queue also need to check this bit before they can write to a register.

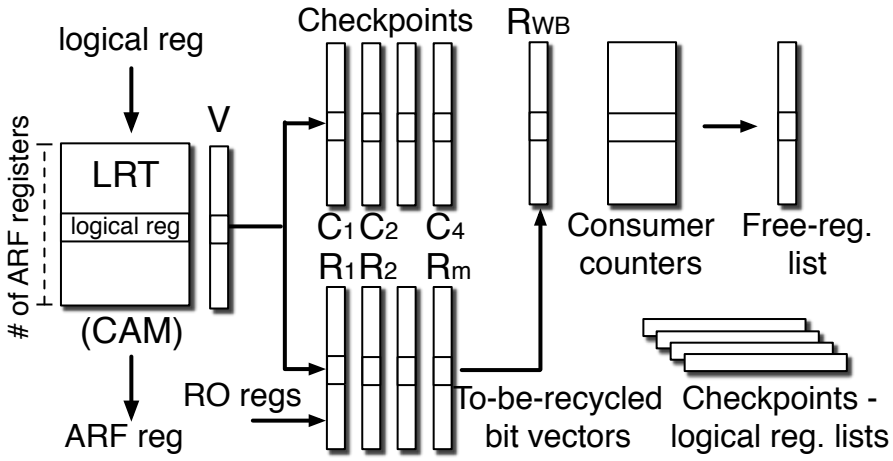

Figure 4.4: A core's CAM-based LRT, checkpoints of its Valid (V) bits (C_1, .., C_4), and register recycling support (*to-be-recycled* bit vectors (R_1, .., R_m, and R_{WB}), copy-consumer counters, and free-register list.) A core also keeps the list of logical registers in each of its checkpoints. The figure is not illustrative of port counts. RO is short for remote operand.

4.2.6 Hardware Support for Register Renaming, Checkpointing, and Release

We would like to elaborate on our preferred hardware implementation for register renaming, checkpointing and release (Figure 4.4).

Among the most common SRAM- and CAM-based rename-table implementations [45], we see the CAM-based one to be advantageous in our architecture. A checkpoint of the rename table is simply the copy of the Valid bits in the rename table. A SRAM-based implementation, on the other hand, would require checkpoint storage for full mapping table, while only some of the mappings are actually relevant to the checkpoint. Based on the results of a recent work [50] that compares the two approaches, we expect a CAM-based implementation of

our size and port requirements to be comparable to a SRAM-based one in terms of delay and energy.

Having the LRT, its checkpoints, *to-be-recycled* and *ready-to-recycle* vectors, copy-consumer counters, and the free-register list, all organized as arrays with entries per ARF register simplifies the relevant operations. The only non-trivial implementation issue is as follows. On a *Checkpoint* instruction, ideally only Valid bits for non-stale mappings (Section 4.2.5) need to be checkpointed. The remaining Valid bits should be invalidated, and the corresponding bits in the current *to-be-recycled* vector should be set. The limited number of look-up ports in the LRT renders a one-cycle implementation of locating all the stale ARF registers impractical. We achieve this over time, by first checkpointing all valid bits, and then lazily identifying the stale mappings by checking against checkpoint's list of logical-registers.[9] Any lazy approach that identifies a stale mapping should invalidate the LRT mapping, clear the checkpoint bit, and set the *to-be-recycled* bit of the ARF register. A lazy approach does not compromise correctness on checkpoint release or recovery, as stale mappings are already not visible/usable by any subsequent instruction. It only increases register pressure. (Such issue also arises in a SRAM-based LRT implementation.)

4.2.7 Satisfying Memory Dependences

We assume a widely adopted cache hierarchy for chip multiprocessors. Fused cores have private write-through and write-no-allocate L1 d-caches. All share

[9]We perform the check and clear the checkpoint bit of a stale mapping when it is already invalidated/overwritten in the LRT by a regular instruction at rename. This avoids any extra look-up port to the LRT. We also use the idle look-up ports on a flush to identify and free any remaining stale mapping in the restored checkpoint.

an L2 cache. The L2 cache sends invalidations to the other L1 caches when it processes a write from one of the L1 caches.

Stores: We propose a simple mechanism to handle memory dependences without introducing load queues. Stores are replicated to all cores at the time of steering. In a core, each replica performs address calculation and is inserted into the write buffer as usual. Once committed, each store replica is issued to the memory system. A store replica that hits in the L1 cache updates the cache line and is always issued to the L2 cache, as is the operation of a write-through cache. Store replicas from different cores reach the L2 cache in the same order. However, allowing all stores to perform to the L2 cache may violate correct store ordering due to any slip between cores. Therefore, we allow only the leading store replica to perform in the L2 cache. The subsequent replicas are directly acknowledged back upon reaching the L2 cache.[10] Tracking this is straightforward: The L2 cache keeps four store counters, one per core. A store arriving from a core is processed only if its count (after the store is accounted for) is larger than those of others–this is the *leading* core. Stores by trailing cores are accounted for and simply acknowledged. Counters are all decremented by one whenever their counts all become nonzero–the important thing is the difference between counts and not the counts themselves. Note that, the L2 cache need not send invalidations on any of the store replicas, all L1 caches already see all the stores. Fortunately, write-no-allocation policy reduces the cache-line replication across the L1 caches due to store replication.

To reduce the number of copy instructions needed to communicate address and data operands to store replicas, we use a special multicast copy instruction that can encode multiple destinations (core, remote-operand-id pairs). After

[10] A write-no-allocate L1 cache does not expect the cache line on stores.

reading the source register, such copy instruction multicasts the value along with the proper remote-operand id over the operand crossbar.

Loads: Because all L1 caches see all the stores in the program, load instructions can be steered to any core. This improves load balance and operand locality. Loads that hit in the write buffer or L1 cache complete. Load misses, on the other hand, necessarily can only be dependent to stores in prior checkpoint intervals. Stores in the same checkpoint interval, if any, have to be in the write buffer. If a load miss reaches the L2 cache before any younger store replicas from other cores, either from the same or a later checkpoint interval, it will read the correct value from the L2 cache. However, if these younger stores are to be committed and released to the memory, the load miss will race with them in reaching the L2. To prevent a younger store replica to the same cache line from updating the L2 cache before the load miss, cores wait all primary load misses in all L1 caches at the time of a checkpoint commit to reach the ordering point in L2 cache before releasing the stores committed by the checkpoint to the memory system.[11] (Secondary misses obtain their value from the cache line brought by a primary miss.)

Memory fences: Finally, a memory fence instruction is always replicated to all cores. Each replica ensures the fence operation is satisfied in a core, which is acknowledged to the SMU. Upon receiving a full set of acknowledgments from all cores, the SMU signals the global completion to all cores. A forward progress issue arises when a load instruction comes after the fence with no intermediate checkpoint allocation (i.e., commit point). The speculative fence and older stores will be released from the write buffer on a subsequent checkpoint commit, but such a checkpoint allocation cannot progress in the pipeline because the issue

[11] We do not block issue of new load instructions.

stage will block on the load that waits the fence to complete. We break this cycle by always allocating a new checkpoint right after steering a fence instruction.

4.2.8 SMU Organization

With the discussed mechanisms in mind, we detail the SMU structure and its policies in our architecture.

Mentioned a few times by now (Section 4.2.3), the SMU maintains logical register-to-core mappings in a table, which it uses to steer instructions. In addition, it has an output buffer per destination core that is logically partitioned to a copy-instruction and regular-instruction space. The dispatch bandwidth to a core is restricted to a single instruction per cycle.

The SMU steers collectively-fetched blocks of instructions, one block at a time. It uses dependence-based steering policy [45], except for store and fence instructions which are replicated to all cores (Section 4.2.7). An instruction is steered to the core where the producer instruction of any of its source operands has been lastly dispatched. If this is not possible, it goes to an output buffer with empty regular-instruction space.[12] If that is not possible either, the steering stalls. After a steering decision and copy-instruction generation (if any), SMU stall is inevitable if there is no space in a target output buffer or credit for core-side issue/copy-out queue.

The SMU initiates a new checkpoint allocation in the following cases: (1) When a core signifies a reached write-buffer threshold (Section 4.2.7) or free-register threshold (Section 4.2.3), (2) following the first branch after a flush, to

[12]The original steering policy in [45] looks for an empty issue queue.

ensure forward progress, (3) preceding a low-confidence branch, (4) if it has been more than 32 program instructions since the last checkpoint, (5) after a fence instruction (Section 4.2.7), and (6) when transitioning into and out of fused mode. (1), (3), and (4) do not stall the SMU if there is no available checkpoint.

In addition, the SMU occasionally (just before allocating a new checkpoint on exceeded free-register threshold in any of the cores) performs register balancing through move instructions to keep logical registers mapped to a core under certain threshold. A move instruction (e.g. *mv rx, rx*) and its copy instruction effectively transfers a logical register (*rx* in the example) from the current core to another less loaded core. The move instructions update the logical register-to-core mapping table as well.

Finally, when the SMU encounters an instruction that should not be executed speculatively (e.g. I/O operation), the execution transitions to a special mode in a single core. For this, the SMU first allocates a new checkpoint, waits for its commit, and issues move instructions to gather the program state to the first core. In this mode, instructions are always steered to the first core. When it is safe to resume normal fused mode, the SMU first drains the pipeline, issues fence instruction to ensure all stores take effect in the other caches, re-distributes the state among cores via move instructions, and allocates a new checkpoint.

4.2.9 Performance Enhancements

We now discuss several optional techniques that improve performance significantly, but add relatively little design effort and complexity over the base core-fusion architecture.

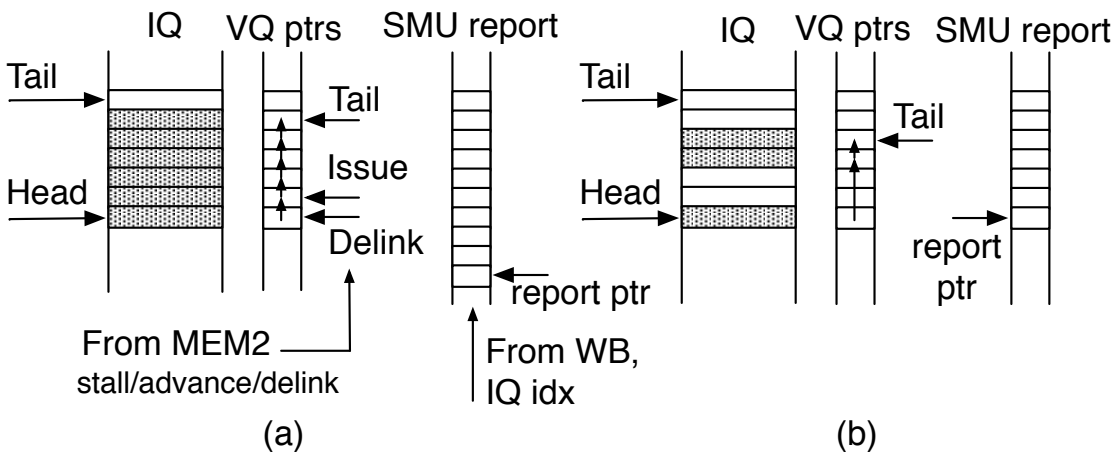

Figure 4.5: Lookahead execution (LE) support: virtual queue (VQ) and in-order report to the SMU. Issue-queue (IQ) entries in gray hold non-issued instructions. (a) depicts the initial pointers in a LE, while (b) depicts the VQ on exiting the LE. VQ Issue and Delink pointers will be re-initialized on a subsequent LE.

Lookahead execution

Lookahead execution (LE) mode inside a core aims to hide remote-operand latencies and load misses, without resorting to typical out-of-order processing support for wake-up, select, etc. It tries to issue some of the instructions ahead at the processing rate of a base core, utilizing otherwise idle resources, while waiting for a blocking event to resolve. It comes with little cost on top of the base core-fusion architecture. Specifically, register renaming and free ARF registers handle register dependences. The global checkpointing support already maintains the program state. We take a conservative approach for handling memory disambiguation. As a result, the main extensions are related to selecting an instruction from the issue queue, and preserving the in-order communication to the SMU. Eventually, the mechanism is transparent to the SMU and other cores. The required support in a core is depicted in Figure 4.5.

Entering lookahead execution: The mode is entered on an issue stall due to a non-ready remote operand or dependency to a miss load. The mode is entered only if there are other non-issued instructions.

Instruction selection: We reuse the base FIFO RAM issue queue. A *virtual queue (VQ)* of non-issued instructions is constructed using an auxiliary RAM array extending the issue queue. VQ essentially tracks these instructions in a linked list. An entry keeps the issue-queue index of the next instruction in the list. On inserting an instruction into the issue queue, it is also linked to the tail of the VQ[13], regardless of the execution mode. Notice that, *IQ Head* is also the head of the VQ.

Upon entering LE mode, a *VQ Delink* pointer is initialized with *IQ Head*. A *VQ Issue* pointer is initialized to the next entry in the linked list. At a high level, instructions in the VQ are selected for issue one at a time by *VQ Issue*, which traverses the linked list. On a successful issue, the issued entry is delinked using *VQ Delink*[14]. For each issue trial from the VQ, *VQ Delink* either delinks, stalls, or advances, effectively following the *VQ Issue*, however delayed by a few stages for a reason explained below.

Memory disambiguation: We conservatively do not allow store and fence instructions to issue during LE. We experimented with also not issuing loads if there is a prior store or fence in the VQ. However, this resulted in not so great performance improvement. To aid in memory disambiguation, we leverage the knowledge that store addresses can be calculated typically much earlier than their data. As a result, we "pre-issue" stores (with no delinking) for the

[13]This is simply achieved by writing the current *IQ Tail* value into the entry pointed by *VQ Tail*. Then *VQ Tail* gets this value as well.

[14]Notice that *VQ Delink* pointer is used to delink the next entry in the VQ.

purpose of address calculation only, pre-filling a write-buffer entry as well. Pre-filling uses an auxiliary write buffer tail pointer which is reset to the actual tail pointer on entering/exiting LE mode. In case a store cannot resolve its address, subsequent loads are not issued. If an issued load matches any of the pre-issued stores in the write buffer, the load operation is cancelled; hence is the need to delay delinking from the VQ. Otherwise, the load completes successfully. Finally, we do not allow a *Checkpoint* instruction to issue in LE mode if there is a prior store or fence in the VQ, not to update the speculative level in the write buffer while there is still non-issued prior stores (Section 4.2.4).

Typically, *VQ Issue* skips an instruction that cannot issue, advancing to the next instruction in the VQ. However, it stalls on instructions whose non-ready operands are likely to be resolved soon through local bypassing.[15] Nonetheless, *VQ Issue* can eventually reach *VQ Tail*, unless LE terminates earlier.

In-order report to the SMU: WB stage is not suitable anymore to report to the SMU any flush request, *Checkpoint* acknowledgment, or exception (Section 4.2.4). This is also the case for register recycling (Section 4.2.5).

We introduce a second RAM array, again extending the issue queue, to temporarily save these information for each instruction passing through the WB stage. Clearly, entries can be updated out of order. However, they are processed in order: Any relevant information is sent to the SMU, registers are marked ready-to-recycle if indicated (Section 4.2.5), and the issue queue entry is released by sending credit back to the SMU. We employ this mechanism regardless of the execution mode.

[15]We wait the Ready bit of the local operands to be set, except if the producer is still non-issued. We track the latter by annotating the destination of the previously skipped instructions while issuing from the VQ in an additional bit vector.

Exiting lookahead execution: LE mode is terminated either when the end of the VQ is reached by *VQ Issue*, or earlier if the blocking event that triggered LE resolves. In-order issue resumes from *IQ Head*, which advance on the linked list to skip already issued instructions. Before another LE can start, any delayed delinks are awaited a few cycles to complete.

Register replication

A simple extension to our design allows increasing source-operand locality, which in turn reduces the copy instructions and their associated overhead.

Recall that a register allocated for a remote operand is not reflected in the LRT mapping, as well as in the logical register-to-core mapping in the SMU (Section 4.2.3). Thus, this register replica is unusable by subsequent instructions. By simply having the SMU and LRT reflect the new mapping for a remote operand, a logical register can be made globally visible in multiple cores. For this, the SMU tracks multiple core locations per logical register using a bit vector, instead of a single core id. It also annotates the remote operand to update the LRT as well. Notice that, on a checkpoint allocation, replicas will be included in the partial checkpoints in cores. This technique, though, increases register pressure. We alleviate it by allowing only at most one register replication per instruction. This adds only one write port to the mapping tables per replica.

When the SMU performs register balancing for a core (Section 4.2.8), it clears the extra core mappings before generating move instructions if still necessary.

Copy-out queue optimization

Since copy instructions in the copy-out queue are all register reads, in principle, they can issue in any order with respect to each other, even if queue entries are released in order. We have observed a significant performance gap between a strictly in-order issue and an ideal selection that always picks the oldest ready instruction for issue. We propose a simple mechanism to partially bridge this gap. We increase the candidate copy instructions for selection to three–the two oldest non-issued copy instructions in the queue plus the new incoming copy instruction that bypasses the queue. The oldest one is already at the head of the queue. Then, we use a second pointer on the copy-out queue that points to the second non-issued instruction (initially the next instruction). If the instruction at the head pointer issues, the head pointer gets the value of the second pointer, and the second one is advanced one entry. If the instruction at the second pointer issues, then that pointer advances to the next entry, while the head pointer remains the same. In case none of the two instructions can issue, the pointers are preserved. In Section 5.5.2, we evaluate these selection polices.

All three instructions check the scoreboard for operand readiness, but only the oldest ready one issues. These operations require two additional read ports to the scoreboard, and an extra read port in the copy-out queue.

CHAPTER 5

EVALUATING FUSION OF IN-ORDER CORES

5.1 Experimental Setup

We evaluate a 4-way core-fusion architecture (*ioCF*) and compare it to three other configurations: single-, dual-, and quad-issue in-order cores, which we refer as *BaseCore*, *IO-2i*, and *IO-4i*, respectively. Core and memory system parameters are summarized in Table 5.1. Resource counts/sizes within cores are increased proportionally to issue width, except for write buffer and RAS. The memory system below private caches is the same for all configurations. Cache access latencies are estimated using CACTI 5.3 [56]. Parameters specific to *ioCF* are provided in Table 5.2. Notice that each core in ioCF has an issue queue size of 24, whereas the single-issue baseline has eight issue queue entries. A larger issue queue is essential to the success of our lookahead execution, and we account for that area overhead properly. We did verify that providing the base cores with a 24-entry issue queue did not yield significative performance benefits, and thus provide the baseline with the configuration that yields the smallest area overhead.

We use the SESC [43] simulator to model all configurations. We do not model wrong-path execution due to branch mispredictions. However, we model instruction re-fetch and re-execution (and the associated time penalty) due to flushes in *ioCF*.

Table 5.1: Core and memory-system parameters. In the table, FX, GHR, BHR, BTB, RAS, MSHR, WrBuff stand for fixed-point, global history register, branch history register, branch target buffer, return address stack, miss-handling status register, and write buffer, respectively. Cycle counts are in processor cycles.

In-order Cores			
	1-issue	2-issue	4-issue
Frequency	4GHz	4GHz	4GHz
Branch min. cycles	5	5	5
Issue queue entries	8	16	32
FX/Branch units	1	2	4
FP / Ld/St units	1/1	2/1	4/2
WrBuff entries/fwd. delay	24/2	24/2	24/2
IL1/DL1 cache size	16KB	32KB	64KB
MSHR entries	8	16	32
Branch predictor (Hybrid of GAg + SAg)			
GHR/BHR bits	12/10	13/10	14/10
BHR entries	1K	2K	4K
Chooser entries	4K	8K	16K
BTB/RAS size	2KB/32	4KB/32	8KB/32
	L1	L2	L3
Cache size		2MB	32MB
Cache associativity	4	8	16
Cache access cycles	2	9	70
Cache writeback policy	WT	WB	WB
Cache block size	32B	64B	64B
Cache MSHRs		32	32
Stream prefetcher	16 streams, 1 depth		
Memory bus bandwidth	1×64GB/s		
Memory latency	200 cycles		

Applications

We evaluate sequential-thread performance of *ioCF*, hence we use we use ten integer and nine floating-point applications from the SPEC CPU2000 suite [22] (our simulation infrastructure currently does not support the remaining applications). We use MIPS binaries compiled with -O3 optimization level. We fast-

Table 5.2: *ioCF* specific parameters. Cycle counts are in processor cycles.

Core Fusion Parameters	
Extra pipeline stages	4
SMU steering width	4
SMU output buff. size	3 copy + 3 inst
SMU to core BW	1 inst/cycle
Checkpoints	4
Branch confidence estimator	4K-entry
Branch confidence threshold	15/15, -7 penalty
Operand Xbar latency	1 cycle
FMU latency	1 cycle
Issue queue size	24
Copy-out queue size	12

forward two billion instructions to pass the initialization phase (after which we start modeling timing and collecting statistics) and execute one billion committed instructions. We use *ref* input data sets.

5.2 Area and Delay Estimations

We estimate the areas of all configurations using the methodology described below. We list the end results in Table 5.3.

Table 5.3: Area-estimation results for in-order cores as well as the core-fusion architecture.

	Relative Area
BaseCore	1 ($0.3mm^2$ @32nm)
IO-2i	1.98
IO-4i	5.12
ioCF	4 cores + 0.60

Area estimation: We estimate the areas of storage arrays using CACTI5.3 [56] for 32nm technology node, carefully setting their input/output

port requirements. These include L1 i- and d-cache, i- and d-TLB, branch-predictor components, decode ROMs, issue queue, register file and scoreboard, and write buffer. On top of these, we add the areas of fixed-point execution unit(s) and double-precision floating-point unit(s) extrapolated from data on SPE element of CELL processor [40, 35]. In addition, we account for a PC logic and branch-target-address-calculator area per fetched instruction, equal to the area of an adder. We also assume that 10%, 15%, and 20% of the core area with no caches is other related logic and latches for *BaseCore*, *IO-2i*, and *IO-4i*, respectively, as superscalar designs introduce increasing logic and latching complexity.

We find the base core area to be $0.3mm^2$ at 32nm process technology. (*IO-2i* and *IO-4i* turn out to be 1.98 and 5.12 times larger than the base in-order core.) We estimate that an ITRS [26] projected $310mm^2$ die can fit 92 cores plus 23 L2 caches (one for every four cores), assuming half of the die is dedicated to these, leaving enough additional space for other components including interconnect, L3 cache and memory controllers. We find the L2 cache area using CACTI5.3.

For the core-fusion design, we use updated port counts for the storage arrays, and add the areas of newly introduced structures within each core and globally shared. For global structures, we account for communication fabrics to/from FMU and SMU, operand crossbar, and array structures in SMU such as the logical register-to-core mapping, its checkpoints and output buffers. In wiring-area estimations, we assume 96nm minimum global-wire pitch [26] and that wires start/end at half width of a core.

The core fusion overhead corresponds to a 15% area increase per core, or 60% over four cores excluding their shared L2 cache. Of that 60% aggregate

overhead, 35% is due to the shared structures, with the operand crossbar being the largest contributor. Within a core, the major contributor is the extra read port in the register file. The other significant overheads come from the confidence estimator in the branch predictor, larger issue queue, and the LRT. When core-fusion support is added, the same $310mm^2$ die area can fit 88 cores plus 22 L2 caches.

Delay estimation: We estimate global wire delay assuming 96nm minimum global wire pitch and ITRS device performance and interconnect projections [26] for power-performance optimized repeatered global wires, using the methodology in Ho et al. [24]. We find the wire delay to be 100ps/mm. Accounting for a worst-case wire length of four core widths and 4FO4 latching delay, a trip through any cross-core link fits in one cycle at 4GHz operating frequency.

5.3 Base-Core Performance

Figure 5.1 shows the execution-time breakdown based on issue outcome each cycle in *BaseCore*. The issue logic may issue an instruction successfully (*No Stall*), stay idle or process later-to-be-flushed instructions (*Idle/Flushed*), or encounter a stall reason, which we further break into multiple categories: *RAW-Ready Bit* (source value not ready for bypassing), *RAW-Load miss* (non-ready source value is a result of a load miss), and *Other* stalls such as WAW hazard on a load result, and no space in write buffer or MSHR. We check the stall reasons in the listed order, and a cycle is counted for at most one stall reason. *No Stall* gives the IPC for each application. Overall, *BaseCore* is generally utilized efficiently.

Furthermore, we evaluate *BaseCore* with L1 d-cache and MSHR four times

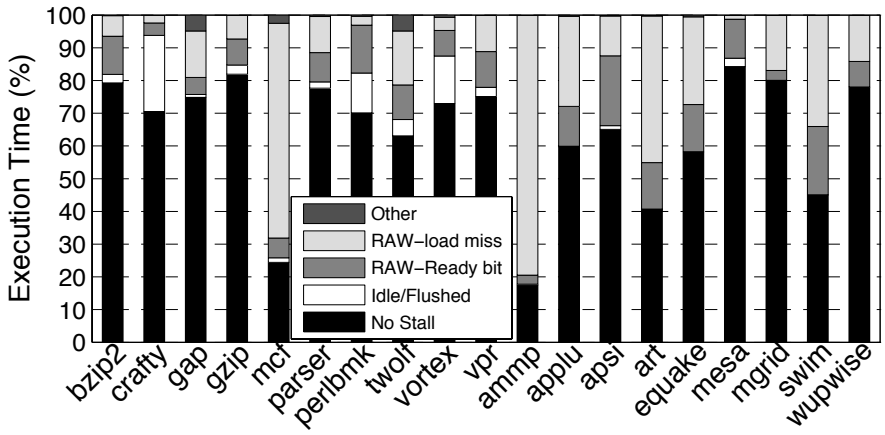

Figure 5.1: *BaseCore* execution-time breakdown based on issue outcome each cycle.

as large, to see what fraction of the benefit comes from constructing such effectively larger structures via core aggregation. The results show that the impact is actually small: 1.6% on average across all applications, with the largest improvement being 3.9% for *mcf*. This indicates that enlarging such structures alone is not the reason behind the speedups of ioCF, which we discuss next.

5.4 In-order Core Fusion Performance

Figure 5.2 shows the performance of *ioCF* with all the performance optimizations described in Section 4.2.9 relative to *BaseCore* for Int (top plot) and FP (bottom plot) applications. It is also compared to other in-order cores (*IO-2i* and *IO-4i*). The *IO-2i* and the *IO-4i* are set up somewhat optimistically, by assuming the same number of pipeline stages as in *BaseCore* and no timing penalties.

ioCF achieves significant speedups, 37.3% (53.8%) for Int (FP) applications on average, with maximum of 85.1% (108%) for *crafty* (*art*), despite the over-

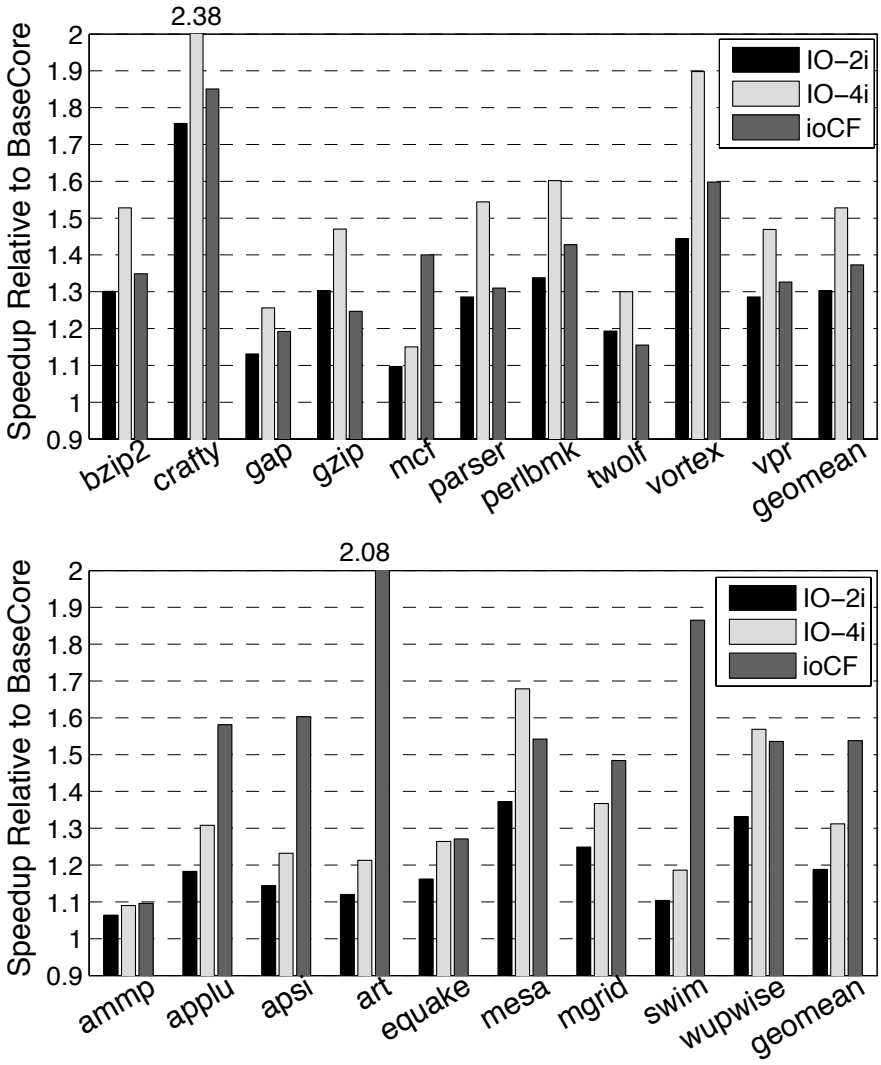

Figure 5.2: Speedup of core fusion (*ioCF*) relative to *BaseCore* for SPECINT (top) and SPECFP (bottom) applications. It is also compared to other in-order cores.

heads of distributed execution. When compared to the other in-order cores, *ioCF* achieves higher performance than *IO-2i* for Int applications, and significantly better than both *IO-2i* and *IO-4i* for FP applications. Besides the computation-intensive applications such as *crafty, vortex, wupwise*, it significantly benefits the memory-intensive applications, such as *mcf, art*, and *swim*.

When core-fusion architecture is put into context of a dynamically recon-

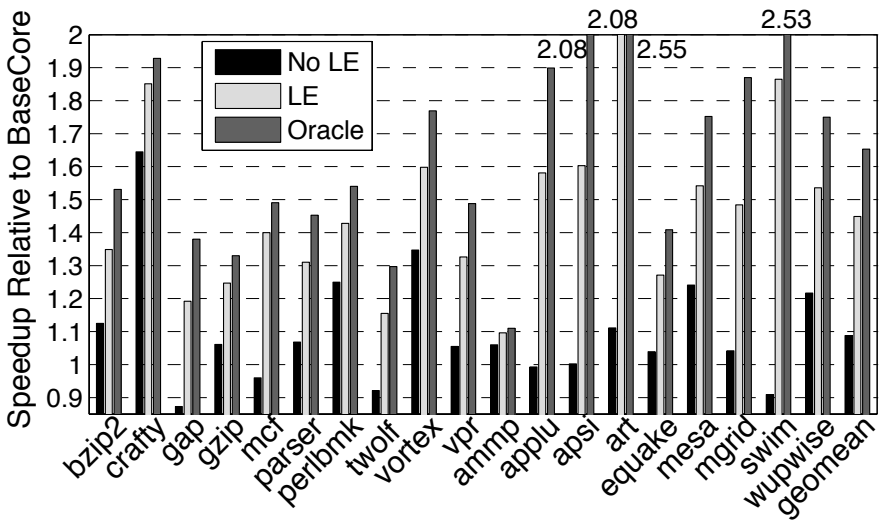

Figure 5.3: Speedup of core fusion with no lookahead execution (No LE), with lookahead execution (LE), and oracle instruction selection, relative to *BaseCore*.

figurable CMP architecture, this improved single-thread performance can be of great benefit, while maximizing thread-count for parallel application regions.

5.5 Performance Analysis

Next, we analyze the effectiveness of the performance optimizations described in Section 4.2.9.

5.5.1 Lookahead Execution

In Figure 5.3, the first two bars compare the performance of *ioCF* without and with LE, respectively. Clearly, LE is critical to performance. Several computation-intensive applications, such as *crafty*, *vortex*, *mesa*, and *wupwise* do

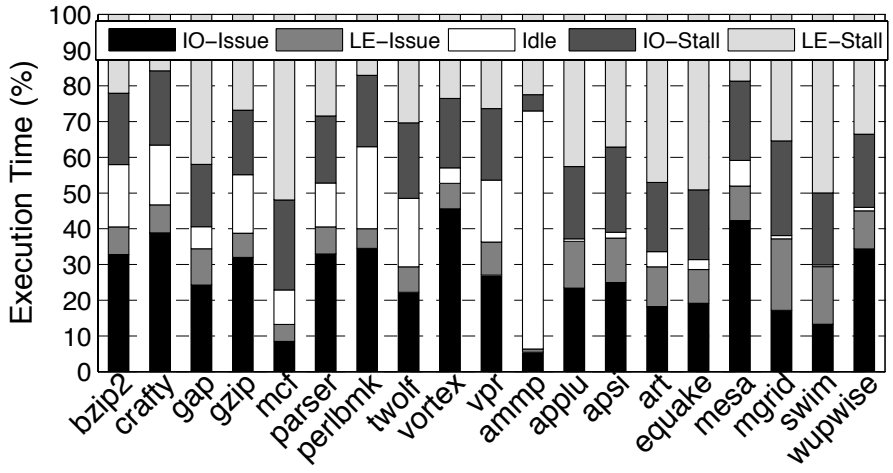

Figure 5.4: Issue-time breakdown in core fusion when running SPEC applications, averaged over four cores.

well even with no LE support. To assess the opportunity for further improvement over LE, we evaluate an optimistic configuration where cores always select instructions from the issue queue using oracle knowledge of the oldest instruction that can issue successfully, subject to the same issue constraints as in LE, and assuming perfect memory-dependence knowledge in the issue queue. The results are shown in Figure 5.3, third bar. *ioCF* falls short of *Oracle* selection by 20.5% on average. The gap is larger in FP applications, 28.2%, while it is 13.7% in Int applications.

Figure 5.4 shows the issue-time breakdown in fused mode averaged over four cores. A core's issue logic is either issuing in order (*IO-Issue*) or in LE mode (*LE-Issue*), or is idle (*Idle*)[1], or is stalling/skipping in in-order (*IO-Stall*) or LE mode (*LE-Stall*). Except in *ammp* application, all four cores demonstrate similar breakdown as the average. In *ammp* we observe a great load imbalance, with the core-0 executing most of the time, and the other cores being idle for 88% of the time. This must be due to the steering policy that insists on chaining depen-

[1]Execution down a branch misprediction path falls into this category.

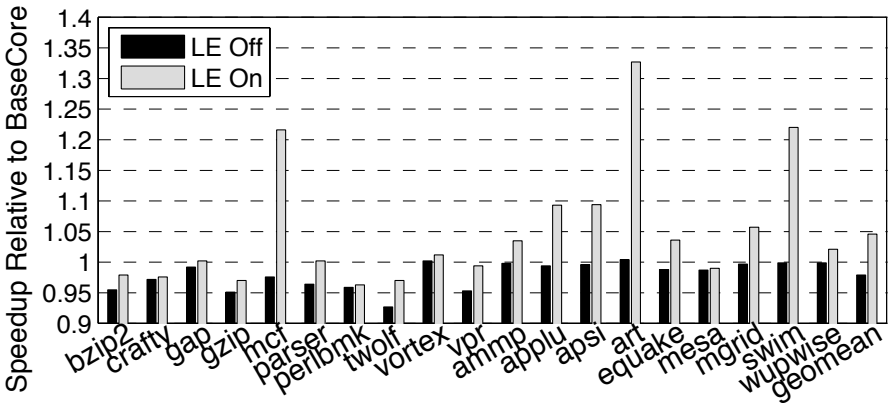

Figure 5.5: Speedup of a base core enhanced with LE relative to *BaseCore* for SPEC applications, with LE disabled (*LE Off*) and LE enabled (*LE On*).

dent instructions in the same core. Cores remain more idle on Int applications relative to the FP applications. This can be a result of larger number of branch mispredictions or of load imbalance due to the steering policy as in *ammp*. Another observation is that Int applications spend less time in lookahead execution than FP applications. In return, FP applications successfully issue larger fraction of instructions in LE mode. Applications that perform well even without LE in Figure 5.3 (*crafty, vortex, mesa*, and *wupwise*) already issue in order mode most of the time.

Lookahead execution in a base core: It would be interesting to see whether LE could be as effective alone in a single base core without core aggregation. We evaluate a base core enhanced with LE and all the necessary mechanisms from our architecture. We provide enough registers to eliminate any lack of free registers. We also increase the issue queue size to 24 as in the case of *ioCF*.

Figure 5.5 provides the performance results of the enhanced base core, relative to the plain base core. We also provide the performance of the enhanced

core with LE disabled to show the overheads that are mainly due to additional rename stage, delaying recovery from branch mispredictions at least until WB stage, and checkpoint rollback penalties. The enhanced core with LE off has 3.5% degradation in Int applications, and with LE execution on, it can barely exceed the base core performance on average. For FP applications, LE achieves 9.3% average improvement. Memory-bound applications (*mcf*, *art*, and *swim*) benefit more than 20%. As seen in Figure 5.1, most of the Int applications already spend little time stalling due to dependences to load misses (RAW-load miss category). As a result, little time is spent in LE mode (on average 11.2% with *mcf* and 6.5% without *mcf*), where successful issue time is even less (on average 2.8% with *mcf* and 2.12% without *mcf*). For FP applications on the other hand, more time is spent in LE (on average 21.8%), and 4.8% of the total time LE issues instructions successfully.

We experimented with LE and an even larger window, and found that speedups fell still well short of *ioCF*. This experiment clearly shows that the proposed LE mechanism makes sense in the core fusion context only.

5.5.2 Copy-out Queue Optimization

In Figure 5.6, we show results for copy-out queue optimization (Section 4.2.9). We evaluate various combinations: selecting the oldest non-issued instruction in the copy-out queue, without and with support for copy-out queue bypass (*1i/no-bypass* and *1i/bypass*), and selecting from the oldest two non-issued instructions, again without and with support for queue bypassing (*2i/no-bypass* and *2i/bypass*).

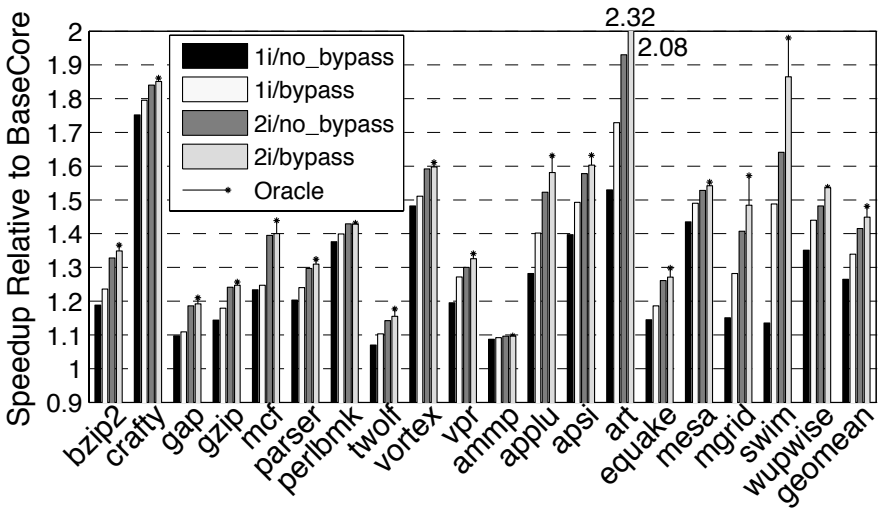

Figure 5.6: Speedup of core-fusion architecture with different policies for issuing copy instructions from the copy-out queue, relative to *BaseCore* performance.

Including all of the copy-out queue optimizations increases the speedup of *ioCF* on FP applications by 25.9% on average, and on Int applications by as much as 11.3%. Involving the second oldest non-issued copy instruction alone provides most of the benefit for Int applications. For FP applications, both optimizations bring significant improvements. We would like to point out that all this additional benefit comes at relatively low-cost.

In the figure, we also show the speedup obtained when an oracle selection policy is used in the copy-out queue (*Oracle*), which selects the oldest ready instruction. Across the board, the gap between our simple selection mechanism and the oracle selection is very small. Only three applications (*art*, *mgrid*, and *swim*) have significant potential for further improvement. The overall result is very encouraging.

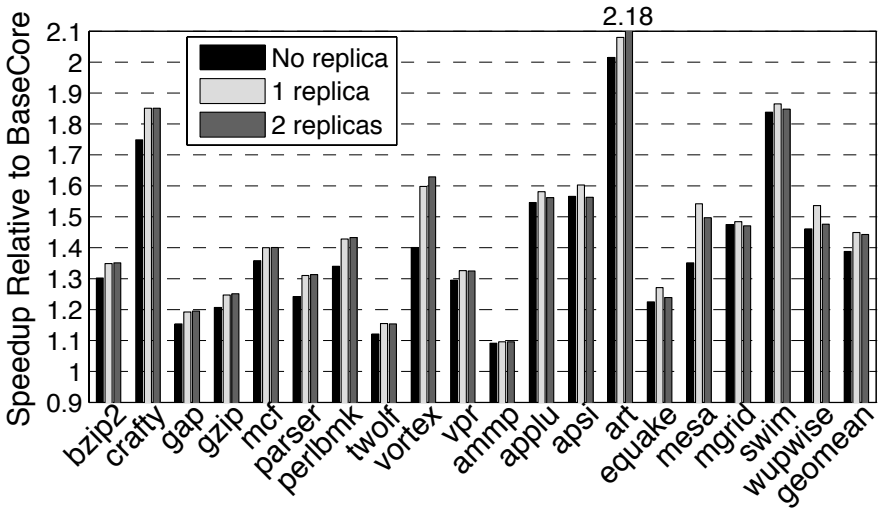

Figure 5.7: Speedup of core-fusion architecture with no, with up to one, and up to two register replications per instruction, relative to *BaseCore* performance.

5.5.3 Register Replication

In Figure 5.7, we show the impact of register-replication optimization (Section 4.2.9) on *ioCF* performance. We provide results for no, up to one, and up to two replications per instruction. Results suggest that up to one replication per instruction is clearly beneficial. Allowing a second replica even slightly degrades performance on FP applications due to the SMU stalls or rename failures in response to increased register pressure. This justifies our decision of supporting up to one replica per instruction in our design.

5.6 Comparison to Out-of-Order Cores

In this section, we compare the performance and area of *ioCF* to single-threaded single- and dual-issue dynamically scheduled out-of-order cores (*OOO-1i* and

Table 5.4: Out-of-order core parameters. In the table, FX, GHR, BHR, BTB, RAS,WrBuff stand for fixed-point, global history register, branch history register, branch target buffer, return address stack, and write buffer, respectively. Cycle counts are in processor cycles.

Out-of-order Cores		
	1-issue	**2-issue**
Frequency	4GHz	4GHz
Fetch/issue/commit	1/1/1	2/2/3
ROB entries	32	64
Register file size	96	128
IQ entries	24	48
LQ/SQ entries	12/24	24/24
Branch min. cycles	7	7
FX/Branch units	1	2
Ld/St units	1	1
FP/Mul/Div units	1	2
SQ forward delay	2	2

OOO-2i, respecitively). We use the SESC simulator's original out-of-order core models. They assume perfect memory disambiguation, that is, ready load instructions do not issue if there is a prior conflicting address-unresolved store in the instruction window, which assumes oracle knowledge.

Table 5.4 lists the core parameters. The branch predictor and memory subsystem of the out-of-order cores are the same as their in-order core counterparts (Table 5.1). To note a few details, the out-of-order cores have extra rename stage and register read stage (after issue). Although in the area estimation we assume a unified register file, which holds the architectural register values, of size given in the table, the simulations assume separate integer and floating-point register files, each having the same amount of free registers as in the unified register file.

Performance evaluation: Figure 5.8 shows the performance of *OOO-1i* and *OOO-2i* relative to *BaseCore* performance. For comparison, we also provide the

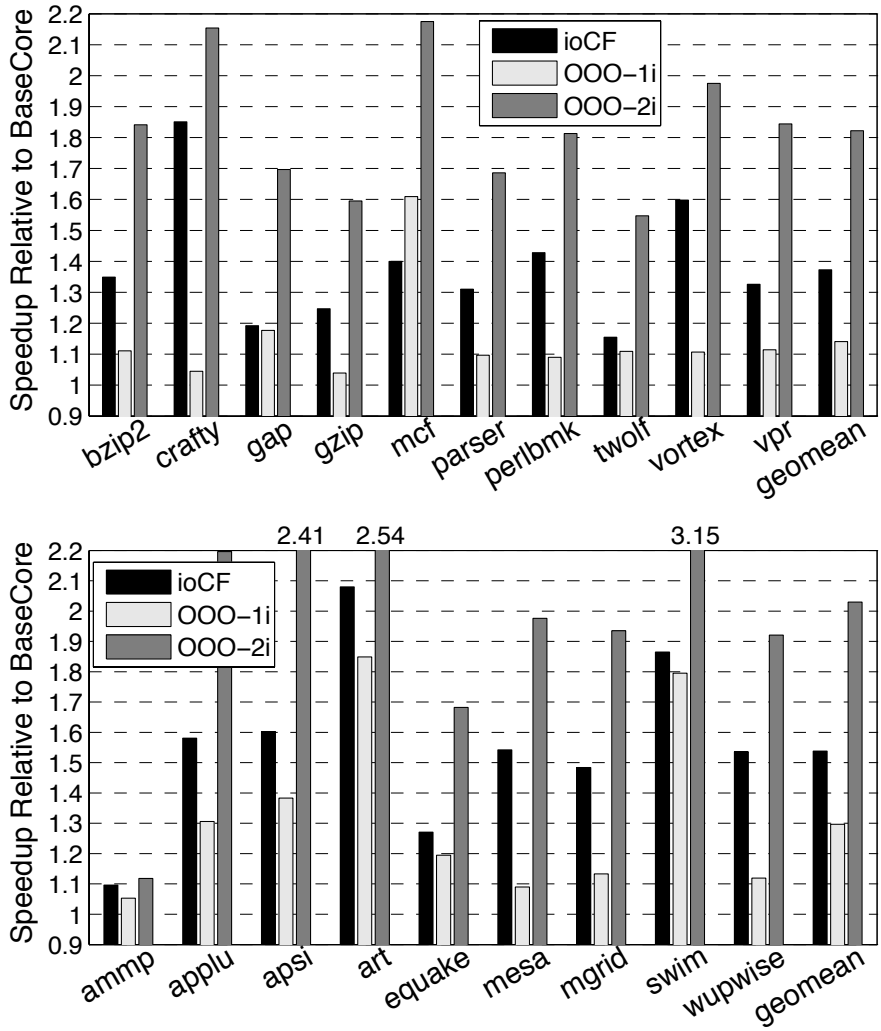

Figure 5.8: Speedup of *ioCF*, *OOO-1i*, and *OOO-2i* configurations relative to *BaseCore* performance for SPECINT (top) and SPECFP (bottom) applications.

performance of *ioCF*. *OOO-1i*, similarly to the results of enhanced *BaseCore* with lookahead execution support (Section 5.5.1), has limited potential, and helps mainly in the memory-bound applications. The speedups larger than ones obtained with lookahead execution may come mainly from the following: out-of-order issue can better hide multi-cycle execution latencies, executed instructions free-up the issue queue entry leaving room for subsequent instructions,

and more efficient memory operation handling. However, *OOO-1i* has significantly inferior average performance than *ioCF* configuration. On the other hand, *OOO-2i* has on average 1.82 and 2 times speedup over *BaseCore* for Int and FP applications, respectively. *ioCF* obtains about half of these speedups.

Area estimation: We estimated the area overhead of realizing out-of-order execution on top of the single-issue *BaseCore* and dual-issue *IO-2i*. We keep the front-end and cache areas the same. We re-estimated areas of larger register files. Using CACTI5.3 [56] again, we added the areas of reorder buffer (ROB), rename table, commit-time rename table, register free-list, and load queue. Finally, to estimate the area for out-of-order issue queue together with wakeup and selection logics, we scaled down (using 0.5 area scaling factor per technology generation) the area of the 20-entry collapsing issue-queue implementation of Alpha 21264 processor [19], and optimistically doubled the result for *OOO-2i*. This part turn out to constitute 21.3% area in both out-of-order cores. As a result, we estimate the *OOO-1i* configuration to be 1.31 times the area of *BaseCore*, and *OOO-2i* configuration area to be 1.39 times the area of *IO-2i* and 2.75 times the area of *BaseCore*.

As a means of verification of our area estimations, we also estimate the areas based on empirical relation that relates performance to area stated long ago by Pollack [47]. According to Pollack's Rule, in the same technology generation, integer performance is roughly proportional to the square root of core's area. Using this relation and average speedups for Int applications obtained relative to the *BaseCore* (Figure 5.8), we find *OOO-1i* and *OOO-2i* to be 1.30 and 3.3 times larger than *BaseCore*, respectively. It is very encouraging to see that the area we calculate for *OOO-1i* closely matches the area estimated from Pollack's Rule.

On the other hand, it seems we have underestimated the area for *OOO-2i*. We observe that our estimation yields a linear increase form *OOO-1i* to *OOO-2i*, while it is known that out-of-order execution structures scale quadratically [45] with issue width. Thus, we believe, the area for *OOO-2i* is between 2.75 and 3.3 of the *BaseCore* area.

OOO-1i has both higher area per core and worse average performance than *ioCF*, so it is clearly not a good design choice. *OOO-2i*, on the other hand, has higher single-thread performance, but its larger area compromises core/thread count on the chip. *ioCF* would be favorable design choice in throughput oriented designs providing good sequential performance as well. The power aspect needs to be taken into account to reach more comprehensive conclusions.

CHAPTER 6

BANKING MEMORY OPERATIONS IN IN-ORDER CORE FUSION

Store-instruction replication in in-order core fusion is a lightweight approach with low hardware overhead for distributed memory-operation disambiguation across cores. However, it significantly increases the number of processed instructions (e.g. 30% increase assuming 10% of instructions are stores), increases the pressure on the write-buffers, and results in more L1 d-cache accesses. The flexibility of steering a load instruction to any of the cores may also reduce the d-cache hit rates.

In this chapter, we study an alternative mechanism for distributed memory-operation handling. Similar to the out-of-order core fusion design in Chapter 4, we bank memory operations into different cores based on lowest bits of cache-line addresses, and use the same modified L1 d-cache indexing and tagging scheme to achieve effectively larger cache, while avoiding flushing the caches on reconfiguration. One downside of banking memory operations, though, is increased load imbalance across cores, due to steering successive operations to the same cache line to the same core.

In the rest of the chapter, we first describe the banking-based memory-operation handling mechanism that we implement in in-order core fusion. Then, we estimate its hardware overhead, and evaluate its performance relative to the store replication mechanism. Finally, we study a dynamic scheme to switch between store replication and banking based on application behavior.

6.1 Mechanism for Memory-Operation Banking

Since effective addresses are not known at the time of steering, we use bank predictor to predict the bank of a memory operation and steer accordingly. Upon address calculation, the bank prediction is verified. Correctly predicted memory operations proceed to the memory system as usual. Memory operations to the same cache line are naturally ordered within the same core. A mispredicted memory operation cannot issue to the memory system. Due to lack of a load queue in cores, the correction mechanism in out-of-order core fusion of sending the operation to the correct core, disambiguating it there, and issuing it to the memory system (Section 2.1.2) is not applicable in this context. Instead, we leverage the checkpointing support in in-order core fusion, and trigger recovery to the last committed checkpoint in the same way as branch mispredictions do. In this respect, the average penalty of a bank misprediction is much higher than in out-of-order core fusion.

To reduce bank mispredictions and the associated penalty, we employ confidence estimation. We steer a memory instruction with high-confidence prediction to the predicted core. Otherwise, we replicate a memory instruction to all cores, which we call *guarding*. With guarding, one of the replicated memory instruction instances is known to be in the correct core, so the other replicated instances do not trigger checkpoint recovery. The one in the correct core issues to the memory system as usual, and the other instances turn into NOP upon address calculation and bank verification. Nullified load instructions mark their destination ARF register as *bogus*.

While guarding a load instruction, the SMU does not know the correct lo-

cation of the load's destination register. It is not informed later either by the cores. Subsequent consumer instructions of that register still need to obtain their source operand from the correct core. We achieve this by adding support for unknown register locations: Each entry in the steering table is extended with an *uncertain* flag to indicate that only one of the cores has the correct register value, but its exact id is unknown. A guarded load's result register is marked as *uncertain* at the time of steering. Later, on an instruction with an *uncertain* source operand, the operand is considered as remote, and a copy instruction is generated and replicated to all cores. All instances of the copy instruction are identical with the same *remote-operand id*. Copy instructions that find their source ARF register as *bogus* turn into NOP. Eventually, only the replicated copy instruction in the core with the true register value will survive and transfer the value to the destination core.

When a non-guarded instruction updates the steering table for its destination register, it clears the *uncertain* flag. Also, when register replication (Section 4.2.9) is applied on an *uncertain* remote operand of an instruction, the new mapping for the source register becomes the destination core of the instruction and the *uncertain* flag is cleared.

Uncertain bits are included in the steering-table checkpoints. On a checkpoint allocation, the SMU includes the logical registers with *uncertain* flag set in all cores' checkpoint register lists sent along *Checkpoint* instructions (Section 4.2.4), so that all possible mappings are checkpointed. In the cores, the *bogus* bit for a register is not cleared until the register is freed. Thus, on a checkpoint recovery, the SMU and LRT mappings for uncertain registers in the checkpoint are restored, and their bogus bits are preserved.

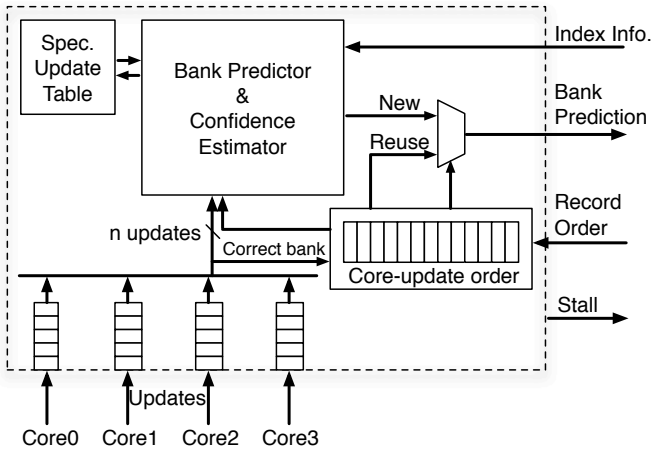

Figure 6.1: Bank predictor and interface to the SMU.

For maintaining forward progress on checkpoint recoveries due to bank mispredictions, we use the guarding mechanism. In the checkpoint interval right after a recovery, the first encountered memory operation is always guarded followed by a new checkpoint allocation.

6.2 Hardware Support for Bank Prediction

Figure 6.1 depicts the main hardware support that we introduce. There is a bank predictor enhanced with confidence estimation, support for speculative update at prediction time in a Speculative Update Table (SUT), in-order update using a FIFO buffer that holds expected core order for updates, and support for reusing verified updates when re-executing instructions after a checkpoint recovery. Predictor update information from a core is processed from an input buffer per core.

6.2.1 Choice of Bank Predictor

We propose a novel bank predictor that has higher prediction accuracy and sharper confidence estimation. It has two levels of tables. The first-level table is indexed with instruction address bits, and maintains the bank and cache-line offset most recently accessed by a memory instruction. This data is hashed with the instruction address to form the index to the second-level table. The second-lavel table gives the next bank and cache-line offset the instruction is expected to access given the current bank and cache-line offset. The data in the second-level table is directly used as the speculative update for the first-level table entry in the SUT.

Owing to the cache-line access locality, especially for regular patterns, the prediction principle of our predictor works quite well. It can correctly capture most stride-based patterns without requiring adder/subtractor as in the stride-two delta predictor. It also does not keep bank histories, and therefore, is more scalable as the number of banks increases than the local bank-history based predictor in terms of storage requirement.

The second-level table has a saturating counter per entry for confidence estimation. On a correct prediction, a counter is incremented by one, and on a misprediction it is decremented by two. We employ 3-bit counters, and only the maximum value of seven corresponds to high-confidence.

Finally, we observe that, in most cases, a significant portion of the predictions exhibit a constant-bank pattern. Trying to predict such sequences with a bank predictor targeted for other non-constant patterns, may lead to wrong predictions due to the limited predictor size, aliasing, or even irregularities in the

effective address sequences that still fall into the same bank. For this reason, we first filter the memory instructions using a constant-bank filter before accessing the actual predictor. It initially assumes constant pattern for each memory instruction. On the first wrong prediction the entry is identified as non-constant. To prevent aliasing from interfering with a constant pattern, we add a few bits (3) as tag to each entry. Only after the third constant-bank occurrence of a memory instruction we assign high-confidence to its subsequent constant-bank predictions.

The constant filter can be embedded into the first-level table of our bank predictor, as it has enough storage and already maintains the last-bank information. A bit per table entry is used to determine whether a constant-bank prediction or normal bank prediction should be performed.

6.2.2 Speculative Update

It is possible to have multiple outstanding instances of a memory operation until the oldest is verified. Late updates reduce the prediction accuracy. To mitigate this, we perform speculative update at the time of prediction for the first level table. To contain the complexity, we do not perform the speculative updates on the predictor table itself, but we maintain them in a separate Speculative Update Table (SUT). It is accessed in parallel with the predictor table. On a match, the speculative update is used to obtain the prediction, otherwise data in the predictor entry is used. Each entry in the SUT keeps count of the unverified predictions to a predictor table entry and the most recent speculative update to that entry. On the first outstanding speculative update, a new entry is allo-

cated, and its counter is initialized. On a verification, the count is decremented, and when it reaches, zero the SUT entry is invalidated. The SUT can be implemented as a (set)associative table indexed with the lower bits of the predictor table index, and tagged with the remaining bits of the index. On a checkpoint recovery, all entries in the SUT are invalidated. Notice that, no action is needed on the actual bank predictor on speculative updates or checkpoint recovery. It is only updated at verification time.

6.2.3 In-order Update

Each new bank prediction for a memory instruction is inserted into the FIFO buffer in order, together with the id of the core the instruction is steered to. Update information for memory operations are sent back from a core to the SMU in oder. Updates from different cores, however, can potentially come out of order. The bank predictor performs the updates in the steering order maintained in the FIFO buffer.

6.2.4 Reusing Verified Bank Predictor Updates

In the context of a checkpointed architecture, even verified updates to the predictor may need to be undone on a checkpoint recovery. We find it both simpler and beneficial to reuse verified updates when re-executing instructions after a flush and checkpoint recovery. For this, the FIFO buffer is also updated with the correct bank on a misprediction, and entries are released form the buffer only when their corresponding checkpoint is released (this requires tracking

checkpoint boundaries in the buffer). On a recovery to the oldest committed checkpoint, any verified predictions starting from the head of the FIFO buffer are traversed and used as predictions. The actual bank predictor is not inquired or updated on prediction reuses. Prediction reuses are considered to have high-confidence.

Steering in the SMU is stalled in case there is no free entry in the in-order update buffer, input buffer, or the SUT.

6.3 Evaluation of Memory-Operation Banking

In this section, we estimate the hardware overhead needed for memory-operation banking, and we evaluate its performance. We assume 4K entry for the first-level table, and 8K entry for the second-level table. We also have 64-entry in-order update FIFO buffer, and 64-entry SUT.

6.3.1 Hardware Overhead Estimation

We first estimate the total hardware overhead of memory banking. In addition to bank predictor structures, we account for a 16-entry input buffer per core in the SMU, as well as a RAM array extending the issue queue in a core to keep the bank predictor update information (lower bits of effective addresses). This information is sent to the SMU in order. We estimate the core fusion total overhead to increase from 60.5% to 84% of a base core area. We conclude that banking adds significant overhead to the fusion mechanism.

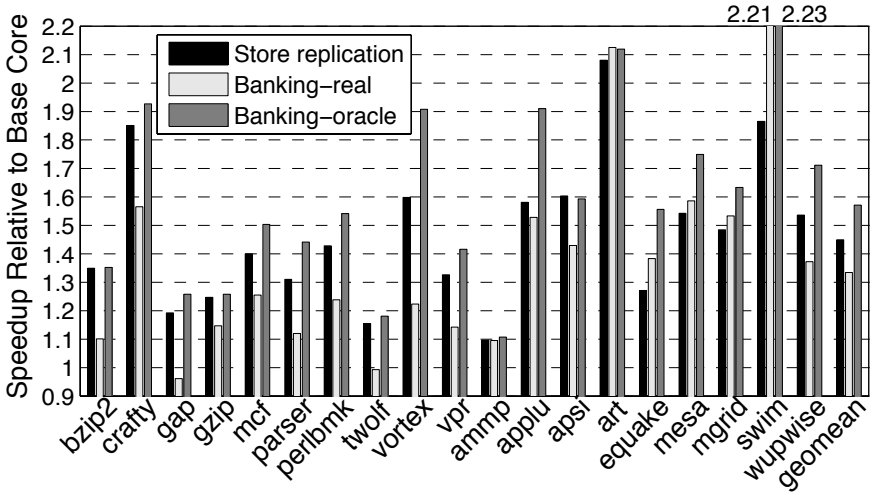

Figure 6.2: Speedup for SPEC applications of in-order core fusion with store replication, memory banking with realistic bank predictor, and memory banking with oracle bank predictor relative to the base in-order core performance.

6.3.2 Performance Evaluation

Figure 6.2 shows the performance results for SPEC applications of core fusion with store replication, memory banking with realistic bank prediction mechanism explained before, and memory banking with steering memory operations always to their correct core. The speedups are all relative to the base in-order core performance.

There is a trade-off between different mechanism. At one hand, with store replication, there is no misprediction and flush due to memory operations, and there is better load balancing. On the other hand, with banking, there is higher cache locality. In both approaches there is memory replication overhead. Though, its extend depends on store instruction percentage in applications or bank prediction accuracy and confidence estimation. It is challenging to achieve the best of both worlds.

Results show that store replication seems to work very well, closely following the ideal banking for many of the applications. *vortex, applu, equake, mesa*, and *swim* applications show very significant gap that may worth improving over the store replication mechanism by introducing extra overhead and complexity. On average, store replication falls short of ideal banking by about 12%.

Realistic banking using bank predictor and confidence estimation performs fairly poor in Int applications. FP applications show more regular and predictable access patterns, and therefore, banking performs very close or better than store replication for these applications. *swim* , for example, benefits significantly from banking.

To better understand the realistic banking results, Figure 6.3 breaks down the predictions made by the block-offset predictor into four categories based on accuracy and confidence. Wrong but high-confidence predictions cause execution rollback. Unconfident predictions also incur penalty due to replicating memory operations to all cores. However, they do not result in rollback.

On top of each bar is the percentage of memory operations that are provided prediction by the bank predictor. The rest either reuse the previous updates on re-execution or are mostly filtered by the constant filter. Constant filter provides effective filtering with almost hundred percent accuracy. It also guards the first three occurrences of memory instructions, the first time they are encountered or after they have been evicted due to aliasing.

As expected, FP applications exhibit higher prediction accuracies and relatively less low-confidence predictions. For Int applications on the other hand, the predictor provides in most cases less than 50%, and for several applications

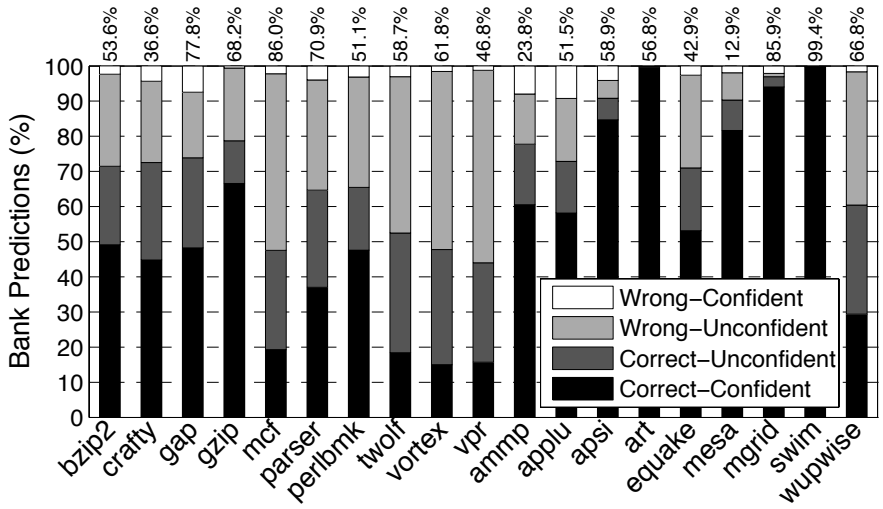

Figure 6.3: Breakdown of bank predictions by the block-offset predictor based on accuracy and confidence. Percentage numbers on top of each bar represent memory operations that use the bank predictor to obtain a prediction. Update reuses and constant-bank predictions constitute the remaining fraction.

less than 20%, correct and high-confidence predictions. Although the predictor successfully keeps the wrong high-confidence predictions low, it results in a lot of guarding on low-confidence predictions. Notice, however, that the results are only for the fraction of memory operations provided on the top of each bar. Most of the remaining predictions can be considered correct and high-confidence.

6.4 Dynamic Policy for Memory-Operation Handling

Results clearly show that realistic banking approach performs worse than store replication for applications that have significant amount of low-confidence predictions. Indeed, for Int applications, low-confidence predictions cause several

times more instructions to be replicated than just store instructions. On top, there are still some bank mispredictions that cause rollback and re-execution. As a result, a dynamic policy that switches between store replication and banking based on application behavior seems an attractive approach that can bring the benefits of both worlds.

Initially, we start with memory banking off. Periodically, just before a checkpoint allocation at the end of a period, we evaluate which scheme is expected to be more beneficial to performance. If the number of stores in the last period is less than the number of unconfident predictions (that would cause guarding) then store replication is suggested policy for the next period. We count an unconfident load instruction as two, and an unconfident store instructions as one, since load instructions have destination registers and bring more overhead when being guarded. Otherwise, banking is suggested. We switch to the suggested policy if it was suggested for at least two times in a row. The bank predictor needs to be updated at all times for this dynamic scheme. (Note that, only one of the replicas of a memory instruction should attempt to update the predictor.)

Differently than stores in store-replication intervals, stores in banking intervals do not update the store counter in the L2 cache, they always perform to the L2 cache, and invalidate any cache-line copies in the other L1 d-caches, as in the baseline coherence scheme. Also, there is no need to delay releasing stores in a checkpoint interval with banking on from the write-buffer to the memory system upon commit of the checkpoint.

To preserve correct memory dependences on transitioning from banking to replication and vice versa we do the following: We constrain the transitioning

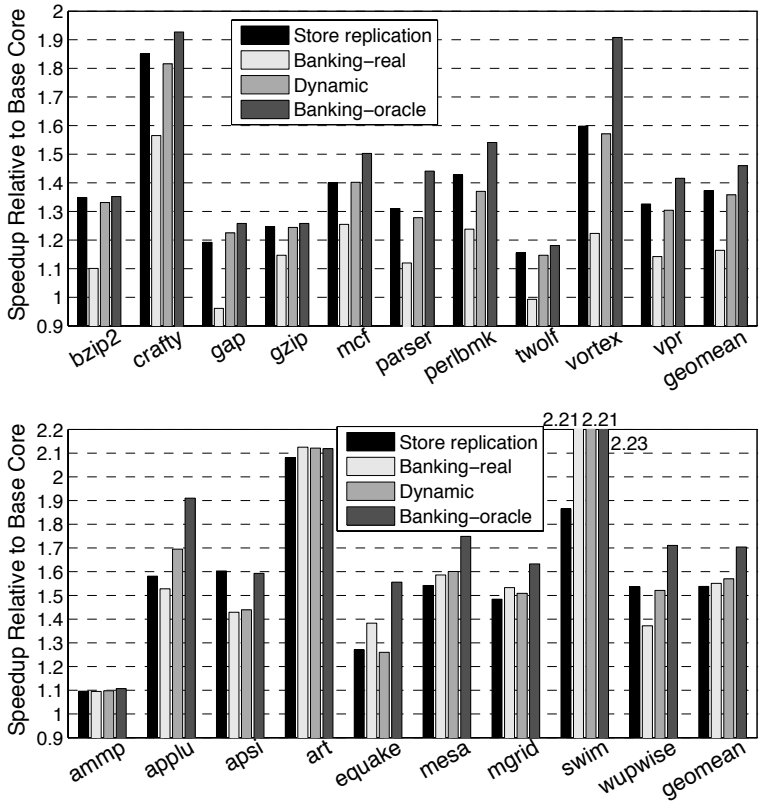

Figure 6.4: Speedup for SPEC Int (top) and FP (bottom) applications, relative to base in-order core, of in-order core fusion with dynamic policy to select between store replication and banking mechanisms. We also provide results for each mechanism alone.

from replication to banking to happen only if there is at least one (replicated) store instruction in the last checkpoint interval. This ensures that subsequent non-replicated stores will not race with any outstanding prior load misses to the same cache-line from other cores in reaching the L2 cache. The replicated stores in the last checkpoint interval already will not be released to the memory system until all primary load misses from all cores from that and prior checkpoint intervals are seen by the L2 cache (Section 4.2.7).

On transitioning from banking to replication, on the other hand, the SMU generates a fence operation replicating it to all cores, just before allocating the

checkpoint. The fence operation ensures that all prior non-replicated stores are made visible to all cores before any subsequent possibly conflicting load or store operation issues to the memory system from any of the cores.

Figure 6.4 shows performance results of in-order core fusion with dynamic policy to select between store replication and banking, relative to base core for SPEC Int and SPEC FP applications. We again provide the results with store replication and banking alone, for comparison.

We see that, for Int applications we were able to obtain results very close to those with store replication, since for these applications clearly it is the favorable policy. Interestingly, *gap* achieves higher performance with dynamic policy than both static polices. For several FP applications, the dynamic policy exceeds the performance of both static policies. This shows that these applications have distinct phases where each of the policy is better, and our dynamic selection policy is good at identifying the correct approach. Only for *apsi* and *equake*, the dynamic policy has performance of the lower performing approach. For example, in Figure 6.3, we see that *apsi* has high prediction accuracy, however, it has relatively large number of wrong high-confidence predictions as well, which may be the reason for the performance degradation. The dynamic scheme improves the average performance on FP applications relative to base core from 53.8% to 57%, while mostly preserving the Int performance.

CHAPTER 7

RELATED WORK*

7.1 Reconfigurable Architectures

Smart memories [36] 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 cores while remaining transparent to the ISA, and it does not require specialized compiler support.

TRIPS [52] 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. Composable Lightweight Processors [27], which is based on out-of-order cores implementing the EDGE ISA [8], can put together a large number of execution units, thanks in part to EDGE's compiler and hardware support for block-atomic execution and explicit encoding of instruction dependences.

In Voltron [58], four VLIW cores can selectively execute in lockstep to behave like a wide-issue multicluster VLIW processor. The architecture depends on

*© ACM, 2007. Partly reprinted, with permission, from "E. Ipek, M. Kırman, N. Kırman, J. F. Martínez. Core Fusion: Accommodating software diversity in chip multiprocessors. In *Intl. Symp. on Computer Architecture (ISCA)*, pages 186-197, San Diego, CA, Jun. 2007. http://doi.acm.org/10.1145/1250662.1250686"

compiler support to partition single-thread work among cores, schedule execution on each core, and orchestrate communication and synchronization between cores.

Tarjan et al. propose Federation [55], which strives to partially leverage two single-issue in-order cores to construct a two-issue out-of-order CPU. At a high level, their approach is to incorporate centralized, area-conscious structures for out of order processing, which bypass those in the in-order cores, essentially using the two in-order cores as clustered execution units. It is unclear how sensitive Federation is to resource provisioning in the base cores (e.g., the paper assumes sufficient storage to support four hardware threads), or to close integration of the base cores (e.g., one cycle to bypass an operand over to the other core's register file, or a "cache controller" of unspecified latency to forward requests to, and collect replies from, both cores' L1 caches).

7.2 Clustered Architectures

Out-of-order core fusion borrows from some of the mechanisms developed in the context of clustered architectures [4, 5, 7, 12, 13, 14, 18, 45, 59]. Our proposal is closest to the recent thrust in clustered multithreaded processors (CMT) [15, 16, 34]. 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 7.1 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

Table 7.1: Comparison to recent proposals for clustered processors. FE and BE stand for front- and back-end, respectively. NS stands for not supported.

Architecture	*Performance Potential*		*Throughput Potential*	*Modularity*			*Reconfigurability*		
	Sequential	Parallel		FE	BE	Caches	FE	BE	Caches
[15]	Low	High	High	Yes[1]	Yes	No	No	Yes	No
[16] (Shared Banks)	High	Low	Low	Partial[1]	Yes	Yes	No	Yes	No
[16] (Private Banks)	Low	NS	High	Partial[1]	Yes	Yes	No	Yes	No
[34] (Fewer, large FEs)	High	Low	Low	Partial[1]	Yes	Yes	No	Yes	Yes
[34] (More, small FEs)	Low	High	High	Yes[1]	Yes	Yes	No	Yes	Yes
[46]	High	NS	NS	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

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. Hence, the lack of reconfigurability in the memory system and the front-end, coupled with the lack of coherence and consistency support makes this architecture inadequate for supporting workload diversity.

Latorre et al. [34] 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. Inevitably, these inefficiencies limit the total number of threads that can be supported on the same die, thereby prohibiting the exploitation of high levels of TLP and making this architecture inadequate for supporting workload diversity.

Collins et al. [15] explore four alternatives for clustering SMT processors. Among them, the most relevant to 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. Hence, this architecture is also inadequate for supporting workload diversity.

Parcerisa [46] 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 or multiprogrammed workloads are not discussed and reconfiguration is not considered. The branch predictor is interleaved on high-order bits, which may result in underutilized space. Mechanisms for keeping consistent global history across different branch predictor banks are not discussed.

Chaparro et al. [14] distribute the rename map and the ROB to obtain tem-

perature 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 out-of-order core fusion, we also fully distribute our ROB, but without requiring expensive pointer chasing mechanisms across cores. In their architecture, although the rename tables are distributed, renaming destination registers of instructions and maintaining free-register lists are still centralized. In our in-order core fusion even these functionalities and structures are distributed.

Prior works on distributed/clustered architectures that try to enable out-of-order execution using multiple in-order processing elements [28, 42, 57] have several common features. First, they employ two-level hierarchical register files both in the ISA and in the microarchitecture: a local register file [42, 57] or accumulator [28] per processing element and a centralized global register file. This hierarchy is exposed to the ISA (although each work assumes different ISA). Second, compiler forms dependence chains that execute contiguously and in order on a processing element. Instructions in a chain are dependent on each other through local register/accumulator, while inter-chain dependences are satisfied over global register file. The compiler accordingly extracts the chains and performs register renaming. Chains in [42] are typically much longer than ones in [28, 57], which are few instructions long.

Salverda and Zilles [51] investigate the fundamental performance challenges of single-threaded execution on clustered in-order processors, using a model where many clustering overheads are idealized. They conclude that, despite the absence of important clustering overheads, a design with a realistic imple-

mentation of instruction steering would fall well short of the performance of a typical out-of-order processor. However, they do show that potential for higher performance exists in theory, and as a result they encourage research in alternative mechanisms for core aggregation, or in creative ways to provide the base in-order cores with some out-of-order execution capability.

7.3 Scalable Issue-Queue Designs

Reducing the complexity of issue queues in dynamically scheduling processors has drawn significant attention. Several works apply prescheduling among in-flight non-issued instructions, which reorders instructions based on dependences or estimated/predicted latencies, and confines selection to relatively small number of instructions that are ready or close-to-ready. Palacharla et al. [45] propose employing multiple FIFO queues each feeding exclusive functional units, and only instructions at FIFO heads are considered for issue. The prescheduling (steering) logic tries to chain dependent instructions consecutively in a queue, and if not possible waits for an empty queue. This policy to steer instruction to multiple FIFO queues remains the state-of-the-art. [1] modifies this steering policy for FP applications to take into account their estimated issue times when inserting into the issue queues. In another approach, they modify the selection from a queue to consider expected issue time and instruction age information. Other proposals [10, 11, 38, 48] employ at least one smaller conventional issue queue in their prescheduling designs. In the first three, prescheduling is done through a RAM-based array with in-order entry release, while in the last approach, smaller conventional issue queues are used.

To reduce wake-up complexity in issue queues, [10, 11] propose a RAM queue indexed with register identifier to wake up the first consumer [10] or at most N consumers [11] of a register. Huang et al. [25] propose a hybrid approach for wakeup logic by handling the common case of one in-flight consumer by enabling only its comparator in the issue queue, while in case of multiple consumers all or a part of comparators in the issue queue are enabled. This requires consumer instructions to subscribe into the producer's issue queue entry.

The overall distributed issue logic employed in our in-order-core-based work is based on the design in [45]. On top of that, lookahead execution is a prescheduling technique that tries to select only one instruction from a queue to be considered for issue. However, it is simpler than prior prescheduling proposals as it is based on a single issue queue design, does not need to compute or predict issue times, and does not assume anything for instruction dependences in a queue.

7.4 Other Related Work

Trace Processors [49] 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. Multi-Scalar processors [53] 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.

CHAPTER 8

CONCLUSIONS

Inferior sequential-code performance in future's highly-parallel CMPs is an important problem when executing future's dynamic and diverse set of applications, ranging from purely sequential to highly parallel, and everything in between. To address this problem, in this dissertation, we have proposed a novel reconfigurable CMP architecture that we call *core fusion*. It allows relatively simple CMP cores to dynamically fuse into larger, more powerful processor in order to speedup sequential-code. The result is a flexible CMP architecture that can adapt to a diverse collection of software. It does so without requiring higher software complexity, a customized ISA, or specialized compiler support.

We propose and evaluate solutions for both out-of-order and in-order base cores. For each case, we respect the inherent capabilities of the base cores and their independent nature. While out-of-order base cores provide the design with valuable opportunities for latency hiding, minimally-provisioned in-order base cores leave little margin for inefficiencies. We have demonstrated that core fusion concept can be carried out effectively in the domain of simple in-order cores, despite the fact that the base in-order cores have very limited latency-hiding ability. Overall, very significant differences exist between both solutions.

The modular and distributed mechanisms require little central processing and functional replication. In the course of formulating our in-order core fusion solution, we devise: (1) A distributed checkpoint-based mechanism for bookkeeping of the program state; (2) a distributed register renaming mechanism; (3) a lightweight approach for distributed memory disambiguation, and (4) a lookahead execution within each base core to partially hide cross-core commu-

nication an other latencies. Our evaluation shows that a four-way fused configuration delivers a 45% performance gain over a single in-order core, at the cost of 15% hardware overhead per core.

BIBLIOGRAPHY

[1] J. Abella and A. González. Low-complexity distributed issue queue. In *International Symposium on High-Performance Computer Architecture*, pages 73–82, Madrid, Spain, February 2004.

[2] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *International Symposium on Microarchitecture*, pages 423–434, San Diego, CA, December 2003.

[3] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The impact of performance asymmetry in emerging multicore architectures. In *International Symposium 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 *International Symposium 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 *International Symposium 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 *International Symposium 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 *International Symposium on Computer Architecture*, pages 264–274, San Diego, CA, June 2003.

[8] D. Burger, S. Keckler, K. McKinley, M. Dahlin, L. John, C. Lin, C. Moore, J. Burrill, R. McDonald, and W. Yoder. Scaling to the end of silicon with EDGE architectures. *IEEE Computer*, 37(7):44–55, July 2004.

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

[10] R. Canal and A. González. A low-complexity issue logic. In *International Conference on Supercomputing*, pages 327–335, Santa Fe, NM, May 2000.

[11] R. Canal and A. González. Reducing the complexity of the issue logic. In *International Conference on Supercomputing*, pages 312–320, Sorrento, Italy, June 2001.

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

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

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

[15] J. D. Collins and D. M. Tullsen. Clustered multithreaded architectures - pursuing both IPC and cycle time. In *International Parallel and Distributed Processing Symposium*, 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 *International Symposium 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 *International 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 *International Symposium on Microarchitecture*, pages 149–159, Research Triangle Park, NC, December 1997.

[19] J. A. Farrell and T. C. Fischer. Issue logic for a 600-MHz out-of-order execution microprocessor. *IEEE Journal of Solid-State Circuits*, 33(5):707–712, May 1998.

[20] 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.

[21] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier Science Pte Ltd., third edition, 2003.

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

[23] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *IEEE Computer*, 41(7):33–38, July 2008.

[24] R. Ho, W. Mai, and M. A. Horowitz. The future of wires. *Proceedings of the IEEE*, 89(4):490–504, April 2001.

[25] M. Huang, J. Renau, and J. Torrellas. Energy-efficient hybrid wakeup logic. In *International Symposium on Low Power Electronics and Design*, pages 196–201, Monterey, CA, August 2002.

[26] The ITRS Technology Working Groups, http://www.itrs.net. *International Technology Roadmap for Semiconductors (ITRS) 2007 Edition*.

[27] C. Kim, S. Sethumadhavan, D. Gulati, D. Burger, M. S. Govindan, N. Ranganathan, and S. W. Keckler. Composable lightweight processors. In *International Symposium on Microarchitecture*, pages 381–394, Chicago, IL, December 2007.

[28] H.-S. Kim and J. Smith. An instruction set and microarchitecture for instruction level distributed processing. In *International Symposium on Computer Architecture*, pages 71–81, Anchorage, AK, May 2002.

[29] N. Kırman, M. Kırman, M. Chaudhuri, and J. F. Martínez. Checkpointed early load retirement. In *International Symposium on High-Performance Computer Architecture*, San Francisco, CA, February 2005.

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

[31] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for pro-

cessor power reduction. In *International Symposium on Microarchitecture*, pages 81–92, San Diego, CA, December 2003.

[32] 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 *International Symposium on Computer Architecture*, pages 64–75, München, Germany, June 2004.

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

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

[35] N. Mäding, J. Leenstra, J. Pille, R. Sautter, S. Büttner, S. Ehrenreich, and W. Haller. The vector fixed point unit of the synergistic processor element of the Cell architecture processor. In *Design, Automation and Test in Europe*, pages 244–248, Munich, Germany, March 2006.

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

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

[38] P. Michaud and A. Seznec. Data-flow prescheduling for large instruction windows in out-of-order processors. In *International Symposium on High-Performance Computer Architecture*, pages 27–36, Nuevo Leone, Mexico, January 2001.

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

[40] S. M. Mueller, C. Jacobi, H-J. Oh, K. D. Tran, S. R. Cottier, B. W. Michael, H. Nishikawa, Y. Totsuka, T. Namatame, N. Yano, T. Machida, and S. H.

Dhong. The vector floating-point unit in a synergistic processor element of a CELL processor. In *IEEE Symposium on Computer Arithmetic*, pages 59–67, Cape Cod, MA, June 2005.

[41] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *International Symposium on High-Performance Computer Architecture*, pages 129–140, Anaheim, CA, February 2003.

[42] S. Narayanasamy, H. Wang, P. Wang, J. Shen, and B. Calder. A dependency chain clustered microarchitecture. In *International Parallel and Distributed Processing Symposium*, Denver, CO, April 2005.

[43] University of Illinois at Urbana-Champaign. `http://sesc.sourceforge.net`, 2005.

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

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

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

[47] F. Pollack. New microarchitecture challenges in the coming generations of CMOS. In *International Symposium on Microarchitecture*, Haifa, Israel, November 1999. (Keynote presentation).

[48] S. E. Raasch, N. L. Binkert, and S. K. Reinhardt. A scalable instruction queue design using dependence chains. In *International Symposium on Computer Architecture*, pages 318–329, Anchorage, AK, May 2002.

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

[50] E. Safi, A. Moshovos, and A. Veneris. A physical level study and optimization of CAM-based checkpointed register alias table. In *International*

Symposium on Low Power Electronics and Design, pages 233–236, Bangalore, India, August 2008.

[51] P. Salverda and C. Zilles. Fundamental performance constraints in horizontal fusion of in-order cores. In *International Symposium on High-Performance Computer Architecture*, pages 252–263, Salt Lake City, UT, February 2008.

[52] 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 *International Symposium on Computer Architecture*, pages 422–433, San Diego, CA, June 2003.

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

[54] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual flow pipelines. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 107–119, Boston, MA, October 2004.

[55] D. Tarjan, M. Boyer, and K. Skadron. Federation: Out-of-order execution using simple in-order cores. Technical Report CS-2007-11, University of Virginia, August 2007.

[56] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. CACTI 5.3, HP Laboratories Palo Alto, `http://quid.hpl.hp.com:9081/cacti/`, 2009.

[57] F. Tseng and Y. N. Patt. Achieving out-of-order performance with almost in-order complexity. In *International Symposium on Computer Architecture*, pages 3–12, Beijing, China, June 2008.

[58] H. Zhong, S. A. Lieberman, and S. A. Mahlke. Extending multicore architectures to exploit hybrid parallelism in single-thread applications. In *International Symposium on High-Performance Computer Architecture*, pages 25–36, Phoenix, AZ, February 2007.

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