Overcoming Single-Thread Performance Hurdles in the Core Fusion Reconfigurable Multicore Architecture

Janani Mukundan[†]
mukundan@csl.cornell.edu

Saugata Ghose[†]
ghose@csl.cornell.edu

Robert Karmazin[†]
rob@csl.cornell.edu

Engin İpek[*]
ipek@cs.rochester.edu

José F. Martínez[†]
martinez@cornell.edu

[†]Computer Systems Laboratory
Cornell University
Ithaca, NY 14853 USA
http://m3.csl.cornell.edu/

[*]University of Rochester
Rochester, NY 14627 USA
http://cs.rochester.edu/~ipek/

ABSTRACT

Though the prime target of multicore architectures is parallel and multithreaded workloads (which favors maximum core *count*), executing sequential code fast continues to remain critical (which benefits from maximum core *size*). This poses a difficult design tradeoff. *Core Fusion* is a recently-proposed reconfigurable multicore architecture that attempts to circumvent this compromise by "fusing" groups of fundamentally independent cores into larger, more aggressive processors dynamically as needed. In this way, it accommodates highly parallel, partially parallel, multiprogrammed, and sequential codes with ease. However, the sequential performance of the original fused configuration falls quite short of an area-equivalent, monolithic, out-of-order processor.

This paper effectively eliminates the fusion deficit for sequential codes by attacking two major sources of inefficiency: collective commit and instruction steering. We demonstrate in detail that these modifications allow Core Fusion to essentially match the performance of an area-equivalent monolithic out-of-order processor. The implication is that the inclusion of wide-issue cores in future multicore designs may be unnecessary.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architectural Styles—*adaptable architectures*; B.7.1 [**Integrated Circuits**]: Types and Design Styles—*microprocessors and microcomputers*

Keywords

microarchitecture, multicore, software diversity, Core Fusion, collective commit, instruction steering, genetic programming

1. INTRODUCTION

Core Fusion [16] is a recently-proposed reconfigurable multicore architecture where four fundamentally independent two-issue out-of-order cores can fuse on demand into larger "supercores." Core Fusion supports software diversity because it may be configured for fine-grain parallelism (by providing more lean cores), coarse-grain parallelism and multiprogramming (by providing fewer, but more powerful 4-issue supercores, up to capacity), and sequential execution (by executing on one 8-issue supercore). Core Fusion scales with the number of cores on chip not by providing ever-wider-issue supercores, but by providing the ability to instantiate more supercores simultaneously.

The original paper compares Core Fusion against a multitude of competing symmetric and asymmetric multicore configurations of different granularities for sequential, multiprogrammed, and parallel codes. In all cases, Core Fusion is either the fastest or second fastest configuration; and all other configurations trail significantly from Core Fusion in performance for at least one type of workload. Moreover, for codes with significant parallel and serial sections, Core Fusion's dynamic reconfiguration capability compares favorably against any other static design.

However, the original paper also shows that, when executing sequential code, Core Fusion's performance is still noticeably lower than that of an area-equivalent monolithic out-of-order processor. This paper effectively eliminates Core Fusion's sequential performance deficit by attacking two major sources of its inefficiency—distributed commit (through efficient ROB storage) and remote operand communication (through improved instruction steering). We generate transistor-level netlists and simulate the circuit using HSPICE, verifying that our proposed steering mechanism operates within a single cycle with margins in a 4 GHz implementation at 22 nm.

We show that the performance of our improved Core Fusion architecture is within 98% of that of an area-equivalent monolithic out-of-order processor for both SPEC2000 Int and FP applications (vs. 88% and 82%, respectively, for the original Core Fusion proposal). The implication of this result is that the inclusion of wide-issue cores in future multicore designs may effectively be unnecessary, since a Core Fusion organization can now match their sequential performance when needed while retaining the ability to offer fine-grain parallelism on demand.

2. BACKGROUND: CORE FUSION

Core Fusion [16] provides the capability to dynamically fuse four two-issue out-of-order cores to form a large "supercore" with up to four times the fetch, execute, and commit width, as well as four times the aggregate cache and branch prediction capacity (Figure 1). In this section, we provide an abridged, non-comprehensive

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICS'12, June 25–29, 2012, San Servolo Island, Venice, Italy.
Copyright 2012 ACM 978-1-4503-1316-2/12/06 ...$10.00.

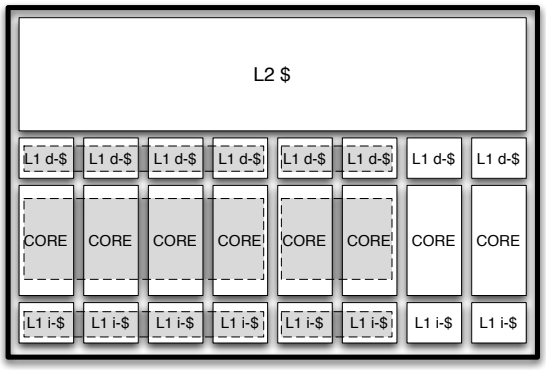

Figure 1: Conceptual depiction of an eight-core Core Fusion chip (not an actual floorplan). At left, four independent cores are fused to form a larger "supercore." To the right of them, two independent cores are fused into a smaller "supercore." The two rightmost cores are shown operating independently as distinct processing elements.

summary of the behavior of Core Fusion in fused mode as proposed originally; more details can be found in that paper.

Collective Fetch — When a four-core group transitions into fused mode, their instruction caches are flushed and they begin fetching code in a distributed fashion: On an instruction cache miss, a cache block of eight words is distributed across the four cores (a two-word sub-block for each core), and the tag for the cache block is replicated across cores. A centralized *fetch management unit* (FMU) coordinates collective fetch by receiving and communicating fetch information across cores. The collective fetch mechanism allows each core to fetch two instructions every cycle. Instruction fetch is aligned: Core 0 is responsible for the oldest two instructions, Core 1 for the next two, and so on. On a branch or jump, the oldest instruction in the fetch group might not be mapped to Core 0. In such cases, to preserve fetch alignment, the appropriate lower order cores skip fetch for that cycle. On a branch, the appropriate core accesses its own branch predictor, and BTB prediction table events are communicated to all other cores along with the target program counter. When events occur that stall the front end or require a pipeline flush (e.g. branch mispredictions, replay traps, fetch stalls), the cores send the program counter of the instruction causing the event to the FMU, which coordinates either the flush or stall by sending appropriate messages to each core.

Steer & Rename — After fetch, each core pre-decodes two instructions and sends the source and destination register specifiers to a common *steering management unit* (SMU). The SMU stage in the pipeline is similar to the rename stage of an out-of-order superscalar processor. It includes the rename map table and free lists for register allocation. Additionally, the SMU is also responsible for distributing instructions across cores. For this, the SMU uses a *global steering table*, which keeps track of the mapping between architectural registers and cores. The steering logic responsible for assigning instructions to cores also resides with the SMU. No more than two instructions are dispatched to each core every cycle. The SMU is also responsible for generating copy instructions to transfer operands to consumers that are not co-located with their producers. The copy instruction injection bandwidth is restricted to two per core, per cycle. The original paper identifies the overhead associated with remote operand communication between producer-consumer pairs steered to different cores as one important source of performance degradation (which we improve in this paper).

Collective Execution — Once the instructions have been steered and renamed, they are sent to their respective back ends based on the steering decisions. The cores make use of an operand crossbar to support remote operand communication using the SMU-generated copy instructions. Additionally, the back end also includes copy-in and copy-out queues for incoming and outgoing copy instructions, respectively.

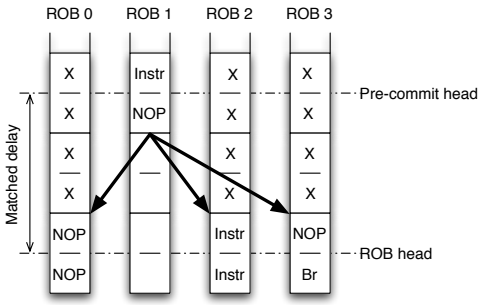

Figure 2: In this distributed ROB, a fetch group with four inserted NOPs is preparing to commit. The pair in ROB 1 is not ready to commit, so it broadcasts a a *Stall* signal to the other ROBs. Pre-commit and ROB heads are spaced to match the communication delay; the signal arrives just in time to prevent the other cores' pairs from committing. When the pair in ROB 1 becomes ready, a *Resume* signal is broadcast as the pair advances to the ROB head. The delay matching allows all of the ROBs to retire the entire fetch group in sync.

Collective Memory Access — Memory accesses are handled by the cores' data caches and load-store queues in a distributed fashion, using a bank-by-address approach. This allows the architecture to keep data coherent without requiring data cache flushes after dynamic reconfiguration, and to elegantly support store forwarding and speculative loads. The core issuing each load/store is determined using the effective address, and memory disambiguation is handled mostly locally at that core. Because effective addresses are generally not known when instructions are steered, a bank prediction mechanism is employed.

Collective Commit — The four cores can synchronously commit a fetch group of eight instructions (two per core) in one cycle. When instruction commit is blocked in one of the cores, commit of the fetch group in all cores must be stalled to guarantee (later) synchronous commit. This is done via *Stall*/*Resume* signals. Each ROB is equipped with a *pre-commit head pointer* placed ahead of the actual ROB head. Instruction pairs that are not ready to commit at the time they reach the pre-commit head will stall in place and send *Stall* signals to the other cores. Once they are ready, they send *Resume* signals to the other cores and continue moving toward the ROB head. The number of ROB entries between the pre-commit head and the actual ROB head is enough to cover the number of cycles it takes for the *Stall*/*Resume* signal to reach the other cores (2 cycles, requiring 4 entries).

In the next few sections we discuss two sources of inefficiencies in the Core Fusion pipeline and provide solutions to overcome them.

3. DISTRIBUTED COMMIT

Recall that, to simplify collective commit, Core Fusion commits one fetch group at a time (eight instructions, two per core). However, fetch groups are often not filled up to capacity, due to branch mispredictions and PC redirections after predict-taken branches. When this happens, Core Fusion inserts NOPs into the ROBs at the end of the fetch stage, so that each fetch group still occupies exactly two slots in each core's ROB.

Figure 2 shows an example of a fetch group padded with NOPs. The second instruction in ROB 1's pair is the target of a branch (not shown); thus, the ROB slots that precede this instruction are filled with NOPs to preserve alignment. Also, the first instruction in ROB 3's pair is a taken branch, and thus the subsequent instructions in the fetch group (only one in this case) have been nullified.

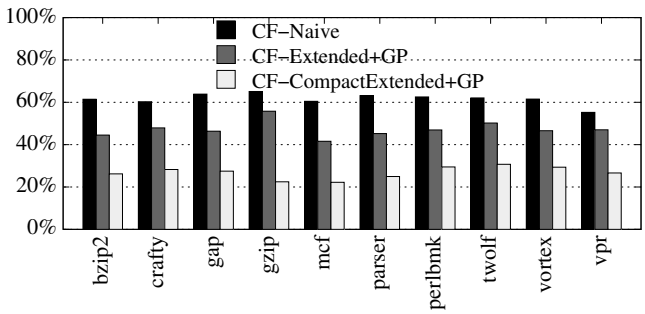

Figure 3: Percentage of ROB entries occupied by NOPs when instruction fetch stalls due to a full ROB while running SPEC-Int applications. GP stands for the steering algorithm derived from genetic programming (see Section 4).

The main downside of this approach is that the effective size of a core's ROB may be significantly reduced because of NOP insertions. Figure 3 shows that during fetch stalls caused by a full ROB, on average, 60% of the ROB entries are taken up by NOPs for SPEC-Int applications (CF-Naive shows the original Core Fusion mechanism). The original Core Fusion proposal already identifies NOP insertions as one important source of performance degradation [16]. In this section, we propose three new mechanisms to achieve synchronous commit of fetch groups while reducing the storage overhead of fetch group alignment in the ROBs. We do this by compacting the representation of such NOPs in the ROBs.

CF-Naive — This is the mechanism in the original Core Fusion proposal. NOPs inserted at the end of fetch for fetch group alignment occupy one ROB entry each, taking up to two ROB slots per core per fetch group. The top row of Figure 4 shows possible ways in which an instruction pair can be represented in a core's ROB according to this scheme.

CF-Compact — In this organization, an attempt is made to lessen storage overhead for the NOP+NOP pair case as follows: (i) Legitimate instruction pairs in each core take up two ROB slots as usual. (ii) A NOP+instruction pair is encoded by systematically storing the instruction in the first ROB slot, followed by the NOP. This is true even for a misaligned fetch where, strictly speaking, the inserted NOP should precede the instruction; notice that such reordering does not change the program semantics in any case. (iii) A NOP+NOP pair is encoded by a single NOP in the first ROB slot, in which case the second ROB slot becomes the first ROB slot for the next pair. When pre-commit and ROB heads encounter a pair for which the first ROB slot contains a NOP, they can easily recognize that the NOP encodes a NOP+NOP pair (case iii), and when appropriate they move one slot forward (instead of the usual two) in order to process the next pair. Thus, the Compact configuration halves the ROB storage requirement for the NOP+NOP pair case. The second row of Figure 4 shows possible ways in which an instruction pair can be represented in a core's ROB according to CF-Compact.

CF-Extended — In this organization, each ROB slot is extended with one NOP bit. (i) Legitimate instruction pairs in each core still take up two ROB slots as always; their NOP bits are set to zero. (ii) A NOP+instruction pair is encoded in a single ROB slot, by having the instruction take up the ROB slot and setting the NOP bit. (iii) The NOP+NOP pair is also encoded in a single ROB slot, by storing a NOP in the ROB slot proper and setting the NOP bit as well. Thus, in the Extended configuration, pairs with one or two NOPs (cases ii and iii) can be represented with a single ROB slot. The third row of Figure 4 shows possible ways in which an instruction pair can be represented in a core's ROB using this scheme.

CF-CompactExtended — In this configuration, we attempt to reduce the storage overhead even further for the NOP+NOP case. Each ROB slot is again extended with one NOP bit as follows:

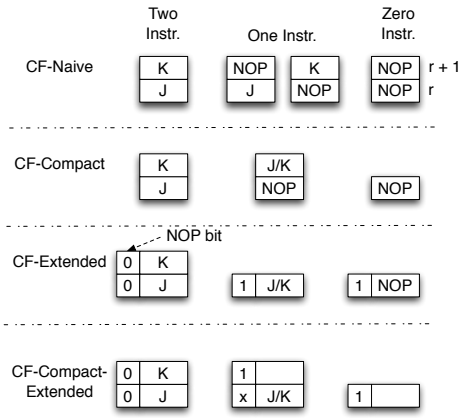

Figure 4: Possible ways to encode instruction pairs, including any inserted NOPs at the end of the collective fetch phase, in the original (CF-Naive) and proposed ROB management schemes.

(i) As before, legitimate instruction pairs in each core take up two ROB slots; their NOP bits are set to zero. (ii) A NOP+instruction pair is encoded by having the instruction take up the first ROB slot and setting the NOP bit for the *next* ROB slot, which becomes the new tail of the ROB. (iii) The NOP+NOP pair is encoded, by setting the NOP bit for the *current* ROB slot, without advancing the ROB tail. Thus, a NOP+NOP pair does not really occupy an ROB slot. (In situations where there are back-to-back NOP pairs, the first NOP pair is represented by setting the NOP bit of the ROB entry, and the second NOP pair is represented by inserting an actual NOP instruction in the ROB entry.) The bottom row of Figure 4 shows possible ways in which an instruction pair can be represented in a core's ROB according to this scheme.

At retirement, we must distinguish between the NOP+instruction and the NOP+NOP scenarios. While retiring instructions, the head slot of the ROB is first checked. If it contains an instruction, and the NOP bit is not set, we retire the instruction, move the head pointer, and check the next slot. If the second slot also contains an instruction and the NOP bit is not set (in the case of a legitimate instruction pair), we retire the instruction and advance the head pointer by one. If, however, the NOP bit is set for the second slot, this means that the bit was set as a result of a NOP+instruction pair, so we simply clear the NOP bit, but do not advance the ROB head pointer. On the other hand, while checking the first slot, if we find the NOP bit to be set, this means that the bit was set because of a NOP+NOP pair. In such a case, we clear the NOP bit and do not move the head pointer to the next subsequent ROB entry. Finally, recall that a ROB slot containing a proper NOP instruction also encodes a NOP+NOP case. In that case, assuming that the NOP bit was already reset during the last cycle (corresponding to the earlier NOP+NOP pair), the head slot is retired as a second NOP+NOP pair.

3.1 Branch Handling

Two details need to be addressed for the space-saving configurations to handle branches correctly. Notice that the number of ROB entries across cores is no longer in sync, because cores with NOPs will now try to encode them more efficiently. This presents a problem upon branch misprediction handling, for those cores which do not hold the offending branch: Where is the ROB slot beyond which everything must be squashed? Fortunately, the original Core Fusion proposal can readily accommodate this. Recall that, to keep the Global History Register consistent across cores, Core Fusion already uses a mechanism very similar to Alpha 21264's Outstanding Branch Queue (OBQ) [16, 19]. In our space-saving configurations, we can easily extend each OBQ entry with a pointer to the appro-

priate ROB entry in each core. That way, upon notification by the FMU of a misprediction, each core can, through the OBQ, easily identify the location beyond which everything must be squashed.

Second, for CF-CompactExtended, in the same ROB slot, we must be careful not to mix a pre-branch NOP bit with a post-branch instruction in those cores which do not directly store the branch instruction. Should we allow that, on a branch misprediction, we would be unable to figure out whether that NOP bit pertains to code before or after the branch. To resolve this, at the time the FMU notifies each core of a branch, if the ROB tail points to an entry with its NOP bit set, the core resets the NOP bit and explicitly stores a NOP in it, unequivocally making it part of the code preceding the branch. (Later at retirement, this special case of a NOP instruction with NOP bit = 0 can be easily recognized and processed.)

3.2 Area Analysis

The CF-Compact organization requires minimal hardware additions over CF-Naive—mainly a means for the pre-commit and ROB heads to recognize the NOP+NOP case. There is no additional raw storage overhead. In the CF-Extended organization, we do increase raw storage in each core's ROB by one bit per ROB slot. Assuming that each ROB entry takes on the order of 50 bits (32-bit PC, 6-bit destination register, 6-bit recycle register, 1-bit valid field, 1-bit exception field, 4-bit interrupt field), the ROB storage overhead of CF-Extended is 2%. (Later, in Section 6, we compensate for this overhead by providing CF-Naive and CF-Compact with extra ROB slots.) We also add six bits to each OBQ entry, for a total of 72 bits per core. The CF-CompactExtended has the same additional overhead as the CF-Extended setup.

4. INSTRUCTION STEERING

Instruction steering must strike a delicate balance between minimizing remote operand communication and making effective use of all cores. Excessive operand co-location in any one core reduces remote operand communications at the expense of load balancing. Conversely, excessive load balancing may result in high communication overheads. Optimal instruction steering is provably NP-complete [37]. Although several heuristics have been proposed and compared against each other, there is no known way to establish a tight upper bound to instruction steering performance. Moreover, Core Fusion's architecture differs in important aspects from the clustered architectures that served as context to most of these proposals.

Rather than engaging in (yet another) "expert" analysis of a problem of such complexity using strictly human intuition, we propose to take an automated approach. We use genetic programs (GP) [22, 23] to search for a static, high-performance steering policy that suits our context and whose hardware implementation is simple enough to execute within a single processor cycle. To our knowledge, this is the first paper to address instruction steering systematically in this way.

4.1 Original Steering Algorithm

The original Core Fusion steering algorithm is a modified version of the dependence-based algorithm [27]. Each steering link has an instruction queue, drained at a rate of two instructions per cycle (matching the destination core's issue width). If the instruction has one operand, it is sent to the least-loaded of the queues whose core has a *copy* of the source operand or producer instruction. If the instruction has two operands, it is sent to the least-loaded queue whose core has both; if no such core exists, it is sent to the least-loaded queue whose core contains a copy of either $source_1$ or $source_2$. This algorithm considers dependencies, copies, and load balancing, with several levels of decisions.

Algorithm 1 Improved instruction steering policy for CF-CompactExtended, derived via genetic programming.

```
1:  for all instructions in the fetch group
2:    if instruction contains both source1 and source2 operands then
3:      steeredCluster ← core where source1 operand produced
4:    else
5:      if instruction contains a source1 operand only then
6:        steeredCluster ← core where source1 operand produced
7:      else
8:        steeredCluster ← least-loaded core (at beginning of cycle)
9:      end if
10:   end if
11: end for
```

4.2 Steering Policy Derivation

4.2.1 Genetic Programming Setup

For the GP mechanism to explore the design space of tree-based steering policies, we first codify a set of 28 functions (which make decisions) and 18 terminals (which assign a particular value for the destination core). These functions and terminals are derived from previously-proposed steering policies. Example functions include checking the type of an instruction (e.g., if it is a branch), determining the number of valid source operands the instruction has (0, 1, or 2), checking to see if each source operand value is ready, or evaluating if the instruction is dependent on a load. Examples of terminals include the least-loaded core (i.e., the core with the least number of instructions in flight), the core that produced either the first source operand or the second one, a random core, the core with the least copy bandwidth filled, or the least-loaded core that contains a copy of both source operands.

We use these states and terminals to come up with an initial random population of 100 steering policies. These are then evaluated on a training set of three integer and three floating-point applications from the MinneSPEC suite [21],[1] using IPC as a fitness function, as measured in detailed simulation models (see Section 5). We select parent policies for the next generation using a fitness-proportional selection methodology called tournament selection, coupled with elitist selection [23]. The parent policies are then evolved using crossover and mutation to get the next generation [22, 23, 24]. The new generation of steering policies is again evaluated on our training set, and so forth.

We stop the training phase after evolving 200 generations, at which point, convergence is observed. Among the policies that perform within 2% of the best-performing policy, we manually select one whose hardware implementation is simple enough that its latency is less than one clock period. This then becomes our policy of choice for our final evaluation (see Section 6). We apply this process separately for the CF-Naive, CF-Compact, CF-Extended and CF-CompactExtended configurations (see Section 5), yielding four steering policies that are best optimized for each case. The rest of this section discusses the policy for CF-CompactExtended.

In order to consider the feasibility of the hardware implementation, we correlate the tree structure to the expected hardware design as a rough approximation—a larger number of terminal nodes will require a larger hardware area, whereas a larger tree depth requires not only greater area, but greater latency as well. Figure 5 shows a scatter plot of the top ten policies, plotting their tree depth versus IPC. Of these, we select the policy in the upper left corner, mini-

[1]The MinneSPEC suite features reduced versions of the full SPEC CPU 2000 [10] applications, which we use in our final evaluation (see Section 5.2). To avoid overfitting, we ensure that (a) the number of applications used in our final evaluation is significantly larger; (b) the applications used for training are simulated for different regions of code than in our final evaluation; and (c) we use reduced input sets for training.

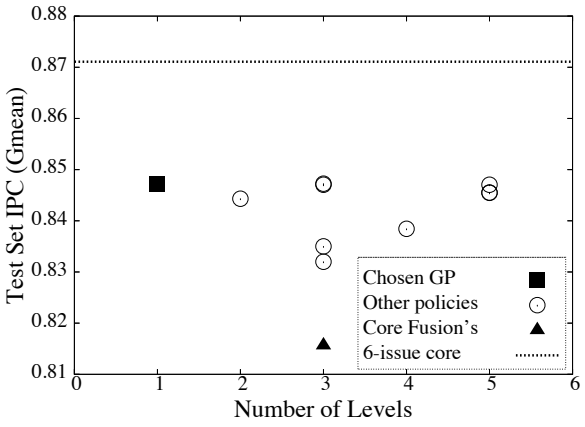

Figure 5: Scatter plot of the top 10 GP-derived steering policies for CF-CompactExtended. The number of levels acts as a corollary to the expected latency of the policy.

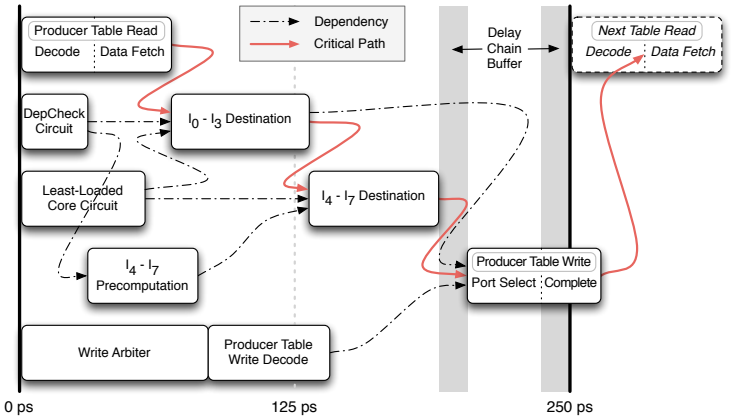

Figure 6: Breakdown of the steering cycle, to scale for a 22 nm process (Borkar scaling). Arrows indicate data dependencies. The gray bars illustrate the buffer used to tolerate the susceptibility of delay chains to process variation. (A delay chain is used to trigger the write port select.)

mizing complexity with near-peak performance. As we will see in Section 4.3, simplicity will be important to make steering implementable in an aggressive 4 GHz, 22 nm pipeline design.

4.2.2 *Selected Algorithm*

Algorithm 1 lists the procedure for our selected policy. For every instruction that needs to be steered, the algorithm determines the core where its source operands are produced. If they happen to be produced in the same core, we steer the instruction to this common producer core. If not, we pick the core where the first operand is produced, and steer the instruction there. If the instruction does not have source operands, we steer it to the least-loaded core.

The resulting algorithm has a number of interesting features:

- Consumers always follow the producer instruction, even if an operand replica may be available elsewhere. This contradicts earlier proposals for instruction steering in the context of clustered processors [1, 8, 30]. While following replicas may improve load balancing, we identified situations where this practice is counter-productive. Consider, for example, an instruction I_1 that produces a value consumed by otherwise independent instructions I_{2a} and I_{2b}. Instruction I_3 uses the values produced by both I_{2a} and I_{2b}. If I_{2a} and I_{2b} are steered to different cores, I_3 will be forced to receive at least one of its operands from a remote core. If, however, both I_{2a} and I_{2b} are steered to I_1's core, steering I_3 to that same core easily avoids the costly remote operand communication.
- The cores' load is not taken into account, except when there are no operands to follow. This also contradicts some of the earlier proposals of instruction steering [1, 3, 8]. Notice that the way Core Fusion does instruction steering already provides a form of load balancing, by assigning a maximum of two instructions per core, per cycle.[2] In this context, the GP procedure did not find that monitoring a core's load was productive in the general case.
- Algorithm 1, as depicted, favors $source_1$ over $source_2$ for two-source-operand instructions (Step 3). Experiments show that picking either operand yields the same performance. The GP did not generate any superior algorithms that checked both source operands (e.g., algorithms that check the load of each operand's core [1]), or even algorithms that picked a source operand randomly. Thus, we favor systematically picking the same operand because it simplifies hardware (see Section 4.3).
- The algorithm is simple, which leads to a hardware implementation that can meet high-speed timing constraints.

[2] If the steering algorithm assigns more than two instructions from the same fetch group to any one core, steering for subsequent fetch groups stalls until all instructions from the current fetch group are injected. This was introduced in the original Core Fusion proposal as a means to achieve a simpler and more feasible hardware implementation.

4.3 Hardware Implementation

Instruction steering is a critical mechanism that cannot be easily pipelined, as steering decisions typically must know the decisions for immediately-preceding instructions. As a result, it is important to demonstrate that all instructions within a fetch group can be processed within a single cycle. Even with as few decision tree levels as our chosen policy (see Figure 5), we find that steering fits into a single cycle with little slack. Therefore, simplicity is of the essence.

Steering eight instructions using Algorithm 1 requires two operations: (a) determining which cores contain the registers that $source_1$ of each instruction references, and (b) finding the core with the least number of instructions pending for issue. To complete successfully within the given clock period, part (a) is split into two steps—determining the cores containing the desired registers at the beginning of the cycle (before any of the current instructions are steered), and then modifying these destinations based on read-after-write dependencies within the fetch group.

Our implementation requires four major components:

- a **producer table**, mapping architectural registers to cores,
- a **least-loaded core circuit**, which compares issue queue occupancies amongst the four cores,
- a **dependency-checking circuit** that tracks read-after-write dependencies within the fetch group, and
- a **destination core circuit**, for aggregating the information from the first three components and determining which core to send the instructions to.

All combinational logic in the steering mechanism was designed using static CMOS logic, optimized to reduce logic depth. We implemented this logic using a NAND-NAND topology. We parallelize many of the basic building blocks within the steering stage; Figure 6 shows our sequence of operations. We elaborate on the producer table and the destination core circuit in the following sections, as they fall on the critical path of the steering circuit. An area and delay analysis is presented in Section 4.4.

4.3.1 *Producer Table*

The producer table is a 32x2b SRAM structure with eight read and eight write ports. Each entry holds the ID of the core that produces the corresponding architectural register. Within a single

clock cycle, the steering management unit (SMU) must read from the table and write back new values before the next read occurs.

The large number of read and write ports required for this design would create an unreasonably large capacitance on the data-storing nodes of a traditional SRAM cell, affecting access times as well as write stability. The need for fast read and write latencies forces us to abandon the 6T SRAM cell and use a modified version. An inverter is added to each bitcell to buffer the output, isolating all of the read ports from the data-storing nodes. Dedicated *set* and *reset* transistors are also added to every cell to enable fast writes. In all, these changes allow us to remove the bitlines and wordlines found in typical SRAM structures, thus removing the latency of charging and discharging highly-capacitive wires. Though this design is less dense than a traditional SRAM structure, given the criticality of latency minimization for this application and the relatively small storage requirements (only 64 bits), this is a worthwhile trade-off.

The producer table read begins at the start of the SMU cycle. From the pre-decode pipeline stage, each core front-end will provide two $source_1$ register specifiers to the SMU (one for each instruction), which will be used to index the table. Once the address is decoded, the core ID is fetched. To ensure that setup and hold times are met for the registers at the end of the SMU steering stage, transparent Earle latches (which are off the critical path) are used, allowing the table to be re-used while the clock is low.

Data write-back technically starts with the write arbiter, which resolves write-after-write register dependencies within the same fetch group. The arbiter grant signals are then processed by the write decoders, to determine which data value to send to the *set*/*reset* transistors for the modified rows. All this can be completed in parallel with the critical path, as the destination register for each instruction is known at the start of each cycle. Once the destination core calculation has completed, a delay chain triggers the write enable pulse, asserting the appropriate *set*/*reset* signals on the bitcells themselves. After a certain pulse width, the bitcell will have passed its switching threshold, and will settle to its intended value even if the enable signal is de-asserted before the bitcell completes its transition. We exploit this completion stage for improved latency, as discussed in Section 4.4.2.

4.3.2 Least-Loaded Core & Dependency-Checking Circuits

The least-loaded core is calculated using the issue queue occupancy values at the beginning of the cycle. In case of a tie, the core with the lower ID is statically chosen (using less-than-or-equal-to comparators). The dependency-checking circuit identifies read-after-write dependencies within the current fetch group, comparing $source_1$ registers with the destination registers. This circuit consists of 28 5-bit comparators. Neither circuit falls on the critical path (see Figure 6).

4.3.3 Destination Core Circuit

For our steering algorithm, an instruction can be sent to one of two places: (a) if the first source operand exists, to the core that produced that value; otherwise (b) to the least-loaded core. The producer table provides a stale value for the core containing the first source operand of each instruction, which we will refer to as S_n, where I_n is the *n*th instruction being steered in the current cycle. We also know whether the instruction has a valid $source_1$ operand (from the pre-decode stage), which we will refer to as V_n.

For instruction I_0, computing the destination core C_0 simply involves choosing either the least-loaded core (LLC) or S_0:

$$C_0 = (LLC \wedge \neg V_0) \vee (S_0 \wedge V_0) \quad (1)$$

Subsequent instructions involve more complex logic, as they may also have read-after-write dependencies on instructions steered in the same fetch group—remember that the value of S_n is not updated after each steering decision. To fit within a cycle, we choose to "unroll" the destination core equations and use deeper Boolean logic gates—while this does replicate some calculations, we reduce the total number of logic levels (and thus, signal transitions) needed to compute each core. This is conceptually similar to how outputs are constructed in a carry-lookahead circuit. For example, the logic to steer I_1 is:

$$C_1 = (LLC \wedge \neg V_1) \vee (S_1 \wedge V_0 \wedge \neg I_1 DI_0) \vee (C_0 \wedge I_1 DI_0) \quad (2)$$

where $I_1 DI_0$ is whether the $source_1$ operand of I_1 is produced by I_0. If we substitute Equation 1 in for C_0, we can flatten the gates for C_1, removing the need for serialization:

$$\begin{aligned} C_1 =& (LLC \wedge \neg V_1) \vee (S_1 \wedge V_0 \wedge \neg I_1 DI_0) \\ & \vee (LLC \wedge \neg V_0 \wedge I_1 DI_0) \vee (S_0 \wedge V_0 \wedge I_1 DI_0) \end{aligned} \quad (3)$$

Likewise, this technique can be applied up to instruction I_3.

While instructions I_4 through I_7 could also be calculated in this flattened manner, the corresponding circuits would become relatively large (for example, I_7 would involve the sum of 256 logic products, some of which include 9 logic terms, themselves requiring four gates to compute). Instead, we compute the destination cores for these instructions once the destination cores C_0 through C_3 are computed. While this does perform a serialization step, we find that we can significantly reduce the area while still meeting timing constraints (I_7 now contains 48 logic products). Furthermore, many of the control signals in these products can be partially pre-computed off of the critical path.

4.4 Circuit Analysis

The entire steering mechanism has been implemented at the transistor level, and its functionality and performance have been verified using HSPICE simulations [34]. Given the tight timing constraints, low-V_t transistors are used for every circuit except for the SRAM cells in the producer table and keepers on dynamic gates, in order to minimize the latency of all the components. All simulations were run using custom transistor-level netlists, with transistor models provided by IBM for their 45 nm 12SOI process [36].

Every transistor was carefully sized in order to balance drive strength with parasitics. Gate capacitances and diffusion capacitances are accounted for in the transistor models themselves, and accurately scale with transistor size, gate topology, and fanout. An additional 4 fF of load capacitance is added to every gate to model wiring parasitics. This value has been extracted from previously-fabricated designs in this process technology.

We assume a 22 nm process technology for all of our steering circuitry. ITRS forecasts are used to used to scale delays from 45 nm to 22 nm technology (conservatively using planar bulk transistor models as opposed to multi-gate transistors, and using 2014 forecasts for 24 nm) [17]. We also use a conservative 22 nm scaling factor from Borkar [2].

4.4.1 Area

In total, the steering mechanism uses 21,340 MOSFET transistors. This represents a very tiny area overhead within a processor. Table 1 breaks down the transistor count by each component. Many of the larger units (e.g., dependency-checking circuit, write arbiter) are the result of logical unrolling, where common sub-circuits have been repeated to improve performance.

4.4.2 Delay

Figure 6 not only shows the timings of each component to scale, but it also illustrates the data dependencies between these blocks. The critical path travels through the producer table read, destination core computation, and producer table write. Unless otherwise noted, all latencies in this section use the more conservative scaling

Table 1: Transistor counts for the steering circuit components.

Circuit	Transistor Count
Producer Table	11872
Least-Loaded Core	914
Dependency Check	2520
Write Arbiter	2706
$I_0 - I_3$ Dest. Core	776
$I_4 - I_7$ Dest. Core	1564
$I_4 - I_7$ Precompute	412
Earle Latches	576
TOTAL	21340

Table 2: Steering circuit latencies (ps). 45 nm values are only provided for circuits simulated using the IBM 12SOI model in HSPICE. Critical path circuits italicized.

Circuit		45 nm	32 nm		22 nm	
			Conservative	Traditional	Borkar	ITRS
Producer Table Read	*Decode*	40.50	36.99	34.90	34.09	30.08
	Data Fetch	41.45	37.86	35.72	34.89	30.78
Producer Table Write	Decode	65.65	59.96	56.58	55.26	48.75
	Port Select	40.00	36.53	34.47	33.67	29.70
	Completion	32.07	29.29	27.64	26.99	23.82
Least-Loaded Core		67.72	61.85	58.36	57.00	50.29
Dependency Check		36.83	33.64	31.74	31.00	27.35
Write Arbiter		101.79	92.96	87.72	85.68	75.59
$I_0 - I_3$ Destination Core		74.39	67.94	64.11	62.62	55.24
$I_4 - I_7$ Destination Core		69.89	63.83	60.23	58.83	51.90
CRITICAL PATH		266.23	243.15	229.44	224.10	197.70

proposed by Borkar [2], and target a 22 nm technology. A conservative FO4 ring oscillator scaling and the traditional NMOSFET scaling, both from ITRS, are used to provide 32 nm latencies as a reference. Table 2 summarizes these numbers, and includes latencies for scaling methods in both 32 nm and 22 nm technology.

Along the critical path, a total of 224.10 ps is required for the circuit to execute. Since the critical path includes the use of a delay chain to trigger the producer table write (see Section 4.3.1), extra timing margins must be left in place. Increased process variations at smaller technologies may shift the trigger time in either direction. To account for this, the extra time left over in the cycle (currently 25.90 ps) is partitioned equally on both sides of the table write. Setup times for the end-of-stage registers are masked by the write, as they only require the output of the destination core circuits.

An important aspect of our circuit timing is the overlap between the producer table write completion and the read decode for the next cycle, as seen in Figure 6. We note that the data inside the table must be stable for reading only after the read ports have finished decoding, as the bitcells are not accessed up to that point. We can take advantage of our observation in Section 4.3.1 by asserting write enable just long enough to cross the switching threshold of the bitcell. Write completion can therefore overlap with the read decode of the next cycle without introducing any data hazards, since the decode latency is longer than the completion time.

5. EXPERIMENTAL METHODOLOGY

5.1 Architectural Setup

We design our experiments assuming a 22 nm process technology. We increase the L2 cache size from 4 MB to 8 MB, and increase associativity from 8 ways to 16. However, we do not change the pipeline structure of the cores from the ones described in İpek et al. [16] for 65 nm. This is in line with recent trends observed in industry [13, 15, 18]: As process technology scales, frequency has stopped scaling beyond the 4 GHz range, and the additional slack achieved in the clock period is typically put to use not by redesigning the pipelines, but by lowering the chip supply voltage, thereby consuming less power per core and enabling more cores to be simultaneously active on the chip with the smaller feature size.

All our experiments have been carried out using a highly-detailed and heavily-customized version of the SESC simulator [29]. We evaluate the performance of the *CF-Naive*, *CF-Compact*, *CF-Extended*, and *CF-CompactExtended* configurations, with both GP-generated steering (*+GP*) and the original Core Fusion steering solution (see Section 4.1). To compensate for extending each ROB entry by one NOP bit, we allocate two additional ROB entries to CF-Naive and CF-Compact. Table 5 shows the various configurations evaluated, and Table 3 describes the microarchitecture of the two-issue base core used in Core Fusion.

We use the performance of an area-equivalent monolithic processor (*Monolithic*) as reference. It is a six-issue out-of-order processor with three times the amount of core resources as the two-issue base core, but four times the size of the base core's L1 cache, branch predictor, and BTB. We model wake-up and selection delays in the base core to be one cycle each, and extrapolate such delays for the six-issue core to 3 and 2 cycles, respectively, using trends presented in the literature [11, 12, 26]. We assume that wake-up and select are fully pipelined for the six-issue core [33].

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; see Table 4). All configurations are clocked at the same speed.

5.2 Applications

Since we focus on improving the performance of sequential applications, we evaluate our proposal using the SPEC CPU 2000 application suite for the MIPS ISA [10]. We use the *ref* input sets, and simulate one billion instructions for each workload, skipping the initialization phase. The SimPoint toolset [32] was used to ensure that the simulated instructions were a representative sample of the entire application.

5.3 Core Fusion Overheads

Area — We estimate the area overhead of all original Core Fusion additions pessimistically. To calculate the area overheads from wiring, we use the wiring area estimation methodology described by Kumar et al. [25], assuming a 22 nm technology and global wires with a 81 nm wire pitch, as projected by ITRS [17]. The delay of global wires with repeater links is obtained from Chen et al. [5]. Accordingly, we calculate the area of fetch wiring (92 bits/link) to be 0.21 mm^2, the area of rename wiring (250 bits/link) to be 0.60 mm^2, the area of the operand crossbar (80 bits/link) to be 1.09 mm^2, and the commit wiring overhead (20 bits/link) to be 0.02 mm^2. The wiring area overhead amounts to 1.92 mm^2 for a single fusion group of four cores. Scaling from the original paper, the extra cache tags, copy-in/copy-out queues and the bank predictors to be 0.24 mm^2, 0.13 mm^2, 0.08 mm^2 and 0.12 mm^2, respectively, for a fusion group of four cores. This amounts to a total of 0.57 mm^2 for the group. Adding these to the wiring overheads, we estimate an overall area overhead to be 2.49 mm^2 per fusion group for the Core Fusion architecture, including our proposed extensions.

Assuming a reticle-limited 400 mm^2 die size,[3] one fusion group takes up an area of 49 mm^2. For the original Core Fusion architecture, with two fusion groups, the area overhead (4.98 mm^2) is less than half that of a two-issue core (12.25 mm^2)—smaller than the overhead reported in İpek et al. [16], which targeted a (now obsolete) 65 nm technology node. The result is reassuring that the overhead of our enhanced Core Fusion solution remains manageable.

[3] Some server multicore chips today, such as Intel's Xeon or IBM's Power7, have a die area in excess of 500 mm^2 [6, 14].

Table 3: Parameters of the base two-issue core.

Frequency	4 GHz
Fetch/issue/commit width	2/2/2
Int/FP/AGU/Br Units	1/1/1/1
Int/FP Multipliers	1/1
Int/FP issue queue size	16/16 entries
ROB (reorder buffer) entries	48
Int/FP registers	32 + 40 / 32 + 40 (Arch. + Phys.)
Ld/St queue entries	12/12
Bank predictor	2,048 entries
Max. br. pred. rate	1 taken/cycle
Max. unresolved br.	12
Br. mispred. penalty	7 cycles min., 14 cycles when fused
Br. predictor	Alpha 21264 (tournament)
RAS entries	32
BTB size	512 entries, 8-way
iL1/dL1 size	16 kB
iL1/dL1 block size	32 B/32 B
iL1/dL1 round-trip latency	2/3 cycles (uncontended)
iL1/dL1 ports	1 / 2
iL1/dL1 MSHR entries	8
iL1/dL1 associativity	direct-mapped/4-way
Memory Disambiguation	Perfect
Coherence protocol	MESI
Consistency model	Release consistency

Table 4: Parameters of the shared L2 and DRAM subsystem.

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

Table 5: Configurations evaluated.

CF-Naive	4×2-issue with (48+2)-entry ROB / core
CF-Naive+GP	— + custom GP steering
CF-Compact	4×2-issue with (48+2)-entry ROB / core
CF-Compact+GP	— + custom GP steering
CF-Extended	4×2-issue + 48-entry Extended ROB / core
CF-Extended+GP	— + custom GP steering
CF-CompactExtended	4×2-issue + 48-entry Extended ROB / core
CF-CompactExtended+GP	— + custom GP steering
Monolithic	Area-equivalent monolithic core (Section 5.1)

Delay — To calculate the wire delays we assume the worst-case cross-core communication distance. From Friedman [9], we find the delay of global wires with repeater links to be 31.8 ps/mm in 22 nm technology. As aforementioned, one fusion group of four cores takes up an area of 49 mm^2. We expect to floorplan the processor such that all cross-core communicating logic will sit in the quadrant of the core—1.75 mm on each side—closest to the center of the chip. Therefore, the worst-case cross-core communication distance will be 7 mm. Assuming the widest possible link, the worst-case communication distance can be traversed in one cycle. However, we conservatively set the communication latencies for fetch, rename, operand transfer, and commit to be two cycles each.

6. EVALUATION

Figures 7 and 8 show the performance obtained by the various CF configurations considered in this study, relative to Monolithic on SPEC CPU 2000 applications. From Figure 8, we see that CF-CompactExtended+GP is the CF configuration that performs best: it improves performance over CF-Naive by 11.5%, and it is within 98% of Monolithic for the SPEC-Int applications. For SPEC-FP, we improve performance over CF-Naive by 19%, and we are again within 98% of Monolithic. In contrast, the performance of CF-Naive is only within 88% and 82% of Monolithic for SPEC-Int and SPEC-FP, respectively. Note that we see some speedups over the 6-issue baseline as in fused mode, Core Fusion can provide 8-wide issue.

As Figure 8 shows, the results for SPEC-Int applications suggest a somewhat synergistic effect between the proposed mechanisms for distributed commit and instruction steering. (SPEC-FP applications, which typically respond very well to mechanisms that exploit instruction-level parallelism, benefit primarily from the ability of the extended ROB to support more in-flight instructions.) These two mechanisms attack different stages of the pipeline: The retirement scheme was added to improve throughput in the fetch stage of the pipeline, by reducing NOP insertions and thereby increasing useful instructions being fed into the subsequent pipeline stages. The improved instruction steering algorithm focuses on improving throughput in the execution stage, by increasing the probability of producers and consumers being co-located in a core. Naturally, only freeing a bottleneck in one stage while allowing the other to persist will not improve performance as effectively. When the two techniques are coupled together, the opportunity to co-locate dependent instructions increases, as does the number of useful instructions in the pipeline. By co-locating more dependent instructions, we reduce the wait time involved for consumers to issue after their producers complete, thereby improving issue rate, and hence commit rate. An increase in commit rate frees up ROB entries faster, which in turn helps prevent fetch stalls, which increases the ROB occupancy. This positive feedback leads to a more free-flowing pipeline, due to a decrease in pipeline stalls in both stages, resulting in better overall performance for CF-CompactExtended+GP than the sum of the two effects separately (CF-CompactExtended and CF-Naive+GP).

Next, we analyze CF-CompactExtended+GP in more detail. For brevity, we provide analysis for SPEC-Int applications only—the findings are consistent with the results for SPEC-FP as well.

6.1 Inside Distributed Commit

Recall that the new retirement scheme was added to the baseline architecture (CF-Naive) to reduce unnecessary NOP insertions in the ROB. If the number of NOPs inserted is reduced, the ROB can be filled with more useful instructions, which may result in more in-flight instructions (and hence higher potential for exploiting instruction-level parallelism). This is particularly important for handling long-latency load operations: With more NOPs filling up the ROB, there are fewer in-flight loads in the pipeline, and thus the possibility of exploiting memory-level parallelism (MLP) by overlapping long-latency loads is smaller. Higher MLP often results in better overall performance [28]. Figures 3 and 9 show that the new retirement scheme indeed increases the effective use of the ROB with respect to CF-Naive, and in particular it improves memory-level parallelism.

6.2 Inside Instruction Steering

In order to understand the effect of the GP-generated steering algorithm, we introduce a metric called *communication penalty*. Every cycle, for each core, we count the number of instructions that are issued. For every free issue slot that is not filled, we check to see if an instruction is not issued due to a source operand not being available. For such instructions, we check if the producer has already completed execution in another remote core. This means that, if the producer and consumer were co-located, the consumer could have issued immediately after the producer; but now it has to wait for the copy instruction to arrive at the consumer core. If an issue slot is not filled for such a scenario, we assign a communication penalty of one. Thus, the maximum penalty a core can be assigned each cycle is two, as each core has two issue slots it can fill every cycle.

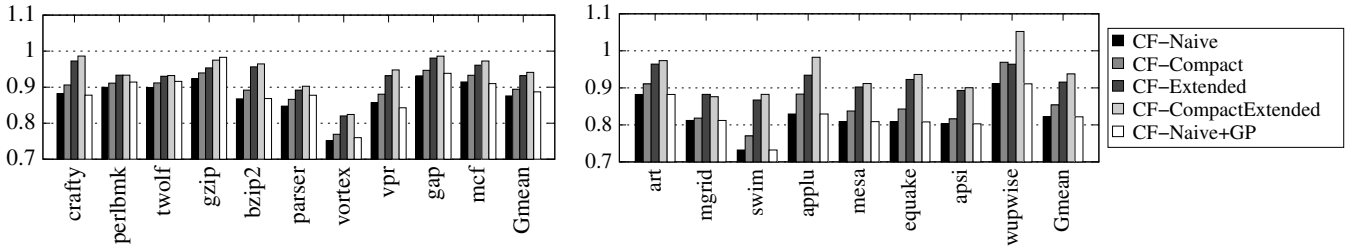

Figure 7: Performance resulting from applying our proposed configurations in isolation, normalized to that of an area-equivalent monolithic core configuration (Section 5.1), for SPEC-Int (left) and SPEC-FP (right) applications.

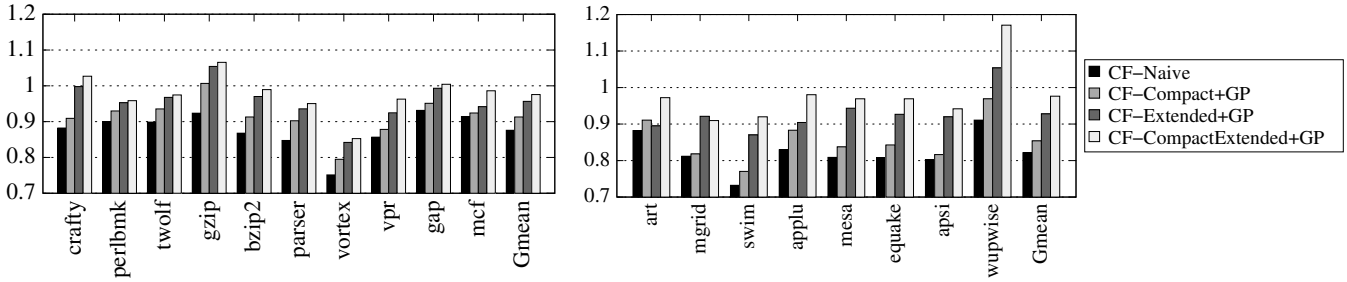

Figure 8: Performance resulting from the synergy between improved commit and improved steering, normalized to that of an area-equivalent monolithic core configuration (Section 5.1), for SPEC-Int (left) and SPEC-FP (right) applications.

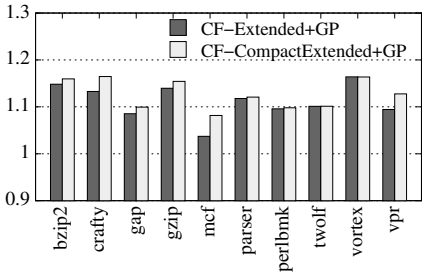

Figure 9: Increase in memory-level parallelism (MLP) when running SPEC-Int applications, relative to CF-Naive.

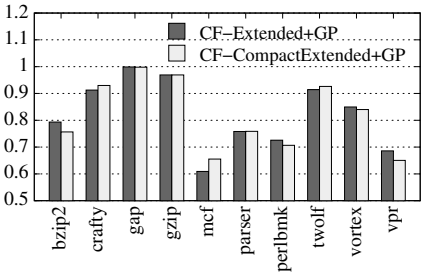

Figure 10: Stall time due to remote operand communication for SPEC-Int applications, relative to CF-Naive.

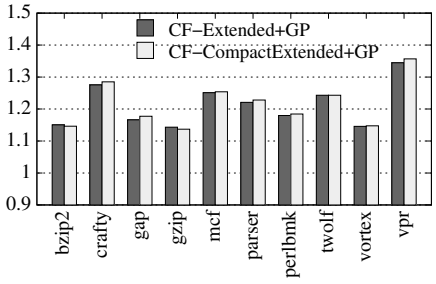

Figure 11: Increase in locally co-located producer-consumer pairs for SPEC-Int applications, relative to CF-Naive.

Figure 10 gives us the communication penalty observed relative to CF-Naive. The results show that CF-CompactExtended+GP incurs a lesser penalty than CF-Naive for all applications; the improvement is rather significant in some of them. This is because the steering algorithm in CF-CompactExtended+GP improves the probability of co-locating producers and consumers (Figure 11). By reducing communication delays among producers and consumers in this way, we issue instructions at a faster rate, which leads to improved performance.

7. RELATED WORK

Reconfigurable multicore architectures — TFlex [20] is a novel reconfigurable architecture comprised of distributed composable tiles that communicate with each other via control and operand networks. While TFlex can use its dataflow-based ISA to track dependent instructions, this custom ISA requires extensive binary modifications and additional compiler support.

Federation [35] is another reconfigurable architecture along the lines of Core Fusion that proposes fusing a pair of scalar cores into a two-way out-of-order processor. Salverda and Zilles demonstrate the fundamental challenges in reconfiguring in-order cores into larger cores, on demand, even under ideal conditions, as it necessitates very complex instruction steering hardware [31].

Clustered architectures — One of the very first approaches to clustering involved designing a microarchitecture that would simplify the wakeup and selection logic by introducing multiple in-order issue queues, and placing chains of dependent instructions in them [27].

Another early approach extended the floating-point cluster to execute simple integer instructions, resulting in a clustered architecture [3]. The steering schemes used are a combination of reducing workload imbalance and inter-cluster communications.

The Multicluster architecture introduced a dynamically-scheduled partitioned architecture which allowed the register file of one cluster to be accessed by instructions being executed on another cluster, by distributing an instruction to multiple clusters [7].

Canal et al. [4] have proposed various dynamic code partitioning mechanisms for clustered microarchitectures. The basic idea of dynamic code partitioning is to steer instructions present in either a *LdSt slice* or a *Br slice* within a register dependence graph to the same cluster. They also experimented with static steering algorithms, and their results indicate that dynamic steering algorithms outperform static cluster assignments in general. Moreover, static cluster assignment tends to be less flexible and needs compiler support as well as extensions to the ISA.

Baniasadi and Moshovos investigated various non-adaptive and adaptive instruction steering techniques for quad-cluster dynamically-scheduled superscalar processors [1]. They studied the sensitivity of steering heuristics to relevant architectural parameters, such as inter-cluster communication latency and pipeline depth, and found that performance is much more sensitive to inter-cluster communication.

8. CONCLUSION

In this paper, we attack two important inefficiencies of Core Fusion—collective commit storage and cross-core operand com-

munication overheads. We introduce a new commit scheme that reduces the storage requirement associated with NOP insertions, thereby increasing the number of ROB entries available to useful instructions. This in turn increases the number of in-flight load instructions, allowing a greater possibility of overlapping long-latency loads, and improving memory level parallelism in the system. The improved performance is achieved with virtually negligible area and delay overheads. We also introduce an improved instruction steering algorithm, which reduces the communication penalty by increasing the probability of co-locating producers and consumer instructions. We derive this algorithm systematically using genetic programming, verify the steering circuit implementation using HSPICE, and provide a detailed area and delay analysis. By combining these two techniques, the performance of our improved Core Fusion architecture comes within 98% of that of an area-equivalent monolithic machine for both SPEC-Int and SPEC-FP. The implication is that Core Fusion may effectively render the inclusion of wide-issue cores in future multicore designs unnecessary, since it can match their sequential performance when needed while retaining the ability to offer fine-grain parallelism on demand through its narrow-width base cores.

9. ACKNOWLEDGEMENTS

This research was supported in part by NSF CAREER Award CCF-0545995, NSF Award CNS-0720773, and gifts from IBM, Intel, and Microsoft. Saugata Ghose was supported in part by DoD, Air Force Office of Scientific Research, National Defense Science and Engineering Graduate (NDSEG) Fellowship, 32 CFR 168a.

10. REFERENCES

[1] A. Baniasadi and A. Moshovos. Instruction distribution heuristics for quad-cluster, dynamically-scheduled, superscalar processors. In *MICRO*, 2000.

[2] S. Borkar. Major challenges to achieve exascale performance. In *Salishan Conf.*, 2009. Presentation.

[3] R. Canal, J.-M. Parcerisa, and A. González. A cost-effective clustered architecture. In *PACT*, 1999.

[4] R. Canal, J.-M. Parcerisa, and A. Gonzalez. Dynamic cluster assignment mechanisms. In *HPCA*, 2000.

[5] G. Chen, H. Chen, M. Haurylau, N. Nelson, P. M. Fauchet, E. G. Friedman, and D. Albonesi. Predictions of CMOS compatible on-chip optical interconnect. In *SLIP*, 2005.

[6] EUROSOI Newsletter. IBM's Power7 heats up server competition at Hot Chips. `http://www.eurosoi.org/public/Newsletter_AugSept2009.pdf`.

[7] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. G. Vranesic. The Multicluster architecture: Reducing processor cycle time through partitioning. *International Journal of Parallel Programming*, 27(5):327–356, 1999.

[8] B. Fields, S. Rubin, and R. Bodik. Focusing processor policies via critical-path prediction. In *ISCA*, 2001.

[9] E. G. Friedman. Predictions, challenges, and opportunities in CMOS compatible on-chip optical interconnect. In *STEP Conf. on Optical Computer Developments*, 2008.

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

[11] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, Q1 2001.

[12] M. S. Hrishikesh, N. P. Jouppi, K. I. Farkas, D. Burger, S. W. Keckler, and P. Shivakumar. The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays. In *ISCA*, 2002.

[13] Intel Corporation. First the tick, now the tock: Next-generation Intel microarchitecture (Nehalem). `http://www.intel.com/technology/architecture-silicon/next-gen/whitepaper.pdf`.

[14] Intel Corporation. Intel Xeon 7400 Processor series. `http://www.intel.com/p/en_US/products/server/processor/xeon7000`.

[15] Intel Corporation. Introducing the 45nm Next-Generation Intel Core Microarchitecture. `http://www.citrix.com/site/resources/dynamic/partnerDocs/Intel_45nm-core2_whitepaper.pdf`.

[16] E. İpek, M. Kırman, N. Kırman, and J. F. Martínez. Core Fusion: Accomadating software diversity in chip multiprocessors. In *ISCA*, 2007.

[17] ITRS. International Technology Roadmap for Semiconductors: 2010 update. `http://www.itrs.net/reports.html`.

[18] H. Iwai. Roadmap for 22nm and beyond (invited paper). *Microelectron. Eng*, 86:1520–1528, Nov. 2009.

[19] R. E. Kessler, E. J. McLellan, and D. A. Webb. The Alpha 21264 microprocessor architecture. In *International Conference on Computer Design*, pages 90–95, Austin, Texas, Oct. 1998.

[20] C. Kim, S. Sethumadhavan, M. S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler. Composable lightweight processors. In *MICRO*, 2007.

[21] A. KleinOsowski, J. Flynn, N. Meares, and D. Lilja. Adapting the SPEC 2000 benchmark suite for simulation-based computer architecture research. In *Workshop on Workload Characterization*, Austin, TX, Sept. 2000.

[22] J. R. Koza. *Genetic Programming: A Paradigm for Genetically Breeding Populations Of Computer Programs to Solve Problems - Technical Report*. Stanford University, Computer Science Department, 1990.

[23] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, 1992.

[24] J. R. Koza, M. A. Keane, M. J. Streeter, W. Mydlowec, J. Yu, and G. Lanza. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, 2003.

[25] R. Kumar, V. Zyuban, and D. M. Tullsen. Interconnections in multi-core architectures: Understanding the mechanisms, overheads and scaling. In *ISCA*, 2005.

[26] S. Palacharla, N. Jouppi, and J. E. Smith. *Technical Report: Quantifying the Complexity of Superscalar Processors*. University of Wisconsin - Madison, 1996.

[27] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *ISCA*, 1997.

[28] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A case for MLP-aware cache. In *ISCA*, 2006.

[29] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC simulator, January 2005. `http://sesc.sourceforge.net`.

[30] P. Salverda and C. Zilles. A criticality analysis of clustering in superscalar processors. In *MICRO*, 2005.

[31] P. Salverda and C. Zilles. Fundamental performance constraints in horizontal fusion of in-order cores. In *HPCA*, 2007.

[32] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS*, 2002.

[33] J. Stark, M. D. Brown, and Y. N. Patt. On pipelining dynamic instruction scheduling logic. In *MICRO*, 2000.

[34] Synopsys. HSPICE: Circuit simulation. `http://www.synopsys.com/Tools/Verification/AMSVerification/CircuitSimulation/HSPICE/`.

[35] D. Tarjan, M. Boyer, and K. Skadron. Federation: Repurposing scalar cores for out-of-order instruction issue. In *DAC*, 2008.

[36] The MOSIS Service. IBM 45 nanometer 12SOI CMOS process. `http://www.mosis.com/ibm/12soi/`.

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