Tuning the Continual Flow Pipeline Architecture with Virtual Register Renaming

KOMAL JOTHI and HAITHAM AKKARY, American University of Beirut, Lebanon

Continual Flow Pipelines (CFPs) allow a processor core to process hundreds of in-flight instructions without increasing cycle-critical pipeline resources. When a load misses the data cache, CFP checkpoints the processor register state and then moves all miss-dependent instructions into a low-complexity WB to unblock the pipeline. Meanwhile, miss-independent instructions execute normally and update the processor state. When the miss data return, CFP replays the miss-dependent instructions from the WB and then merges the miss-dependent and miss-independent execution results.

CFP was initially proposed for cache misses to DRAM. Later work focused on reducing the execution overhead of CFP by avoiding the pipeline flush before replaying miss-dependent instructions and executing dependent and independent instructions concurrently. The goal of these improvements was to gain performance by applying CFP to L1 data cache misses that hit the last level on chip cache. However, many applications or execution phases of applications incur excessive amount of replay and/or rollbacks to the checkpoint. This frequently cancels benefits from CFP and reduces performance.

In this article, we improve the CFP architecture by using a novel virtual register renaming substrate and by tuning the replay policies to mitigate excessive replays and rollbacks to the checkpoint. We describe these new design optimizations and show, using Spec 2006 benchmarks and microarchitecture performance and power models of our design, that our Tuned-CFP architecture improves performance and energy consumption over previous CFP architectures by $\sim 10\%$ and $\sim 8\%$, respectively. We also demonstrate that our proposed architecture gives better performance return on energy per instruction compared to a conventional superscalar as well as previous CFP architectures.

Categories and Subject Descriptors: C.1.4 [Processor Architectures]: Single Data Stream Architectures—Pipeline Processors

General Terms: Design, Performance

Additional Key Words and Phrases: Superscalar processors, register renaming, instruction level parallelism, continual flow pipelines, virtual register renaming

ACM Reference Format:

Komal Jothi and Haitham Akkary. 2014. Tuning the continual flow pipeline architecture with virtual register renaming. *ACM Trans. Architect. Code Optim.* 11, 1, Article 11 (February 2014), 27 pages.
DOI: <http://dx.doi.org/10.1145/2579675>

This article is an extension of conference papers [Jothi and Akkary 2013a, 2013b]. Additional material includes (1) a quantitative analysis of performance limitations of the Simultaneous Continual Flow Pipeline (S-CFP) architecture and the performance contribution of the different optimizations applied to tune the S-CFP architecture (Sections 5.1 and 5.2.2); (2) detailed empirical analysis and comparison of energy efficiency of Tuned-CFP to S-CFP (Sections 5.3 and 5.5) and conventional non-CFP (Sections 5.4 and 5.5), for medium to very large core sizes; (3) analysis of the Tuned-CFP performance with 16 L2 data prefetching stream buffers (Section 5.6 and all performance and energy results in the article).

Authors' addresses: K. Jothi and H. Akkary, ECE Department, American University of Beirut, P.O. Box 11-0236, Riad El Solh, Beirut 1107 2020, Lebanon; email: kmj04@aub.edu.lb, ha95@aub.edu.lb.

Permission to make digital or hard copies of part or all 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 show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2014 ACM 1544-3566/2014/02-ART11 \$15.00

DOI: <http://dx.doi.org/10.1145/2579675>

1. INTRODUCTION

The current era of multicore processors presents microprocessor architects a difficult design challenge: how to design a processor core that would give good single thread performance on applications with limited thread-level parallelism, yet consume less power, therefore making it possible to integrate more cores on a single die for high throughput performance.

To improve superscalar processor performance on difficult-to-parallelize applications, architects have been increasing the capacity of reorder buffers (ROBs), Reservation Stations (RS), physical register files, and Load and Store Queues (LSQs) [Smith and Sohi 1995] with every new out-of-order processor core. For two decades, enlarging the instruction buffers has provided good performance improvement. However, this approach is not effective anymore [Agarwal et al. 2000]. First, there is little instruction-level parallelism left to justify further increase in instruction buffer size. Second, trying to reduce execution stalls from cache misses with larger conventional cores is impractical. Current buffer sizes are sufficient for code that hits the L1 data cache and way too small for code that misses the last-level cache. Load latency to the last-level cache on current processors is more than 20 clock cycles, and latency to DRAM, even with on-chip DRAM controller, exceeds hundred cycles. It is impractical to increase the sizes of power-hungry and cycle-critical buffers, like ROBs, RS, and physical register files to the capacity necessary to handle long load latencies to the last-level cache or DRAM.

A different design strategy is to size the instruction buffers to the minimum capacity necessary to handle the common case of L1 data cache hit and to use new scalable out-of-order execution algorithms to handle code that misses the L1 data cache.

Continual Flow Pipeline (CFP) architecture [Srinivasan et al. 2004] was proposed as energy-efficient large instruction window architecture for reducing the impact of data cache misses on performance, without having to increase instruction buffers and physical register files sizes. CFP breaks away from conventional ROB mechanisms for managing speculative out-of-order execution. Instead of using a ROB for tracking, buffering, and sequentially committing one by one the current set of in-flight instructions, also called the *instruction window*, CFP uses bulk commit of execution results and register state checkpoints to recover from branch mispredictions and exceptions [Hwu and Patt 1987; Akkary et al. 2003; Akkary et al. 2004]. Instead of residing completely in the hardware instruction buffers, the instruction window becomes virtual and composed physically of a partial, discontinuous subset of the conventional instruction window. As such, instructions that do not encounter cache misses enter the instruction window, execute, and commit quickly, freeing all of their hardware resources. On the other hand, instructions that depend on cache miss data enter the physical instruction window and wait as long as necessary for their input operands without blocking the execution pipeline.

CFP handles data cache misses as follows. When a load misses the data cache, a poison bit [Srinivasan et al. 2004] is set in the destination register of the load. Load-dependent instructions in the RS are then woken up, as if the load completed. Poison bits propagate through instruction dependences and identify all instructions that depend on the load miss and their descendants. The miss load and its dependents, identified by the poison bits in the ROB, pseudocommit in program order and move from the ROB into a waiting buffer (WB) outside the pipeline. Since dependent instructions do not tie pipeline resources, the core can execute ahead far into the program without stalling due to the cache miss.

When the miss data is fetched, the dependent instructions wake up and replay from the WB into the pipeline to complete their execution. When the WB is emptied

and all miss-dependent instructions complete, independent and dependent instruction results are merged using a flash copy operation in the Retirement Register File (RRF). Execution then resumes normally.

CFP was originally proposed to tolerate long latencies of L2 cache misses that go to DRAM [Srinivasan et al. 2004]. In that work, the miss-independent and miss-dependent instructions execute at different times, based on the timing of the load miss event and the data arrival event. Switching between the two executions is costly because it involves a pipeline flush, making this proposal unsuitable for L1 misses that hit the on-chip cache. In a more recent work, Simultaneous Continual Flow Pipeline (S-CFP) [Jothi et al. 2011] executes the independent and dependent instructions simultaneously to avoid the costly pipeline flush, thus making S-CFP more suitable for first-level data cache misses. However, in S-CFP, many applications or execution phases of applications incur excessive amount of replay and/or rollbacks to checkpoints because of branch mispredictions during CFP execution. This is because CFP recovers from replayed mispredicted branches by rolling back execution to checkpoints taken at the load misses. Excessive replay increases the chance of replaying mispredicted branches and consequently rollbacks to the checkpoints. This frequently cancels any desired improvement resulting from S-CFP handling of L1 data cache misses and can even cause performance degradation.

1.1. Article Contributions

In this article, we focus on the excessive replays and checkpoint rollbacks to improve CFP performance and reduce its power consumption. For this, we use a novel Virtual Register Renaming (VRR) substrate [Sharafeddine et al. 2013; Jothi and Akkary 2013b] and fine-tune the replay policies to mitigate excessive replays and rollbacks to the checkpoint.

The key contributions of this work over prior CFP work are:

- On previous CFP architectures, all load miss-dependent instructions have to be moved into the WB and then replayed once the load miss is moved into the WB. This is necessary because the miss load releases its renamed destination register when it pseudocommits and moves to the WB. This breaks the dependence links between the miss load and its dependents, requiring the full dependent thread to be replayed and renamed again to reestablish the dependence relation. This work uses virtual register names, which persist for the full lifetime of the instructions, to specify the dependences between instructions. This allows partial replay of only those dependents moved into the WB while the miss is outstanding, significantly reducing the number of replayed instructions and the total execution time.
- This work introduces an improved CFP policy that keeps miss-dependent instructions in the RS as long as they do not block the pipeline. However, when the instruction buffers become full with miss-dependent instructions, thus stalling the pipeline, CFP moves the miss-dependent instructions into the WB. Moving instructions into the WB on a resource need basis significantly reduces the amount of replay and costly rollbacks to the checkpoints resulting from branch mispredictions that depend on load misses.
- This work introduces a hardware predictor to predict miss-dependent branches likely to mispredict. We use this predictor as a branch confidence mechanism to reduce the runahead distance of CFP and the corresponding increase in checkpoint rollback risk. This prediction mechanism improves performance slightly but significantly reduces power resulting from checkpoint rollbacks on benchmarks that have many mispredicted branches that depend on load misses.

Fig. 1. S-CFP block diagram.

- The optimized CFP architecture in this work removes the ROB from the replay loop, which reduces the replay loop latency of miss-dependent instructions evicted into the WB and thus reduces execution time and dynamic power.
- Using a microarchitecture performance simulator and architectural-level power model, the article shows that the optimized CFP architecture improves execution time and energy consumption by 10% and 8%, respectively, over S-CFP architecture. The article also shows that as the instruction buffer sizes are scaled over a wide range, the Tuned-CFP architecture consistently provides better performance return on Energy Per Instruction (EPI) compared to the conventional ROB superscalar architecture, but that the nonoptimized previous S-CFP architecture does not.

The rest of this article is organized as follows. Section 2 presents an overview of the S-CFP architecture. Section 3 follows with a description of VRR and our Tuned-CFP architecture and its optimizations. We outline our simulation methodology in Section 4. Section 5 evaluates performance and energy of the new CFP design and compares with previous CFP architectures as well as conventional baseline cores of equivalent instruction buffers capacity. We discuss related work in Section 6 and conclude in Section 7.

2. CFP ARCHITECTURE

This section gives an overview of S-CFP [Jothi et al. 2011] and illustrates, with execution examples, some of its drawbacks. It also explains the new mechanisms that we use to tune S-CFP and overcome its drawbacks.

2.1. S-CFP Architecture Overview

The S-CFP performs register renaming using a ROB, as in Intel P6 architecture [Papworth 1996].

Figure 1 shows a block diagram of S-CFP microarchitecture [Jothi et al. 2011]. Unlike previous latency-tolerant out-of-order architectures, the S-CFP core executes cache miss-dependent and miss-independent instructions concurrently using two different hardware thread contexts. The S-CFP hardware is similar to Simultaneous Multithreading (SMT) architectures [Tullsen et al. 1995], except that in S-CFP, two simultaneous threads are formed of the miss-dependent and miss-independent instructions constructed dynamically from the same program, instead of being two different programs that run simultaneously in the same core. In order to support two hardware threads, S-CFP has two Register Alias Tables (RATs) for renaming the independent

Fig. 2. Execution sequence showing S-CFP moving a dependent into WB eagerly.

and the dependent thread instructions. S-CFP also has two Retirement Register File (RRF) contexts, one for retiring miss-independent instruction results and the other for retiring miss-dependent instruction results. The two threads share the ROB, load queue (LQ), store queue (SQ), RS, and data cache.

The independent hardware thread is the main execution thread. It is responsible for instruction fetch and decode of all instructions, branch prediction, memory dependence prediction, identifying miss-dependent instructions, and moving them into the WB. The dependent thread execution starts when the load miss data is brought into the cache, waking up the load instruction in the WB, and continues until the WB empties. At the end of dependent execution, when all of the instructions from the WB have retired (i.e., committed) without any mispredictions or exceptions, the independent and dependent execution results are merged together with a flash copy of the dependent and independent register contexts within the RRF. To maintain proper memory ordering of loads and stores from the independent and dependent threads execution, S-CFP uses LSQs, a Store Redo Log (SRL) [Gandhi et al. 2005], and a store-set memory dependence predictor [Chrysos and Emer 1998].

S-CFP renames instructions using the ROB [Smith and Sohi 1995]. When a load miss reaches the head of the ROB, it is pseudoretired and moved into the WB immediately. The load miss instruction releases all of its pipeline resources, including its ROB ID, before entering the WB, breaking its links with its readers that are still in the pipeline. For this reason, the entire dependence chain of the load miss and its dependents needs to be renamed and replayed from the WB until all miss-independent and miss-dependent instructions complete execution and their results are merged. This can last for a long distance forward in the program, causing excessive replays and rollbacks. We show next, in more detail, examples that illustrate the drawbacks of S-CFP targeted by our optimizations.

2.2. S-CFP Execution Examples

Figures 2(a) and 2(b) show snapshots of the ROB and WB states in S-CFP at different execution times. All shaded instructions correspond to either a load miss or a load miss dependent, both of which are potential-candidates to move into the WB.

In Figure 2(a), the WB has a load miss *X* at the head waiting for its wakeup. Instruction *A* misses the first-level cache and is marked as a potential candidate to be moved into the WB. When *A* reaches the head of the ROB, there are still free entries available in the ROB. Nevertheless, S-CFP eagerly pseudoretires and moves *A* into the WB. In order to execute when the miss data is fetched, *A* has to be replayed from the WB back into the pipeline to be renamed again and allocated resources for execution.

Fig. 3. Execution sequence showing a scenario leading to rollback in S-CFP, which is avoided in Tuned-CFP.

Figure 2(b) shows that the load miss hits the L2 data cache and A is woken up from the L1 data cache shortly after it enters the WB. However, it is stuck in the WB behind instruction X that has missed to DRAM. For a long time afterward, and until the miss data of load X is fetched from DRAM, many of the dependents of A will be poisoned and moved into the WB. Therefore, even though A has hit the L2 cache, it is replayed with its dependents from the WB as if it has encountered a miss in the L2 data cache and has needed to go all the way to DRAM for the data.

Figure 3(a)–(c) shows an execution sequence to illustrate a situation in S-CFP that leads to a rollback to the checkpoint. Similar to the earlier example, the WB has a load instruction X that has missed. Instruction A misses the first-level cache. In this example, F is a branch instruction dependent on A. A is moved into the WB from the head of the ROB, as shown in Figure 3(a). F also follows A into the WB, even though the wakeup for A arrives while F is still in the ROB/RS, as shown in Figure 3(b). Both A and F are replayed behind instruction X, as shown in Figure 3(c). On replay, branch F is found to be mispredicted, and branch misprediction recovery has to be performed by rolling back execution to the checkpoint, since by then, the sequential state in the register file has been corrupted by the out-of-order pseudoretirement of instructions during the cache miss processing.

Figure 4(a)–(d) shows another execution sequence to illustrate why S-CFP needs to replay a load and all of its dependents once the load enters the WB. In this example, A is a load miss, and B is dependent on A. The two instructions are separated by miss independents shown as dotted lines. The figures show only the miss dependents in the pipeline for clarity. The renamed physical source and destination register tags (*src1*, *src2* → *dest*) are shown alongside the instruction. Also shown below their ROB entries are the ROB IDs or physical destination registers of A and B. The WB, initially empty, is also shown in the figures.

In Figure 4(a), A reaches the head of the ROB. It pseudoretires and moves into the WB, releasing all of its pipeline resources, including its ROB ID #3, as shown in Figure 4(b). When the wakeup for A arrives and it replays, it is allocated a new entry at the tail of the ROB, as shown in Figure 4(c). Notice that A gets a new ROB ID #24 when it is reintroduced into the pipeline. Because of this new ID, even though B is still in the RS and the ROB while A is being replayed, A's data writeback cannot wake up B, because B still has the physical register destination ID #3 as its source operand. B reaches the ROB head, pseudoretires, and moves into the WB. When B is replayed and reintroduced into the pipeline, it goes through the rename stage, gets a new

Fig. 4. Comparison of full replay in S-CFP and partial replay in Tuned-CFP.

ROB ID #28, and receives the correct physical source register ID #24, reestablishing its link with A from the dependent RAT, as shown in Figure 4(d). Notice that an 'x' is shown for the other source IDs to indicate that the IDs of these sources are "don't care" for illustrating this example.

2.3. Tuned-CFP Execution Examples

As illustrated in the previous examples, S-CFP causes excessive replays and rollbacks, negatively impacting performance and energy consumption. Our Tuned-CFP architecture in this work handles these cases differently, as explained next.

Figures 2(c) and 2(d) shows the ROB and WB states in the Tuned-CFP architecture. Instruction A misses the first-level cache and is marked as poisoned. However, unlike in S-CFP, it does not release its RS entry until it becomes a blocking instruction, just in case the load hits the L2 cache providing the miss data to the CFP core shortly. A may reach the head of the ROB before the L2 data cache loads the data, but it will still be kept in the RS and ROB by stalling pseudoretirement as long as there are free entries in the ROB and other instruction buffers for the pipeline to continue execution of other instructions without blocking. If the miss data arrives before the pipeline blocks, A is woken up from the RS and ROB by clearing its poison bits, as shown in Figure 2(d). A and its dependents do not need to go through the replay loop at all in this example, saving significant time delay and energy.

Figures 3(d) and 3(e) shows how the rollback situation in S-CFP is avoided with the Tuned-CFP architecture. Similar to the previous example, instruction A stays in the ROB, even if it reaches the head, as long as it is not blocking execution. A gets its wakeup before it moves into the WB, as shown in Figure 3(e). Even though F is a miss-dependent and mispredicted branch, it executes before it pseudoretires. When it

Fig. 5. Block diagram of CFP architecture with VRR.

reaches the head of the ROB, the ROB flushes the pipeline to clear all of the wrong path instructions that have been fetched after the branch and signals to the fetch unit to restart fetch and execution from the corrected target. The costly S-CFP branch recovery from the checkpoint has been avoided.

Figures 4(e)–4(g) illustrates a partial replay in the Tuned-CFP architecture, representing the same scenario discussed earlier in Figure 4(a). In Tuned-CFP, virtual register IDs (VIDs) that are not associated with any physical locations are used for register renaming and in the RS wakeup and scheduling logic. The VIDs of instructions *A* and *B* are shown under their ROB entries in addition to the renamed source and destination VIDs. As before, when *A* reaches the head of the ROB, it pseudoretires and moves into the WB, as shown in Figure 4(e). However, unlike in S-CFP, *A* releases its RS but carries its VID #3 along with it into the WB, as shown in Figure 4(f). Later on, when it wakes up and replays, *A* still carries with it its original VID #3, still maintaining its link with its dependent instruction *B* intact. This allows the RS to schedule *B* without replaying and renaming it again, as shown in Figure 4(g).

Avoiding excessive replays and rollbacks not only saves significant execution time by miss dependents but also considerable power consumption.

3. CFP WITH VRR

In this section, we describe the core architecture of CFP with virtual registers renaming (Tuned-CFP), its key mechanisms, and its improvements over previous CFP architectures.

3.1. Tuned-CFP Architecture Overview

Figure 5 shows a block diagram of the Tuned-CFP core. Tuned-CFP microarchitecture uses Tomasulo’s algorithm and RS to perform data-driven, out-of-order execution [Tomasulo 1967].

Like other superscalar architectures, Tuned-CFP uses a ROB to commit instructions and update register and memory state in program order. However, it does not use the ROB for register renaming. Instead, it performs register renaming using VIDs generated by a special counter. These VIDs are not mapped to any fixed storage locations in the core, thus they can be large in number and allocated to instructions throughout their lifetime, including miss-dependent instructions evicted to the WBs. Since the VIDs are plentiful, Tuned-CFP does not run the risk of pipeline stalls resulting from miss-dependent instructions holding on to their renamed registers for a long time while waiting for the long-latency load miss. The VID counter is finite in size and cannot be allowed to overflow in order to prevent allocating the same VIDs

to multiple instructions in the pipeline. Tuned-CFP opportunistically resets the VID counter whenever it can—for example, when the pipeline is flushed to recover from a mispredicted branch. Otherwise, a pipeline stall and drain is forced to reset the counter when it overflows. With 10-bit VID counter, our simulations show that the impact on performance of forced pipeline stalls to reset the counter is negligible.

VRR gives Tuned-CFP a significant advantage over previous CFP architectures by allowing Tuned-CFP to replay only a part of the load miss dependence chain.

3.1.1. Miss Independents Execution. Like previous CFP cores, the Tuned-CFP core is capable of executing cache miss-dependent and miss-independent instructions concurrently in the pipeline, supported by two RRF contexts, one for retiring miss-independent instructions and the other for retiring miss-dependent instructions.

In Tuned-CFP, execution initially starts using an RRF context that we call the *independent RRF*. When an L1 data cache load miss occurs, a poison bit is set in the destination ROB entry of the load. Load-dependent instructions in the RS capture the poison bit from the common writeback data bus. They are then woken up, as if the load completed, and are scheduled by the RS control logic for pseudoexecution. Pseudoexecution of poisoned instructions does not actually use any execution units. However, pseudoexecution consumes RS dispatch ports and writeback bus cycles to propagate poison bits through instruction dependences and to identify all instructions in the RS that depend on the load miss data. After pseudoexecution, miss-dependent instructions stay in their RS until they wake up for real execution when the load miss data arrives, or until they are moved into the WB.

3.1.2. Replay Loop and Miss Dependents Execution. Figure 5 shows the reduced replay loop in Tuned-CFP consisting of two stages: the RS and the WB. The WB basically acts as a second-level storage for the RS. With VRR, entries can be freely evicted from the RS to the WB and then loaded back again to the RS to be scheduled for execution at a later time.

In Tuned-CFP, miss-dependent instructions are evicted from the RS to the WB only when their buffer resources are needed to unblock the execution pipeline. Therefore, when a miss is processed and its data fetched to the L1 data cache, the miss-dependent instructions may still be in the RS. In this case, when the data is written back, the miss data is captured by all of the RS that have instructions that depend on the miss. The captured writeback data sets the source operand “ready” state bits and clears the poison bits of the dependent RS entries, making the instructions in these entries ready for scheduling and dispatch to execution.

Evicting miss-dependent instructions to the WB on a resource need basis significantly reduces the number of replayed instructions, especially in the case of medium-latency load misses, which are those that miss the L1 data cache but hit the on-chip L2 cache.

In case of a load miss to DRAM, it is often the case that the long miss latency causes the instruction buffers to fill up and some or all of the in-flight miss-dependent instructions to evict to the WB. When the load miss is serviced, the miss load and its dependents are reinserted from the WB back to the RS where they are scheduled for execution.

Notice that even though replayed miss-dependent instructions do not need to be renamed again, they sometimes, as in Tomasulo’s algorithm, need to read source operands that have already been computed and retired from the ROB to the register file (RRF). State bits that track whether the last instructions to write logical registers have been retired are stored in a special storage structure. These state bits are checked during replay to determine if the operands are ready in the RRF and to read them and move them into the RS with the replayed instructions.

3.1.3. Tuned-CFP Reservation Stations. Tuned-CFP uses a centralized array of conventional data-capture RS [Papworth 1996]. Each RS entry is extended with a poison bit per source operand and L1-DCache-miss bit. The L1-DCache-miss bit is set to 1 if the entry contains a load instruction that has missed the L1 data cache. We say that an instruction is poisoned if one of its source poison bits or the L1-DCache-bit is set to 1. A source operand of an instruction is poisoned if and only if it is the destination of another poisoned instruction. In other words, the poison bits propagate the dependences from L1 data cache misses to later instructions in the program to identify instructions that may encounter long data cache miss delays. These instructions are candidates to move to the WB to avoid pipeline stalls that could occur if any of the RS, ROB, LQ, or SQ arrays becomes full.

The RS array is augmented with a free list and an order list. Tuned-CFP uses the free list to track the occupancy of the RS. The order list tracks the program order of the RS. The RS order list could be implemented as part of the ROB by adding the RS ID of each instruction to its allocated ROB entry. It also could be implemented as a special array separate from the ROB.

Four conditions are checked to determine if an instruction should be moved to the WB: (1) the instruction is at the head of the RS order list; (2) the instruction is poisoned; (3) one of the RS, ROB, LQ, or SQ arrays is full; and (4) every source operand of the instruction is either poisoned or ready. The last condition ensures that the miss-dependent instructions carry their nonpoisoned input values with them when they are replayed, since there is no guarantee that these values would not be overwritten in the register file by replay time. In addition, each RS has a state bit that indicates if the instruction in the entry has been replayed once before—that is, it has been moved earlier in time to the WB and then back to the RS array. Each RS also contains a load miss identifier in case it has a load instruction that misses the data cache. An implementation could use for this purpose the ID of the L1 data cache fill buffer used to handle the load miss.

3.1.4. Waiting Buffer. The WB is a wide single-ported SRAM array managed as a circular buffer using head and tail pointers. Miss-dependent RS entries at the head of the RS array moves to the tail of the WB when any of the instruction buffers fills up due to data cache misses. When a data cache miss is completed, Tuned-CFP replays the miss-dependent entries by loading them back from the head of the WB to the tail of the RS. Ideally, the width of the two buses connecting the RS and the WB would match the pipeline width. Narrower interconnect can also be used, trading some performance for simpler hardware.

A key to the efficiency of Tuned-CFP large window design is the fact that the WB has no CAM ports, connections to writeback buses for capturing data operands, or conventional ready/schedule logic. All of these functions are handled in the RS array after the data cache miss completes and the miss-dependent instructions are replayed. Therefore, the WB array can be designed using nontagged SRAM and made significantly larger than the RS array at much lower area and power cost than if the RS array is large enough to hold the full instruction window.

In order to wake up miss dependents from the WB and replay them, the L1 data cache fill buffer handling a load miss has to receive and save the WB ID of its load miss. When the miss completes, Tuned-CFP replays the load miss and its dependents in program order, as described earlier, from the head of the WB back into the RS allocate/write stage of the execution pipeline.

3.1.5. Register File and Results Integration. Tuned-CFP uses the ROB to handle branch mispredictions and exceptions incurred by miss-independent instructions. On the other

Fig. 6. RRF cell with checkpoint and result integration logic.

hand, it uses checkpoints to handle branch mispredictions or exceptions encountered by miss-dependent instructions.

Like the prior S-CFP architecture [Jothi et al. 2011], Tuned-CFP has a specialized register file for checkpointing register state at the load miss, for later use to handle miss-dependent branch mispredictions and exceptions. The register file also has special logic for integrating the results of independent and dependent instructions and to restore precise register state after all miss-dependent instructions execute.

Figure 6 shows a Tuned-CFP RRF cell with checkpoint flash copy support. Tuned-CFP uses a flash copy of the RRF for creating checkpoints. In one cycle, every independent RRF state bit (left-most latch) is shifted into a checkpoint latch within the register cell (center latch). The register file can be restored from the checkpoint in one cycle by asserting `RSTR_CLK`.

In addition to the checkpoint bit and the independent RRF context bit, Tuned-CFP register file cell contains one context bit for the dependent RRF state (right-most latch). The integration of the independent and dependent instruction results is done in one restore cycle. At the end of dependent execution, after all instructions in the WB have replayed and retired, the RRF has all the live-out registers, some of which are computed by the independent instructions and some by the dependent instructions. This is determined by the poison bits in the RRF. To integrate these results back into one context, a restore cycle is performed from the dependent context into the independent context. However, not all registers are copied. Figure 6 shows that only poisoned registers are copied by using the poison bits to enable the clock of the copy operation. A 2-to-1 multiplexer in the cell restores either the checkpoint bit or the dependent bit during an `RSTR_CLK` cycle.

3.1.6. Load and Store Execution in Tuned-CFP. To maintain proper memory ordering of loads and stores from the independent and dependent instructions execution, Tuned-CFP uses LSQs, an SRL [Gandhi et al. 2005], and a store-set memory dependence predictor [Chrysos and Emer 1998]. A detailed description of Tuned-CFP SRL and the speculative L1 data cache has been presented in previous works [Jothi et. al 2011; Sharafeddine et al. 2012].

3.1.7. Miss-Dependent Branch Predictor. A key reason for CFP performance degradation is dependent branches that go into the WB and are later found to be mispredicted. Factors contributing to this performance degradation include a large window of wrong

path instructions, re-execution of instructions between the load miss and the mispredicted branch, and delayed resolution of the miss-dependent branch while waiting its turn to come out of the WB.

To address this problem, we use an approach similar to pipeline gating in Manne et al. [1998], except that we apply it only to costly dependent branches. We identify branches that are likely to mispredict and take necessary action when they move into the WB. We have observed that in multiple benchmarks, there is a strong correlation between the dependent mispredicted branch and its PC value. We use a small hardware predictor of 32 entries that contains the addresses of previous dependent mispredicted branches to estimate branch confidence. The processor front end is stalled when a branch with low confidence is moved into the WB. The front end of the pipeline is unblocked only after the load miss data is delivered to the cache and the branch is resolved. Our results in Section 5.2.2 show that this mechanism reduces excessive replay and rollback execution.

4. EVALUATION METHODOLOGY

We built our Tuned-CFP architecture model on the SimpleScalar ARM ISA simulation infrastructure (www.simplescalar.com). CFP benefits applications that suffer high data cache miss rates. However, its effectiveness in handling data cache misses is limited by mispredicted branches that depend on data cache misses. For this reason, we used all 14 “C” benchmarks from SPEC 2000 and SPEC 2006 that we succeeded in compiling using the SimpleScalar cross compiler tool.

These benchmarks do not suffer much on average from data cache misses but have high branch misprediction rates that make them useful in exposing the performance limitations and glass jaws of CFP, thus making them appropriate for evaluating the effectiveness of our proposed optimizations. Our choice of ARM ISA is arbitrary and does not change the conclusions in the article, since data cache miss rates and branch mispredictions depend mainly on the application characteristics and not the ISA.

After skipping the initialization code and warming up the caches and predictors for 40 million instructions, we simulated 200 million instructions from each benchmark, consisting of four different samples manually selected from representative execution phases to display wide variation in the cache miss rate as well as the branch misprediction rate, both of which significantly impact CFP execution behavior. Results in Section 5 show the average performance of these selected samples for each benchmark.

For energy analysis, we have obtained measurements from SPICE circuit simulations of all baseline superscalar and CFP core functional blocks, using Cadence tools and 45nm process technology. We combined these with logic switching activity from our SimpleScalar simulator, creating an Architectural-Level Power Simulator (ALPS) [Brooks et al. 2000]. All energy results reported in this article account for total energy, which is the sum of dynamic and static (or leakage) energy. Our model assumes 30% of energy consumed per instruction to be due to leakage, which is typical of current high performance integrated circuits technology. The energy model accounts for the area of additional structures including 8K bytes of SRAM in the SRL and SDB, two thread contexts, and checkpoint and flash copy logic in the RRF. Our estimate for the area overhead of CFP adds up to less than 5% of the non-CFP core configuration shown in Table I. We used 6-transistor SRAM cell design with differential sense amplifiers for the SRL and WB circuits and a single-sided register file cell, augmented with checkpoint and results integration circuit. Unless specified otherwise, the average consumed EPI is reported for each evaluated core relative to the non-CFP baseline configuration that we describe next.

Table I shows the simulated machine configuration. Since our simulations are done with a single-core model, we use a two-level cache hierarchy with an L2 cache size that

Table I. Simulated Machine Configuration

Pipeline	Fetch to retire 15 stages, rename to retire 8 stages, 4-wide
Instruction buffers	80 ROB, 40 RS, 40 LQ, 30 SQ
L1 I-Cache	16KB, 8-way, 3 cycles, 64-byte line
L1 D-Cache	8KB, 2-way, 3 cycles, 64-byte line
L2 cache	Unified, 512KB, 16-way, 32 cycles, 64-byte line
DRAM latency	150 cycles L2 to data return with on-chip DRAM controller
Branch predictor	Combined bimodal and gshare, 4K Meta, 4K bimodal, 64K gshare, 4K BTB, 16-entry Return Address Stack
Hardware data prefetcher	Stream based (16 streams)

Fig. 7. Limit studies showing potential speedup from buffer full, Oracle miss-dependent branch predictor, and minimum replays, individually and combined.

is representative of the L2 capacity per core of current multicore processors. We selected optimum instruction buffers and L1 data cache sizes for maximum EPI efficiency, as described in Section 5.3.

5. EXPERIMENTAL RESULTS

5.1. Limit Studies to Quantify the Disadvantages of S-CFP

To quantify the disadvantages of S-CFP described in Section 2.2, we used ideal studies to eliminate each of these disadvantages and to compare the performance against a nonoptimized S-CFP architecture.

5.1.1. Buffer Full Condition. As we pointed out earlier in Section 2.2, S-CFP does not wait for the buffer full (*bf*) condition before moving a load miss or its dependent instructions into the WB. To measure the impact of this issue, we evaluate a model in which a load miss instruction is prevented from moving into the WB until it blocks the core execution pipeline. Figure 7 shows the performance improvement due to this optimization.

Note that benchmarks, such as gcc and perl, benefit the most from keeping dependent instructions in the ROB/RS until they block the execution pipeline. This is because these benchmarks have a significant number of L1 cache misses that actually hit the L2 and do not encounter the very long DRAM access latency. Therefore, leaving a load

miss in the RS for a few cycles longer significantly increases the probability that the load miss will receive a wakeup signal before it moves to the WB, therefore saving the entire miss-dependence slice from being replayed.

From Figure 7, the average contribution of *bf* optimization over all benchmarks is 2.7%. This speedup is not very high, mainly because the window of opportunity to save unnecessary replay is limited to the time between the load miss reaching the ROB head and the time the instruction buffer fills up. If the miss data does not come back within this time window, the entire dependence chain must be replayed.

5.1.2. Miss-Dependent Branch Mispredictions. Miss-dependent branches that are later found to be mispredicted cause performance degradation in CFP architectures because of the following reasons. First, execution needs to be rolled back to the checkpoint taken at the load miss to recover from the incorrect speculative updates made to the architectural state. Second, the miss-dependent branch itself may be resolved much later in time depending on when it replays from the in-order WB. Until a miss-dependent mispredicted branch is resolved, S-CFP continues to fetch instructions from the wrong path. This not only wastes precious pipeline resources, including cycles in the power-hungry fetch and decode stages, but also exerts additional pressure on the execution pipeline, thus increasing the number of instructions that are moved into the WB and subsequently replayed.

In order to quantify the impact of miss-dependent mispredicted branches on performance, we evaluate an S-CFP model with an Oracle predictor that stalls the processor front end perfectly on a miss-dependent mispredicted branch. This is the *dp* model in Figure 7. As can be seen in Figure 7 from the speedup of the *dp* model, the average impact of miss-dependent mispredicted branches on performance is $\sim 5\%$. This is a moderate performance loss compared to that caused by the large number of miss-dependent instructions replayed from the WB, as we present in the next section.

5.1.3. Unnecessary Replay Performance Overhead. In order to measure the negative impact of instruction replays on S-CFP performance, we evaluate an ideal model that eliminates unnecessary instruction replay. As described earlier in Section 2.2 and Figure 4, unnecessary replays are those caused by having to replay instructions that are still in the L1 RS when the miss load has been replayed in order to restore the dependence links. This model is called *ur* in Figure 7.

From Figure 7, one can see that 12% reduction in S-CFP performance comes from unnecessary replays. This shows that VRR is therefore the most important optimization to S-CFP.

5.1.4. Combining Limit Studies. Figure 7 also shows two models (*ur* + *bf* and *ur* + *bf* + *dp*) that combine individual limit studies. As can be seen from these two ideal models, the potential performance benefit from multiple optimizations is cumulative. In particular, the *ur* + *bf* + *dp* model, which combines the three optimizations, establishes the performance upper bound that can be achieved with a realistic Tuned-CFP machine exhibiting minimum possible replay and rollback. The importance of eliminating unnecessary replays with VRR to CFP is evident from these results.

In summary, on memory-intensive benchmarks, the average contribution from each optimization toward improving performance is 4% for *bf*, 9% for *dp*, 18% for *ur*, and 22% when combined. None of the optimizations require power-hungry structures. Moreover, they provide sizeable speedup, which justifies their implementation cost. Since these optimizations complement each other very well, combining them to get the best possible performance would be the recommended option. We next present simulated performance results for a realistic Tuned-CFP machine.

Fig. 8. Tuned-CFP percent speedup over S-CFP.

5.2. Comparing Realistic Tuned-CFP to S-CFP

This section compares the performance and EPI of Tuned-CFP to S-CFP. This section also isolates the contribution of each optimization scheme when applied by itself toward improving overall performance of Tuned-CFP.

5.2.1. Tuned-CFP Performance. Figure 8 shows the speedup of Tuned-CFP over S-CFP when targeting data cache misses at all levels. Tuned-CFP, with its VRR, short replay loop, and reduced replay/rollback outperforms S-CFP by an average of 10% when the CFP algorithm is applied to L1 data cache misses. The maximum improvement occurs on gcc (39% speedup), which displays high rate of L1 cache misses as well as high rate of branch mispredictions. The reduced replay loop of Tuned-CFP is very favorable to benchmarks like gcc and perl, since reducing the amount of replay also significantly reduces the number of costly miss-dependent branch mispredictions. The variation in the improvement between benchmarks is mainly due to the variation in the cache miss rates and branch misprediction rates of our simulation samples. The long replay loop of S-CFP is a glass jaw that is exposed on benchmarks like gcc that display high cache misses and branch misprediction rates. A carefully designed replay loop is necessary when designing CFP for general purpose processors that target many applications with widely different execution characteristics.

Notice that the results in this section are reported with a Tuned-CFP core having all of the optimizations discussed in Section 3. With the intention to highlight the benefits from the most architecturally significant optimization (i.e., VRR), Tuned-CFP performance is compared to an S-CFP that has all of the optimizations except VRR—that is, buffer full optimization for load misses and stalling the front end based on the miss-dependent branch predictor. For this reason, Tuned-CFP speedup over S-CFP in Figure 8 is not as high as the speedup of the *ur* + *bf* model over unoptimized S-CFP shown in Figure 7.

5.2.2. Tuned-CFP Optimizations. There are three optimizations featured in Tuned-CFP: (1) partial replay of the load miss dependence chain, (2) moving miss-dependent instructions into the WB only when a resource or buffer becomes full, and (3) dependent branch confidence predictor. We call these optimizations PR, BF, and DP, respectively.

The contribution of the PR and BF optimizations toward the performance of Tuned-CFP is exactly the same as the benefit from the ideal models *ur* and *bf* shown in

Fig. 9. Percent speedup contribution from miss-dependent branch predictor.

Figure 7. Figure 9 shows the percent speedup contributed by the simple history-based miss-dependent branch predictor (DP) discussed in Section 3.1.7. The speedup coming from the Oracle miss-dependent branch predictor (DP_Orc) is also shown for comparison. Notice that the $PR + BF + DP$ model and the Oracle $PR + BF + DP_Orc$ model contribute average speedups of 3.7% and 4.2%, respectively, showing that despite of the small size of our 32-entry dependent branch confidence predictor, it achieves a speedup within 0.5% of the perfect upper bound performance of the Oracle model.

We observed that some benchmarks, like gcc and twlf, have a small number of static branches that contribute to a large percentage of dependent branch mispredictions, making our small 32-entry hardware predictor very effective. Other benchmarks, such as gobk and gzip, display more complex control flow patterns with larger set of static branches contributing to mispredictions. For these benchmarks, our dependent branch predictor is less effective. Using global correlated branch prediction methods might help these benchmarks even further than our simple confidence prediction scheme.

5.2.3. EPI Comparison of S-CFP and Tuned-CFP. Figure 10 shows the percentage increase in EPI of S-CFP and Tuned-CFP over the non-CFP baseline for each of our simulated benchmarks. Table II shows replayed instructions, rollback instructions, and wrong path instructions as a percentage of total instructions for both S-CFP and Tuned-CFP.

The effectiveness of Tuned-CFP optimizations in reducing the excessive replay, rollback, and speculative execution is evident and accounts for the performance improvement (Figure 8) and energy advantage of Tuned-CFP over S-CFP. Table III shows average EPI across all simulated benchmarks of various functional blocks as well as the total non-CFP baseline, S-CFP, and Tuned-CFP cores. All numbers in the table are shown relative to the total EPI of the non-CFP baseline core. Tuned-CFP shows ~8% less EPI compared to S-CFP due to reduction in replayed instructions execution, rollback, and wrong path instructions. Finally, notice that Tuned-CFP and the non-CFP baseline consume about the same EPI, with Tuned-CFP measuring on average just about 2% additional EPI over the non-CFP ROB baseline core.

5.3. Analysis of Energy Characteristics of Non-CFP Superscalar Cores

This section presents an analysis of energy efficiency of a conventional 4-wide superscalar core that does not implement CFP, as the core instruction buffers and L1 cache

Fig. 10. EPI percent increase of S-CFP and Tuned-CFP over ROB baseline.

Table II. Tuned-CFP and S-CFP Replay, Rollback, and Wrong Path Instructions in Percentage of Total Instructions

Bmk	S-CFP % Replay	Tuned-CFP % Replay	S-CFP % Rollback	Tuned- CFP % Rollback	S-CFP % Wrong Path	Tuned-CFP % Wrong Path
eqke	7.2	3.2	1.5	0.9	20.4	18.2
gcc	30.6	8.5	14.1	2.7	53.9	21.8
gobk	17.6	5.4	14.4	2.5	61.6	38.0
gzip	9.0	3.5	8.4	1.8	44.4	28.8
h264	10.0	2.5	2.2	0.3	17.7	11.3
hmm	7.8	2.4	0.1	0.1	0.7	0.5
lbm	1.7	0.0	0.0	0.0	0.1	0.1
libq	0.0	0.0	0.0	0.0	0.0	0.0
mcf	0.2	0.1	0.0	0.0	3.3	3.3
milc	0.0	0.0	0.0	0.0	4.0	4.0
perl	20.6	3.7	3.2	0.1	12.6	4.4
sjng	12.9	4.3	6.5	1.1	55.4	37.9
sphx	4.0	0.9	1.4	0.0	11.9	8.0
twlf	20.0	8.6	13.9	6.5	54.6	36.9

Table III. S-CFP and Tuned-CFP Average Energy in Percentage Relative to ROB Baseline Total

	L1 I-Cache	Decode	RAT	RS	EUs	ROB/RRF	LSQ	L1 D-Cache	SRL/WB	Total
Base	8.6	18.9	15.5	10.8	13.8	12.9	10.8	8.6	0.00	100.0
S-CFP	9.2	20.2	16.5	12.3	14.7	15.4	11.5	9.2	1.9	110.9
Tuned-CFP	8.7	19.2	15.7	11.2	14.0	13.2	10.9	8.7	1.1	102.8

sizes increase. We use an intuitive definition of energy efficiency. First, we define return on energy (ROE) from an added hardware feature to be the percent increase in performance divided by the percent increase in EPI resulting from the added feature. Using ROE definition, we say that core *A* is more energy efficient than core *B* if core *A* ROE is larger than 1. In other words, core *A* is more efficient than core *B* if *A* improves performance by a larger percentage than the percentage of additional energy it consumes relative to core *B*.

Fig. 11. Percent speedup of non-CFP 4-wide superscalar cores of different instruction buffer configurations and ideal data cache over minimal configuration core.

5.3.1. Energy Characteristics of 4-Wide Non-CFP Superscalar Core with Ideal Data Cache. The sizes of instruction buffers and the L1 data cache significantly impact performance and EPI of superscalar cores. In this section, we investigate the performance and ROE of non-CFP superscalar cores as the instruction buffers sizes increase. We simulate an ideal data cache to isolate the contribution coming from instruction buffers alone. We vary the sizes of instruction buffers, namely ROB, RS, LQ, and SQ from a 32_16_16_12 configuration to a 192_96_96_72 configuration (ROB_RS_LQ_SQ). All other parameters like machine width, pipeline length, branch predictors, and instruction cache size remain unchanged and are shown in Table I.

Figure 11 shows the percent speedup of non-CFP superscalar core with ideal L1 data cache for various buffer sizes. All speedups are reported as percentage increase in IPC relative to the minimum 32_16_16_12 configuration. Only a representative set of benchmarks are shown in the figure for clarity. It is clear from Figure 11 that for all benchmarks, speedup initially increases linearly and then saturates as the buffers sizes increase.

Figure 12 shows a plot of the ROE for various configurations relative to the minimum 32_16_16_12 machine and across all benchmarks. We identify two configurations of interest in Figure 12. The maximum ROE occurs at the 48_24_24_18 configuration for all benchmarks. This *peak return on energy* configuration is the most energy-efficient configuration by our definition. A second configuration of interest is the configuration for which most benchmarks have a value of ROE around 1. From Figure 12, this is configuration 128_64_64_48. We call this configuration the *point of diminishing return on energy*. If the instruction buffers are increased beyond this point, the resulting speedup will be smaller than the accompanying increase in energy. We next use this configuration to determine an optimal L1 data cache configuration.

5.3.2. Selecting the L1 Data Cache Size. The L1 data cache configuration is critical toward deciding the core power consumption. The data cache is typically optimized for lowest possible load hit latency with high associativity and aggressive circuit design that uses concurrent read of the tags and data from all ways in the indexed set. To find the optimal L1 cache configuration experimentally, we use the point of diminishing return buffers configuration and vary the L1 data cache size from 4KB to 32KB and associativity from 1 to 8. Our simulations show that the peak ROE occurs at 2-way, 8KB

Fig. 12. ROE of non-CFP core with ideal data cache for different buffer configurations.

Fig. 13. ROE of non-CFP core with 8KB L1 D-cache for different buffer configurations.

L1 D-cache, whereas the point of diminishing ROE occurs at 4-way 16KB L1 D-cache size. We do not show the L1 data cache ROE plots due to shortage of space.

5.3.3. Varying Instruction Buffer Sizes with Optimal L1 Data Cache. To see how the ROE curves of the non-CFP core behave when simulated with a practical cache configuration, we re-do the experiment shown in Figure 12 with the peak ROE 8-KB L1 data cache instead of ideal cache. Figure 13 shows the results of this experiment, in that whereas the peak ROE point stays at the 48_24_24_18 configuration, the point of diminishing return moves backward from 128_64_64_48 for an ideal data cache to the 80_40_40_30 configuration. Notice that by the time the 96_48_48_36 configuration is reached, the ROE is well below unity for almost all benchmarks. This indicates that increasing the buffer sizes from 80_40_40_30 to 128_64_64_48 configuration benefits more from the increased ILP due to a larger instruction window than from increased tolerance to

Fig. 14. Percent speedup of Tuned-CFP over non-CFP core for EFF, DIM, and HP models.

L1 data cache misses. This observation validates our hypothesis that a better design strategy for energy-efficient cores is to size the instruction buffers appropriately for code that hits the L1 data cache and use CFP to handle data cache misses.

5.4. Comparing Tuned-CFP to Non-CFP Core Architectures

The experiments presented point to three machine configurations suitable for different design targets:

- (1) A *peak return on energy* machine with 48_24_24_18 buffers and 2-way 8KB L1 data cache. This is the most energy-efficient configuration, so we call it EFF.
- (2) A unity gain ROE machine that represents the *point of diminishing return* with 80_40_40_30 buffers and 4-way 16KB L1 data cache (DIM).
- (3) A large machine that compromises on ROE for *high performance* (HP). This configuration uses 192_96_96_72 buffers and 4-way 16KB L1 data cache. This is what a designer might choose for best single-thread performance.

Figure 14 shows the speedup of Tuned-CFP over a similar-sized conventional non-CFP core for the EFF, DIM, and HP machine configurations when CFP is applied to L1 data cache misses. Table IV shows various relevant execution statistics of Tuned-CFP. The first observation to note is that Tuned-CFP benefits performance mostly on benchmarks that frequently miss the data cache, as to be expected. Second, Tuned-CFP, with its latency tolerance to first-level cache misses, outperforms the non-CFP baseline by an average of 3% to 4% on all of the configurations. Even though the average speedup of all benchmarks over the non-CFP baseline is modest, Tuned-CFP shows considerable performance improvement on benchmarks that frequently miss the data cache—for example, gcc and eqke. Earlier CFP work has similarly shown modest performance improvement on applications like Spec 2006 that do not frequently miss the cache, but significant performance benefit on applications with high cache miss rates, such as server and workstations applications [Srinivasan et al. 2004].

Figure 15 shows the total core EPI increase of Tuned-CFP over the non-CFP baseline for all three machine configurations of interest. We observe that benchmarks that miss the cache incur an additional increase in EPI that depend on the amount of instructions replayed from the WB.

Table IV. Various Tuned-CFP Execution Statistics

Benchmark	L1 Cache Load Misses per 1,000 Instructions	L2 Cache Load Misses per 1,000 Instructions	% Speedup over Baseline	% Replayed Instructions	Dependent Branch Mispredictions per 1,000 Instructions
eqke	10.9	0.9	9.0	3.2	0.11
gcc	25.1	0.5	12.4	8.5	0.81
gobk	24.4	0.4	2.9	5.4	0.80
gzip	15.5	0.0	1.4	3.5	1.07
h264	11.1	0.0	1.5	2.5	0.22
hmm	7.7	0.1	7.1	2.4	0.01
lbm	2.7	0.0	0.0	0.0	0.00
libq	0.0	0.0	0.0	0.0	0.00
mcf	0.1	0.0	0.0	0.1	0.00
milc	0.0	0.00	0.3	0.0	0.00
perl	14.4	0.1	5.6	3.7	0.04
sjng	21.1	0.0	0.0	4.3	0.67
sphx	3.6	0.0	2.5	0.9	0.04
twlf	25.8	3.1	4.8	8.6	1.03

Fig. 15. Tuned-CFP percent increase in EPI compared to non-CFP EFF, DIM, and HP models.

Compare the increase in EPI to the speedup observed for the same benchmarks. Some benchmarks like gcc, eqke, and hmm miss the cache but benefit from CFP speculative execution while the cache miss data is being fetched. This is supported by the large speedup observed in these benchmarks at the cost of a small increase in power. Some benchmarks like twlf and sjng also miss the cache but encounter frequent miss-dependent branch mispredictions that offset the performance gain while burning power in the process. Finally, benchmark traces like libq and milc that do not show any benefit from CFP do not consume additional power. This is because the CFP structures (i.e., the WB and SRL) observe major switching activity only in CFP execution mode.

We summarize this section by saying that it is evident from Figures 14 and 15 that on many benchmarks the speedup is greater than the EPI increase, indicating that Tuned-CFP architecture gives a good performance ROE invested.

Fig. 16. ROE of non-CFP baseline, Tuned-CFP, and S-CFP cores.

We conclude Section 5 by comparing the ROE or energy efficiency of Tuned-CFP and S-CFP to non-CFP superscalar cores for different buffer configurations.

5.5. ROE Comparison of Tuned-CFP, S-CFP, and Non-CFP Cores

Previous works have argued for CFP as energy efficient, scalable, large instruction window architecture. However, there has been no work that quantifies the energy efficiency of CFP and compares it with that of conventional non-CFP superscalar architecture.

In this section, we compare the ROE of Tuned-CFP to that of S-CFP and non-CFP core by sweeping across a range of buffer configurations starting from the best-efficiency EFF configuration to the best-performance HP configuration. The baseline relative to which the ROE is computed is the 32_16_16_12 non-CFP configuration.

Figure 16 shows the average ROE across all of the simulated benchmarks for Tuned-CFP, S-CFP, and non-CFP cores. Tuned-CFP shows better ROE than non-CFP and S-CFP machines on all configurations. This clearly demonstrates the importance of VRR and other optimizations featured in the Tuned-CFP architecture.

Notice that the gap between Tuned-CFP ROE and non-CFP ROE is widest for maximum energy efficiency configuration (EFF) and tends to reduce with increasing buffer sizes as the non-CFP core suffers fewer stalls. With small buffer configurations like 48_24_24_18 and 64_32_32_24, the non-CFP core does not have enough resources to hide even short-latency misses that hit the on-chip L2 cache. Hence, the majority of cache misses end up stalling the non-CFP core. In this case, Tuned-CFP benefits more over the non-CFP core on many benchmarks, particularly when the speculative execution does not have to be discarded. When the buffer sizes are increased moderately (e.g., 80_40_40_30 configuration), Tuned-CFP still maintains a healthy ROE gap between itself and the non-CFP core.

On the other hand, when we move over to a large machine like 192_96_96_72, the gap between non-CFP and Tuned-CFP reduces because there are enough instruction buffers to handle all short-latency misses that hit the on-chip cache. The slight improvement in Tuned-CFP ROE comes because of the inevitable misses that go all the way to DRAM. In this case, even a large machine with 192_96_96_72 configuration is not able to keep the pipeline units busy for the entire time while the miss remains outstanding. It is important to notice the very poor ROE of a large machine for both non-CFP and Tuned-CFP, which is at a disappointing 0.4 value. This shows that

Table V. Tuned-CFP Percent Improvement in ROE over Non-CFP

Configuration	Non-CFP Core ROE	Tuned-CFP Core ROE	Percent Improvement in ROE
48_24_24_18	1.63	1.80	10.4%
64_32_32_24	1.18	1.32	11.8%
80_40_40_30	0.97	1.05	8.2%
96_48_48_36	0.80	0.87	8.7%
128_64_64_48	0.60	0.64	6.6%
192_96_96_72	0.41	0.43	4.8%

Note: ROE is computed relative to the 32_16_16_12 non-CFP configuration.

increasing the instruction buffer sizes to this extent hurts EPI considerably, proving the widely established opinion that traditional methods of increasing buffer sizes to get performance are already on a downward ROE slide.

Figure 16 also shows that the ROE of S-CFP is worse compared to both Tuned-CFP and non-CFP cores. The excessive replay and rollbacks in S-CFP not only bring down its overall performance but also dissipates excessive EPI in the process.

Finally, Table V shows the percent improvement in ROE of Tuned-CFP over the non-CFP core. Tuned-CFP shows up to 11% improvement in ROE for small-sized cores. The ROE gap starts decreasing for medium-sized cores and reaches a minimum for the largest buffer configuration. It is interesting to note that even on the large machine configuration, Tuned-CFP still manages to improve the ROE by 4.8 percentage points.

5.6. Tuned-CFP Performance with Data Prefetch

A core architect might ask whether CFP would still be useful to performance on cores that implement data prefetching hardware, and whether the performance of CFP justifies its hardware and increased energy cost.

The answer to the first question is not complicated. CFP should indeed benefit cores with data prefetch hardware. This is because CFP benefits any cache misses whenever they occur, whereas data prefetch, being a predictive mechanism, benefits only the cache misses that are predictable. Any cache misses that are not anticipated by the data prefetch hardware will be handled more effectively by a CFP core.

The question whether the hardware and energy overhead due to CFP can be justified on cores with data prefetch hardware is a more complex question to answer and requires empirical evaluation. In this work, we have evaluated CFP with an aggressive stream-based data prefetcher with 16 stream buffers. Our empirical results show negligible speedup (<0.01%) from data prefetch on our benchmarks suite, mainly because the memory access patterns of our benchmark traces do not exhibit frequent streams that can be exploited with the stream buffers hardware. We can therefore conclude from our results that CFP has clear performance and energy efficiency benefits, at least on our benchmark traces, with or without data prefetch streaming buffers hardware. Previous work [Srinivasan et al. 2004] that was done on an industrial core architecture model with data prefetch hardware using an extensive set of application traces, including SPEC CPU benchmarks and other server, workstation, and productivity traces, reported significant performance benefit from CFP beyond the data prefetch hardware. Our current study reconfirms the previous results on a smaller set of benchmarks.

6. RELATED WORK

Proposals for latency-tolerant out-of-order cores include the Waiting Instruction Buffer (WIB) [Lebeck et al. 2002], Virtual ROB [Cristal et al. 2002], Cherry [Martinez et al. 2002], Checkpoint Processing and Recovery [Akkary et al. 2003], CFPs [Srinivasan et al. 2004], and Out-of-Order Commit Processors [Cristal et al. 2004a]. None of

these, however, deal with L1 data cache misses or execute miss-dependent and miss-independent instructions simultaneously.

In order to support a large instruction window, WIB [Lebeck et al. 2002] physically buffers the entire window with a multilevel register file and large instruction buffers while releasing only the issue queue entries belonging to long-latency instructions. In Tuned-CFP, the entire instruction window is virtual, with only the miss-dependents, which are far fewer in number, occupying physical locations. In WIB, each instruction dependent on a load miss sets a bit vector dedicated for each outstanding miss, which allows miss-dependents to be reissued post wakeup, without needing the complex broadcast logic of the issue queue. Their WB is organized as a multibanked structure that allows miss-dependents to be reissued in any order as and when the wakeup arrives. Physically buffering the entire instruction window also allows instructions dependent on multiple load misses or miss-dependent misses to be moved in and out of the issue queue multiple times, although at the cost of excessive re-execution energy. In comparison, Tuned-CFP uses an energy-efficient single-ported WB that reissues only when the load miss at the head wakes up. Although this idea is simple and energy efficient, it also does not compromise on performance, because miss-independents continue to be processed by the core while the load at the WB head is waiting for the wakeup to arrive.

Runahead execution increases memory-level parallelism on in-order cores [Dundas and Mudge 1997], and on out-of-order cores [Mutlu et al. 2003] without having to build large ROB. In runahead execution, the processor state is checkpointed at a load miss to DRAM. Execution continues speculatively past the miss for data prefetch benefits. When the miss data return, runahead execution terminates, the execution pipeline is flushed, and execution rolls back to the checkpoint. Except for the prefetch benefit, all work performed during runahead is discarded. In Mutlu et al. [2005a], the runahead overhead is reduced with optimizations targeting short, overlapping, and useless runahead periods; yet, the runahead execution results are still discarded. In comparison to these proposals, Tuned-CFP executes ahead of L1 data cache misses and does not waste energy by discarding large number of instructions. In other works, Mutlu et al. [2005b] and Wolff and Barnes [2011] evaluated the benefit of reusing the execution results from runahead execution.

Flea-flicker [Barnes et al. 2003; Barnes et al. 2005] executes a program on two in-order back-end pipelines coupled by a queue. An advance pipeline executes independent instructions without stalling on long-latency cache misses while deferring dependent instructions. A backup pipeline executes the instructions deferred in the advance pipeline and merges them with results stored in the queue from the advance pipeline. Flea-flicker and Tuned-CFP differ in their execution and result integration methods and their instruction deferral queues. Flea-flicker executes instructions in an in-order pipeline, saves advanced instructions, and results in its queue and merges results sequentially during backup pipeline execution.

CFP on in-order cores was first proposed in Nekkalapu et al. [2008]. This approach is suitable for highly energy constrained computing devices but less suitable for the performance needs of conventional single-thread applications targeted by the Tuned-CFP multicore architecture.

iCFP [Hilton et al. 2009] tolerates cache misses at all levels in the cache hierarchy but uses an in-order pipeline, which is less suitable for the performance needs of conventional single-thread applications. BOLT [Hilton et al. 2010] utilizes additional map tables in SMT architecture to re-rename L2 miss-dependent slice, combined with a program order slice and a unified physical register file that supports aggressive register reclamation. BOLT reuse of SMT hardware is for improving energy efficiency

but does not extend the use of SMT to simultaneous execution of the dependent and independent slices. Neither does BOLT use VRR to improve energy efficiency.

Sun Microsystems Rock is a single-die multicore processor for high-throughput computing. Rock uses Simultaneous Speculative Threading [Chaudhry et al. 2009] to defer dependent instructions into a buffer and executes the deferred instructions from the checkpoint after the miss data returns. The deferred instructions execution uses a simultaneous hardware thread and merges the results into the scout thread future file. Rock uses an in-order pipeline, whereas Tuned-CFP core is out of order and thus provides better performance than Rock on single-thread applications.

Gonzalez et al. [1998] proposed using virtual registers to shorten the lifetime of physical registers. Kilo instruction processors [Cristal et al. 2004b] also used virtual renaming and ephemeral registers to do late allocation of physical registers. In contrast to virtual physical registers and ephemeral registers, VRR [Sharafeddine et al. 2013] and Tuned-CFP [Jothi and Akkary 2013a, 2013b] do not require physical registers for any allocation of execution results, and they accomplish renaming with virtual IDs and RS.

7. CONCLUSIONS

This article presents a tuned CFP architecture that uses VRR and optimized replay policies to improve performance and reduce replay loop circuit activity and checkpoint rollback execution compared to previous CFP designs. It achieves this by reducing the replay latency associated with the processing of miss-dependent instructions after the miss data arrives into the L1 data cache and wakes up the miss load. The architecture achieves this reduction in the miss-dependent instructions execution latency by (1) keeping these instructions as long as possible in the RS and moving them to the WB only when instruction buffer resources are needed and (2) using VRR that allows partial replay of the miss-dependent chain of instructions, requiring only the subset of these instructions that have already moved to the WB to be replayed. Our Tuned-CFP architecture improves performance and power consumption over previous CFP architectures by $\sim 10\%$ and $\sim 8\%$, respectively.

REFERENCES

- Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. 2000. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*. ACM, 248–259.
- Haitham Akkary, Ravi Rajwar, and Srikanth T. Srinivasan. 2003. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proceedings of the 36th Annual ACM/IEEE Symposium on Microarchitecture*. ACM/IEEE, 423–434.
- Haitham Akkary, Ravi Rajwar, and Srikanth T. Srinivasan. 2004. An analysis of a resource efficient checkpoint architecture. In *ACM Transactions on Architecture and Code Optimization* 1, 4, 418–444.
- Ronald D. Barnes, Erik M. Nystrom, John W. Sios, Sanjay J. Patel, Nacho Navarro, and Wen-mei W. Hwu. 2003. Beating in-order stalls with “Flea-flicker” two-pass pipelining. In *Proceedings of the 36th Annual ACM/IEEE Symposium on Microarchitecture*. IEEE, 387–398.
- Ronald D. Barnes, Shane Ryoo, and Wen-mei W. Hwu. 2005. Flea-flicker multi-pass pipelining: An alternative to the high power out-of-order offense. In *Proceedings of the 38th Annual ACM/IEEE Symposium on Microarchitecture*. IEEE, 319–330.
- David Brooks, Vivek Tiwari, and Margaret Martonosi. 2000. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*. ACM, 83–94.
- Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Haakan Zeffer, and Marc Tremblay. 2009. Simultaneous speculative threading: A novel pipeline architecture implemented in Sun’s Rock processor. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*. ACM, 484–495.

- George Z. Chrysos and Joel Emer. 1998. Memory dependence prediction using store sets. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*. ACM, 142–153.
- Adrian Cristal, Daniel Ortega, Josep Llosa, and Mateo Valero. 2004a. Out-of-order commit processors. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*. IEEE, 48–59.
- Adrian Cristal, Oliverio J. Santana, Mateo Valero, and Jose F. Martinez. 2004b. Toward kilo-instruction processors. In *ACM Transactions on Architecture and Code Optimization* 1, 4, 389–417.
- Adrian Cristal, Mateo Valero, Josep Llosa, and Antonio Gonzalez. 2002. Large virtual ROB by processor checkpointing, Tech. Report, UPC-DAC-2002–39, Department of Computer Science, Barcelona, Spain.
- James Dundas and Trevor Mudge. 1997. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the International Conference on Supercomputing*. 68–75.
- Amit Gandhi, Haitham Akkary, Ravi Rajwar, S. T. Srinivasan, and K. Lai. 2005. Scalable load and store processing in latency tolerant processors. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*. ACM, 446–457.
- Antonio Gonzalez, Jose Gonzalez, and Mateo Valero. 1998. Virtual-physical registers. In *Proceedings of the 4th International Symposium on High Performance Computer Architecture*. IEEE, 175–184.
- Andrew Hilton and Amir Roth. 2010. BOLT: Energy-efficient out-of-order latency-tolerant execution. In *Proceedings of the 16th International Symposium on High Performance Computer Architecture*. IEEE, 1–12.
- Andrew Hilton, Santosh Nagarakatte, and Amir Roth. 2009. Tolerating all-level cache misses in in-order processors. In *Proceedings of the 15th International Symposium on High Performance Computer Architecture*. IEEE, 431–442.
- Wen-mei W. Hwu and Yale N. Patt. 1987. Checkpoint repair for out-of-order execution machines. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*. ACM, 18–26.
- Komal Jothi and Haitham Akkary. 2013a. Tuning the continual pipeline architecture. In *Proceedings of International Conference on Supercomputing*. ACM, 243–252.
- Komal Jothi and Haitham Akkary. 2013b. Virtual register renaming: Energy efficient substrate for continual flow pipelines. In *Proceedings of the 23rd Great Lakes Symposium on VLSI*. ACM, 43–48.
- Komal Jothi, Haitham Akkary, and Mageda Sharafeddine. 2011. Simultaneous continual flow pipeline architecture. In *Proceedings of 29th IEEE International Conference on Computer Design*. IEEE, 127–134.
- Alvin R. Lebeck, Jinson Koppanalil, Tong Li, Jaidev Patwardhan, and Eric Rotenberg. 2002. A large, fast instruction window for tolerating cache misses. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*. ACM, 59–70.
- Srilatha Manne, Artur Klauser, and Dirk Grunwald. 1998. Pipeline gating: Speculation control for energy reduction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*. ACM, 132–141.
- Jose F. Martinez, Jose Renau, Michael C. Huang, Milos Prvulovic, and Josep Torrellas. 2002. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Microarchitecture*. ACM/ IEEE, 3–14.
- Onur Mutlu, Hyesoon Kim, and Yale N. Patt. 2005a. Techniques for efficient processing in runahead execution engines. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*. ACM, 370–381.
- Onur Mutlu, Hyesoon Kim, Jared Stark, and Yale N. Patt. 2005b. On reusing the results of pre-executed instructions in a runahead execution processor. In *Computer Architecture Letters* 4, 1, 2–2.
- Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. 2003. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture*. IEEE, 129–140.
- Satyanarayana Nekkhalapu, Haitham Akkary, Komal Jothi, Renjith Retnamma, and Xiaoyu Song. 2008. A simple latency tolerant processor. In *Proceedings of the 26th IEEE International Conference on Computer Design*. IEEE, 384–389.
- David B. Papworth. 1996. Tuning the Pentium Pro microarchitecture. In *IEEE Micro* 16, 2, 8–15.
- Mageda Sharafeddine, Haitham Akkary, and Doug Carmean. 2013. Virtual register renaming. In *Proceedings of the 26th International Conference on Architecture of Computing Systems*. 86–97.
- Mageda Sharafeddine, Komal Jothi, and Haitham Akkary. 2012. Disjoint out-of-order execution processor. In *ACM Transactions on Computer Architecture and Code Optimization* 9, 3, Article 19.
- James E. Smith and Gurinder S. Sohi. 1995. The microarchitecture of superscalar processors. In *Proceedings of the IEEE* 83, 12, 1609–1624.

- Srikanth T. Srinivasan, Ravi Rajwar, Haitham Akkary, Amit Gandhi, and Mike Upton. 2004. Continual flow pipelines. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 107–119.
- Robert M. Tomasulo. 1967. An efficient algorithm for exploiting multiple arithmetic Units. *IBM Journal of Research and Development* 11, 25–33.
- Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. 1995. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. ACM, 392–403.
- Sonya R. Wolff and Ronald D. Barnes. 2011. Reexamining instruction reuse in preexecution approaches. In *Proceedings of the 9th Annual Workshop on Duplicating, Deconstructing and Debunking*.

Received June 2013; revised October 2013; accepted November 2013