Continual Flow Pipelines

Srikanth T. Srinivasan Ravi Rajwar Haitham Akkary Amit Gandhi Mike Upton
Microarchitecture Research Labs
Intel Corporation
{srikanth.t.srinivasan, ravi.rajwar, haitham.h.akkary, amitx.v.gandhi, mike.upton}@intel.com

ABSTRACT

Increased integration in the form of multiple processor cores on a single die, relatively constant die sizes, shrinking power envelopes, and emerging applications create a new challenge for processor architects. How to build a processor that provides high single-thread performance and enables multiple of these to be placed on the same die for high throughput while dynamically adapting for future applications? Conventional approaches for high single-thread performance rely on large and complex cores to sustain a large instruction window for memory tolerance, making them unsuitable for multi-core chips.

We present *Continual Flow Pipelines* (CFP) as a new non-blocking processor pipeline architecture that achieves the performance of a large instruction window without requiring cycle-critical structures such as the scheduler and register file to be large. We show that to achieve benefits of a large instruction window, inefficiencies in management of both the scheduler and register file must be addressed, and we propose a unified solution.

The non-blocking property of CFP keeps key processor structures affecting cycle time and power (scheduler, register file), and die size (second level cache) small. The memory latency-tolerant CFP core allows multiple cores on a single die while outperforming current processor cores for single-thread applications.

Categories and Subject Descriptors

C.1 [**Processor Architectures**]

General Terms

Algorithms, Performance, Design.

Keywords

Non-blocking, Instruction Window, Latency Tolerance, CFP.

1. INTRODUCTION

In keeping with the natural trend towards integration, microprocessors are increasingly supporting multiple cores on a single chip. To keep design effort and costs down and to adapt to future applications, these multiple core microprocessors frequently target an entire product range, from mobile laptops to high-end servers. This presents a difficult trade-off to processor designers: balancing single-thread performance critical for laptop and desktop users, with system throughput critical for server applications. Further, due to the growing gap between processor cycle time and memory access latencies, the processor pipeline increasingly stalls waiting for data in the event of a miss to memory. Achieving high single-thread performance in the presence of such relatively increasing memory latencies has traditionally required large and complex cores to sustain a large number of instructions in flight while waiting for memory. However, achieving high system throughput requires many such cores on the same chip. Unfortunately, the mostly constant chip size and power envelopes result in two contradictory goals: many large cores cannot be placed on a single chip and small cores traditionally do not provide high single-thread performance.

This paper bridges the above dichotomy by presenting *Continual Flow Pipelines*: a processor core architecture that can sustain a very large number of in-flight instructions without requiring the cycle-critical structures to scale up. By keeping these structures small while being able to tolerate memory latencies, the new core can achieve high single-thread performance while allowing multiple such cores on a chip for high throughput. The resulting large instruction window[1] exposes large amounts of instruction level parallelism (ILP) and achieves memory latency tolerance, while the small size of cycle-critical resources allows for a high clock frequency.

Continual Flow Pipelines (CFP) allow the processor to continue processing instructions even in the presence of a long latency cache miss to memory by being efficient in managing cycle-critical resources. In conventional processor designs, a load operation that misses the cache, and any later instructions that depend on this load for data continue to occupy cycle-critical structures such as the register file and scheduler. These blocked and miss-dependent instructions stall the processor since later instructions that are not dependent upon the miss (miss-independent) and can execute are unable to proceed due to a lack of sufficient register file and scheduler resources.

CFP on the other hand ensures that miss-independent instructions will successfully acquire register file and scheduler resources by making these resources non-blocking: the long-latency load operation and its dependent instructions release these resources early once they are known to be miss-dependent. This allows later miss-independent instructions to execute and complete in parallel with the outstanding memory miss. Prior research has shown a significant amount of useful work can be done in the shadow of a memory miss [13]. CFP uses that observation to achieve memory latency tolerance. By providing mechanisms to obtain non-blocking structures, the register file and scheduler sizes are now

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, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ASPLOS'04, October 9–13, 2004, Boston, Massachusetts, USA.
Copyright 2004 ACM 1-58113-804-0/04/0010...$5.00.

[1] All instructions renamed but not yet retired constitute the instruction window. In reorder buffer based (ROB) processors, every instruction that has a reorder buffer entry allocated is considered part of the instruction window.

functions of the number of instructions actively executing in the processor pipeline and not the total number of in-flight instructions in the processor. Eliminating structural stalls in the pipeline implies performance is now limited only by the rate at which the front-end feeds useful instructions into the pipeline.

With CFP, a load that has missed in the cache and its dependent instructions (the forward slice of the load) drain out of the pipeline freeing any scheduler and register file entries they may have occupied. We call this load and its dependents the *slice instructions*. The Slice Processing Unit (SPU) is responsible for managing these slice instructions while the miss is pending. The SPU holds the slice instructions and has all information necessary to execute the slice instructions, including their completed source register values, and data dependence information. Storing values corresponding to source registers written by completed instructions, allows the release of such registers even before a consuming slice instruction completes. The SPU also ensures correct execution when slice instructions are re-mapped and re-introduced into the pipeline. By ensuring schedulers and physical registers are not tied down by slice instructions, the pipeline achieves a continual flow property where the processor can look far ahead for miss-independent instructions to execute. Further, efficient integration of results from executing miss-independent instructions in the instruction window is possible by means of a checkpoint mechanism and without requiring their reexamination.

CFP involves two key actions: draining out the long-latency-dependent slice (along with ready source values) while releasing scheduler entries and registers, and re-acquiring these resources on re-insertion into the pipeline. The CFP concept is applicable to a broad range of processor architectures (see Section 4.3). In this paper we use Checkpoint Processing and Recovery (CPR) as the baseline architecture [2] since it has been shown to outperform conventional ROB-based architectures. CPR is a reorder-buffer-free architecture requiring a small number of rename-map table checkpoints selectively created at low-confidence branches, and capable of supporting an instruction window of the order of thousands of instructions. In addition to decoupling the instruction window from the ROB, CPR provides a scalable hierarchical solution for store queues. However, long latency operations expose CPR resource limitations. CFP addresses CPR limitations in the presence of long latency operations. The synergistic interplay between CPR and CFP allows for the design of truly scalable large instruction window memory latency-tolerant processors.

Paper contributions: CFP is a unified proposal for decoupling the demands of sustaining a large instruction window from both the scheduler and register file. The key contributions are:

- **Non-blocking register file.** CFP presents the first proposal where instructions waiting for long latency operations do not block registers. Doing so avoids using large and complex register files. CFP introduces the notion of a back-end renamer that assigns new physical registers to previously renamed slice instructions using physical-to-physical remapping as opposed to the front-end renamer that performs logical-to-physical remapping. This allows CFP to store only slice instructions in data dependence order without requiring slice storage to scale with instruction window size.
- **Unified mechanism for non-blocking register file and scheduler.** By decoupling instruction window demands from cycle-critical structures (register file, scheduler) in a unified manner, CFP allows for a high-performance and scalable processor core that can support a dynamic and adaptive instruction window (100s to 1000s of instructions).
- **CFP outperforms larger ROB and CPR cores.** For example, a CFP core with 8 map table checkpoints and a 96-entry register file outperforms a 256-entry ROB machine with a 192-entry register file and a much larger scheduler by more than 20% for server suites, 33% for workstation suites, 28% for SPECFP2K, and 5% for SPECINT2K.
- **CFP effectively tolerates long memory latencies.** A CFP core tolerates long memory latencies and isolates branch prediction accuracy as the primary performance limiter for future processors.
- **CFP lends to highly efficient cache hierarchies.** A CFP core with a 512 KB L2 cache outperforms ROB-based cores with 8 MB L2 caches for most benchmarks. For constant die size, such efficient cache hierarchy allows more cores to be placed (instead of cache) to achieve high throughput while still achieving high single thread performance compared to conventional large cache designs.

Section 2 outlines our simulation methodology and benchmark suites. Section 3 provides an overview of our baseline CPR core and quantifies its limitations. Section 4 describes Continual Flow Pipelines and Section 5 presents a performance analysis of CFP. Section 6 discusses CFP comparatively with two other techniques for building latency tolerant processors and Section 7 discusses implications of CFP on future research. Section 8 presents related work and we conclude in Section 9.

2. SIMULATION METHODOLOGY

We use a detailed execution-driven timing simulator for simulating the IA32 instruction set and micro-ops (uops). The simulator executes user and kernel instructions, models all system activity such as DMA traffic and system interrupts, and models a detailed memory system. Table 1 shows parameters of our baseline ROB-based processor based on the Pentium® 4 [12].

All experiments in the paper use an aggressive hardware data prefetcher and a perfect trace cache (*i.e.*, no trace cache misses). Table 2 lists the benchmark suites, the number of unique benchmarks within each suite, and the L2 cache load miss rates. Unless specified, all performance numbers in graphs shown are percent speedups calculated over the ROB-based baseline processor. The baseline CPR processor replaces the reorder buffer with 8 checkpoints created at low confidence branches. In addition, CPR employs larger load and store buffers. The baseline CPR employs a hierarchical store queue with a 48-entry conventional L1 store queue (on the critical path) and a large and slow (L2 cache access latency) 1024-entry L2 store queue (off the critical path). The large L2 store queue is necessary for some benchmark suites to achieve high performance. The baseline CPR uses a store sets predictor [7] to predict load-store memory dependences and to issue loads ahead of unknown stores. Completed stores look up a load buffer to roll back execution to an earlier checkpoint in case of memory dependence

Table 1 ROB processor model

Processor frequency	8 GHz
Rename/issue/retire width	4/6/4
Branch mispred. penalty	Minimum 20 cycles
Instruction window size	256
Scheduler	64 int., 64 fp., 32 mem.
Register file	192 int., 192 fp.
Load/store buffer size	64/48
Memory dependence pred.	Store sets
Functional units	Pentium® 4 equivalent
Branch predictor	Gshare (64K)-perceptron (256) hybrid
Hardware data prefetcher	Stream-based (16 streams)
Trace cache	4-wide, perfect
Level 1 (L1) Data cache	32 KB, 3 cycles (line 64 bytes)
Level 2 (L2) cache	2MB, 8 cycles (line 64 bytes)
Load-to-use latency to memory	100 ns (includes DRAM latency and transfer time)
Max. outstanding misses	128

mispredictions. The load buffer has 2048 entries, is set-associative, and is not in the critical path as it does not store or forward any data.

3. QUANTIFYING CPR PERFORMANCE

In this section, we analyze the performance of our baseline CPR processor. We provide an overview of CPR in Section 3.1 and highlight key differences over ROB-based processors. Section 3.2 presents performance limitations of CPR and Section 3.3 quantitatively analyzes these limitations.

3.1 CPR overview

CPR is a ROB-free proposal for building scalable large instruction window processors [2]. CPR addresses the scalability and performance limitations of conventional branch misprediction recovery mechanisms by using a small number of register rename map table checkpoints selectively created at low-confidence branches. CPR employs this checkpoint mechanism to implement a register reclamation scheme decoupled from the reorder buffer, and for providing precise interrupts. With CPR, instruction completion is tracked using checkpoint counters and entire checkpoints are committed instantaneously thus providing the appearance of a bulk commit and breaking the serial commit semantics imposed by a ROB. CPR aims to decouple key processor mechanisms of misprediction recovery, register reclamation, and commit operations from the reorder buffer.

To allow for fast and efficient forwarding of data from stores to subsequent loads while supporting a large number of stores at any time, CPR uses a hierarchical store queue implementation (see Section 2). The level one queue holds most recent stores while the level two holds older stores displaced from the level one.

3.2 Quantifying CPR performance potential

Figure 1 presents the performance gap between a base and an ideal CPR implementation. We vary scheduler and register file parameters alone because limiting these structures account for most pipeline stalls in the baseline processor. The *y*-axis shows

Table 2 Benchmark suite

Suite	Num. Bench	Examples	L2$ load misses /1000 uops
SPECFP2K (SFP2K)	13	www.spec.org	7
SPECINT2K(SINT2K)	8	www.spec.org	< 1
Internet (WEB)	15	SPECJbb, WebMark	1
Multimedia (MM)	18	MPEG, photoshop, speech	2
Productivity (PROD)	8	SYSMark2k, Winstone	< 1
Server (SERVER)	7	TPC-C	1
Workstation (WS)	18	CAD, rendering	11

percent speedup over the baseline ROB processor of Table 1 (for corresponding baseline frequencies of 3 GHz and 8 GHz). The figure shows two different frequencies corresponding to current and future processor frequencies to understand effects of increasing relative memory latencies. Base CPR uses the register file and scheduler of Table 1. The scheduler is blocking—a long latency operation and its dependents stay in the scheduler occupying entries. The ideal CPR relaxes the register file and scheduler constraints by assuming infinite entries for each. The remaining machine parameters for the base CPR and ideal CPR are the same as Table 1. For a 4-wide 8-GHz processor and a 100 ns load-to-use latency for a miss to memory, a single load miss requires a peak target window of 3200 (=4X8X100) instructions. The presence of a second load miss that depends on the first load miss in the window results in another 100 ns stall right after the first 100 ns stall completes, thus suggesting up to a 6400-entry instruction window to tolerate such a miss. Thus, miss-dependent misses result in greater pressure on the instruction window. We experimentally observed performance gains for some benchmarks as we scaled the target window up to 8192 instructions and this data is presented later. We therefore assume an 8192-entry target instruction window to tolerate latencies across the large set of benchmarks. We emphasize this is a conceptual target since CPR and CFP do not require any hardware structure to scale up to the total number of in-flight instructions, as we will discuss later.

As can be seen, while base CPR provides high performance over a ROB-baseline, the performance gap between base CPR and ideal CPR is substantial for the given register file and scheduler size, and becomes even larger as relative memory latencies increase.

This performance gap between base and ideal, and the negative impact of increasing memory latencies serve as quantitative motivation to investigate new solutions. The next section presents a quantitative analysis to identify the sources of this gap.

3.3 Understanding CPR limitations

This section analyzes the interaction of the scheduler and register file. Figure 2 shows percent speedup over the baseline ROB for five CPR configurations. In the figure, the "perf" prefix indicates a sufficient size of the corresponding resource for an 8192-entry instruction window, and the "real" prefix indicates the corresponding resource size listed in Table 1. The left most bar (first bar) for each benchmark suite corresponds to a baseline CPR model discussed in Section 2 and with the register file and scheduler configuration listed in Table 1. The second bar corresponds to the baseline CPR combined with a perfect store queue (unlimited entries, single-cycle access) and perfect memory dependence predictor (perfSTQ/MD). Since the performance gap

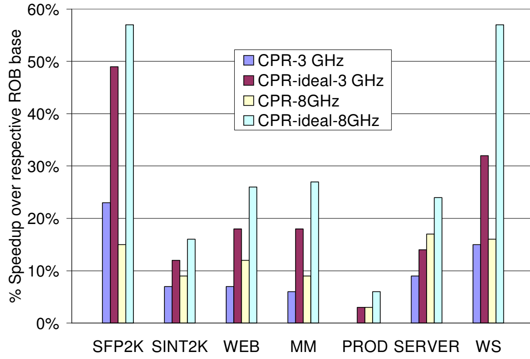


Figure 1 Limit study of CPR processors.

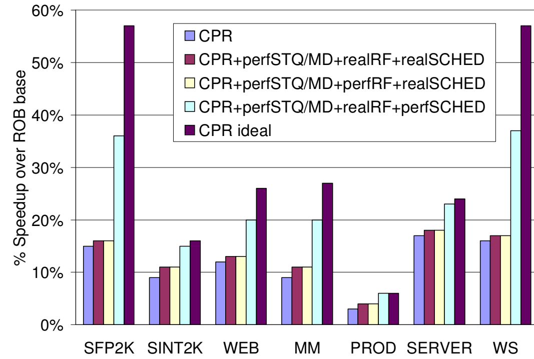


Figure 2 Understanding sources of performance gap.

between these first two bars is small, we do not study the memory dependence predictor and store queue any further in this paper.

The third bar from the left corresponds to a configuration with a perfect register file (unlimited entries, single-cycle access) and the baseline scheduler; the fourth bar corresponds to a configuration with a baseline register file and a perfect scheduler (unlimited entries, single-cycle scheduling); and the fifth bar (right most) corresponds to both the register file and scheduler being perfect.

Increasing the register file without increasing the scheduler does not provide performance benefits. Idealizing the scheduler while keeping the register file finite does help performance but a substantial performance gap remains between this configuration and one where both, register file and scheduler, are ideally sized. The results indicate the importance of increasing the size of both resources to achieve substantial performance gains. We now qualitatively analyze this result.

3.3.1 Schedulers

Our baseline CPR model uses a decentralized blocking scheduler. In blocking schedulers, entries may fill up in the presence of long latency operations. The Pentium 4 [12] uses a decentralized and non-blocking scheduler—a long latency operation (an L1 miss for the Pentium 4) and its dependent instructions do not occupy scheduler entries while waiting for the miss to return. The distributed non-blocking design allows for an effectively large scheduler. The Waiting Instruction Buffer (WIB) [14] employs a small and fast scheduler backed by a larger buffer for storing instructions dependent upon long latency operations.

While non-blocking scheduler designs such as the Pentium 4 style replay mechanism and the WIB allow the release of scheduler entries occupied by instructions dependent on a long latency operation, they assume sufficiently large register files. The fourth bar in Figure 2 corresponds to a CPR processor with a non-blocking scheduler but with limited register file size—an ideal scheduler configuration makes a blocking scheduler behave the same as a non-blocking scheduler because instructions never wait for a scheduler entry. From Figure 2, we see that while a non-blocking scheduler can solve scheduler limitations, a substantial performance gap remains.

3.3.2 Register files

Conventional physical register reclamation algorithms free a register conservatively by reclaiming a physical register only when it is no longer part of architectural state. This happens when the instruction that overwrites the logical register, which maps to the physical register, commits. While this allows architectural state restoration on branch mispredictions, interrupts, and exceptions, physical register lifetime artificially increases because reclamation is limited by the serial retirement semantics of the reorder buffer. This limits the supply of free registers and necessitates larger register files.

Our baseline CPR model uses an aggressive counter-based register reclamation scheme [17] instead of relying on a ROB-based reclamation scheme. CPR thus breaks the limitation of serial register reclamation by guaranteeing architectural state is never lost, while aggressively reclaiming all physical registers except those corresponding to a checkpoint (physical registers mapped to logical registers at the time of checkpoint creation).

While the CPR scheme works well for short latency operations, long latency operations result in increased register file pressure. Consider a blocked instruction dependent upon a long latency operation. In a typical implementation, the physical register corresponding to its destination operand cannot be freed at least until the blocked instruction completes. Further, if a source operand depends on a short latency operation independent of the long latency operation, the physical register corresponding to that source operand cannot be freed even if the short latency producer has completed, until the current blocked instruction executes. These factors restrict availability of free physical registers, and thus stall execution and subsequently limit performance. Even with the aggressive register reclamation techniques of CPR, a significant performance gap remains (Figure 2).

To bridge this performance gap using conventional schemes, sufficient physical registers must be made available for the target window. For example, an instruction window of the order of thousands of entries needed to tolerate future memory latencies would typically require thousands of physical registers in conventional processors. Simply increasing register file size significantly degrades cycle time. Hierarchical solutions have been proposed to reduce this cycle time degradation by adding a small fast level backed by a slower and larger second level. In both cases, the register file size is still tightly coupled to the instruction window size and the designs are not resource-efficient since all physical registers need to be pre-allocated and sufficient buffering must be available a priori. Further, such multilevel designs introduce complexity in the pipeline because the core has to deal with variable access latencies, and these hierarchies need to be managed carefully to prevent performance degradation.

4. CONTINUAL FLOW PIPELINES

Continual Flow Pipeline (CFP) comprises a microarchitecture where cycle-critical structures of a processor, namely the scheduler and register file, are sized independent of the target instruction window required for tolerating increasing memory latencies. Section 4.1 discusses key concepts of CFP for providing a unified mechanism for a non-blocking scheduler (continual flow scheduler) and non-blocking register file (continual flow register file), and Section 4.2 discusses CFP implementation. Because of the inherent scalability characteristics of CPR, CFP uses CPR as the base processor. Section 4.3 discusses CFP in the context of different baseline processors.

4.1 De-linking instruction window from cycle-critical structures

Inefficiencies in management of the cycle-critical register file and scheduler in the presence of long latency operations limit the lookahead required to tolerate such latencies. Removing these inefficiencies requires the register file and scheduler resources to be decoupled from demands of the instruction window. This requires a non-blocking design for the scheduler and register file, where an instruction does not tie down entries in these structures in the event of a long latency miss.

We discuss key CFP mechanisms for achieving a continual flow scheduler (Section 4.1.1), a continual flow register file (Section 4.1.2), and for re-inserting long latency dependent instructions into the pipeline, even after all their scheduler entries and registers have been previously reclaimed (Section 4.1.3). Since we are concerned with instructions dependent upon a long latency operation, we refer to these instructions as the forward slice of the long latency operation, or simply the slice instructions.

4.1.1 Continual flow scheduler

To achieve a continual flow scheduler, CFP uses a first-in first-out (FIFO) buffer to store temporarily the slices corresponding to long latency operations in the instruction window. This is similar to the WIB except unlike the WIB where the buffer must be the size of the target instruction window (*i.e.*, the ROB size itself since the WIB targets a ROB-style processor) CFP only buffers the actual slice, which is significantly smaller than the instruction window (See Section 6).

Since CFP targets instructions experiencing long latency L2 cache misses, such instructions along with their slice immediately leave the scheduler, thus freeing scheduler resources for subsequent instructions. For this, slice instructions treat their source registers, dependent on long latency operations, as ready, even though they may not actually be ready. This allows the slice to drain out of the scheduler and into another buffer without changing the scheduler design. Other instructions not dependent on long latency operations (*e.g.*, those that hit the L1 or L2 cache) naturally leave the scheduler quickly as they become ready to execute.

4.1.2 Continual flow register file

With mechanisms for a continual flow scheduler in place, the next step is developing mechanisms for a continual flow register file. For ease of discussion, we define two classes of registers:

Completed source registers: These are registers mapped to instructions that have completed execution, and are read by instructions in a slice of a long latency operation. Conventional register reclamation cannot free these registers until their values are read by the dependent slice instructions, which may be hundreds of cycles later. Since we assume at most two input operands for an instruction, at least one operand of a slice instruction must depend upon a long latency operation. An instruction in the slice cannot have both source operands correspond to completed source registers.

Dependent destination registers: These are registers assigned to the destination operand of slice instructions. These registers will not be written at least until the long latency operation completes, which may take hundreds of cycles. Conventional reclamation ties these registers down unnecessarily for many cycles.

CFP enables a continual flow register file by providing mechanisms for reclaiming the above two register classes. When an instruction within the slice of a long-latency instruction leaves the scheduler and drains through the pipeline, it reads any of its completed source registers, records the value as part of the slice, and marks the register as read. Since this instruction has the completed source register value available, the register storage itself is now available for reclamation once all readers have dispatched. The slice instruction also records its physical register map as part of the slice. The physical map is used to maintain true data dependence order among slice instructions. The slice in the buffer is a self-contained subset of the program and can execute independently since appropriate ready source values and the data dependence order are available.

With CFP, the ready source values are in the SDB, and any destination physical register will be re-acquired (Section 4.1.3) when the slice re-executes. This allows release of both classes of registers discussed above associated with slice instructions. Registers can now be reclaimed at such a rate that whenever an instruction requires a physical register, such a register is always shortly available—hence achieving the continual flow property.

We now discuss the conditions under which actual reclamation occurs. To recover by using checkpoints, two types of registers cannot be released until the checkpoint is no longer required: registers belonging to the checkpoint's architectural state, and registers corresponding to the architectural live-outs. These however are small in number (∝logical registers) as compared to the physical register file. Further, this state does not depend upon implementation details such as outstanding misses, target instruction window size, *etc*.

Other physical registers can be reclaimed when (1) all subsequent instructions reading the registers have read them, and (2) the physical registers have been subsequently re-mapped, *i.e.* overwritten. CFP guarantees condition 1 because slice instructions mark completed source register operands as read before they even complete but after they read the value. Condition 2 is met due to the basic register renaming principle—for L logical registers, at the latest the $(L+1)^{th}$ instruction requiring a new physical register mapping will overwrite an earlier physical register mapping. Thus, for every N instructions (with a destination register) leaving the pipeline, N-L physical registers will be overwritten and hence will satisfy condition 2.

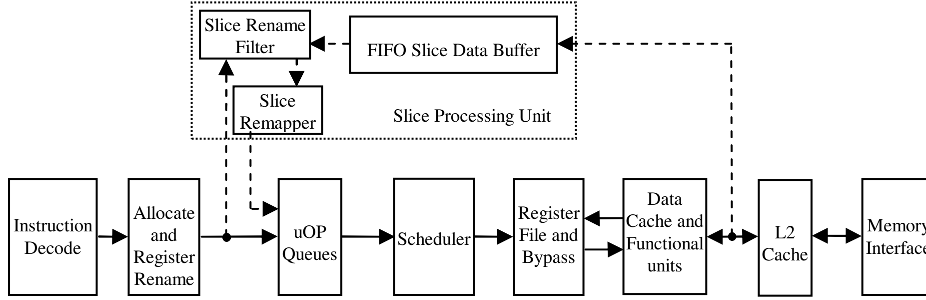


Figure 3 Block diagram of a CFP processor.

4.1.3 Re-introducing the slice back into the pipeline

Slice instructions re-enter the pipeline when the long latency data returns. Since destination registers for these instructions have been released, these instructions are remapped to new physical registers by means of back-end renaming.

Re-acquiring registers with back-end renaming and without deadlocks. Unlike conventional front-end renaming where a logical register maps to a physical register, in back-end renaming a physical register maps to another physical register. When the long latency operation completes, new front-end instructions wait until the slice instructions drain into the pipeline. Doing so allows these re-entering slice instructions to acquire scheduler entries, thus guaranteeing their forward progress and avoiding deadlock. These slice instructions also require new physical registers. New free registers are guaranteed by: (1) exploiting the earlier mentioned basic register renaming principle where registers in the slice are reclaimed when they are overwritten and an overwriting instruction is guaranteed to appear within a fixed number of rename operations, and (2) sizing the register file a priori to guarantee the slice remapper eventually finds a free physical register even when some physical registers cannot be reclaimed because they belong to checkpoints and architectural live-outs.

We now explain how the a priori sizing is done when physical-to-physical back-end renaming is used. Assume P_{FE} is the number of physical registers available in the front end for renaming logical to physical registers. The slice remapper, used for back-end renaming, will thus observe potentially P_{FE} unique physical registers while performing the physical to physical remapping. One uncommitted checkpoint is guaranteed to exist prior to the oldest load miss in the SDB (since we use a checkpoint-based CPR baseline). Thus, the physical registers belonging to that checkpoint (such registers totaling L in number) cannot be released until the checkpoint retires and will not appear as destination physical registers among the slice instructions. Hence, the slice remapper only sees (P_{FE} – L) unique physical register names in the SDB. If the slice remapper has (P_{FE} – L + 1) physical registers available for remapping, deadlock is avoided by the renaming principle discussed earlier. In an implementation that supports C checkpoints, a maximum of (C * L) physical registers may be checkpointed in the window and thus may not be available to the slice remapper. Live-outs occupy an additional L physical registers unavailable to the slice remapper. Thus, to avoid deadlocks, the slice remapper needs (P_{FE} – L + 1) physical registers, but may actually have only (P_{FE} – [(C + 1)*L]) physical registers available to it. Hence, we need to have additional (C*L + 1) physical registers reserved only for the slice remapper. This number is dependent only on the number of checkpoints and the architectural registers and not on any other implementation details such as miss latency or target instruction window size.

Synchronizing dependences between slice and new instructions. Since new instructions fetched after slice reinsertion may have dependences on slice instructions, the slice destination registers that are still live (corresponding to the live-outs) at the time of slice reinsertion, must not be remapped by the slice remapper. This ensures that new instructions will correctly synchronize their dependences to the live registers updated by the slice instructions. A rename filter is used for this and is discussed later.

In summary, with CFP, instructions dependent upon a long latency operation drain from the pipeline along with the long latency operation, releasing any register file and scheduler entries. These instructions, take along any values read from completed source registers, and the data dependence order (via physical maps) into a buffer. They retain all information for execution without requiring reading of the earlier ready sources again from the register file.

4.2 CFP implementation

We now discuss implementation of CFP mechanisms. We present changes to a baseline CPR processor. Section 4.2.1 describes how instructions in the slice of the long latency operation are identified and how they release scheduler and register file entries prior to completion. Section 4.2.2 presents the Slice Processing Unit (SPU). The SPU is responsible for buffering the slice while the long latency miss is outstanding, and for processing the slice prior to re-introduction into the pipeline. While a concrete area model is beyond the scope of the paper, we provide size approximations for structures CFP adds. These additions consist largely of dense SRAM structures and tables whose sizes need not scale with target instruction-window size.

Although we only consider L2 cache misses as long latency operations, other operations such as long-latency floating-point operations may also trigger CFP, if necessary. Figure 3 shows a block diagram of CFP in a conventional processor.

4.2.1 Releasing scheduler entries and physical registers associated with slice instructions

Slice instructions are identified dynamically by tracking register and memory dependences of long latency operations. A bit, *Not a Value* (NAV bit) initialized to zero, is associated with each physical register and store queue entry. On an L2 cache load miss, the NAV bit of the load's destination register is set to one. Subsequent instructions reading this register inherit the NAV bit

for their destination registers. If the destination is a store queue entry, the entry's NAV bit is also set. Destination registers of loads predicted to depend upon an earlier NAV store by the memory dependence predictor also have their NAV bits set, thus constructing dependence between stores and loads. An instruction is in a slice if it reads a source operand register or a store queue entry with a NAV bit set. A register with its NAV bit set is considered ready. Once both sources are ready (or have the NAV bits set), the instruction drains through the pipeline.

During the draining process, slice instructions mark their source registers, whether NAV or completed, as having been read. As in CPR, for CFP this involves decrementing the use counter for the register. The registers are reclaimed when the reclamation conditions discussed in Section 4.1 are met. Although source registers of slice instructions are marked as read, slice instructions do not decrement the checkpoint instruction counter (used to track instruction completion in CPR) since they are not completed. Non-slice instructions however execute and decrement the checkpoint's instruction counter to signal their completion. Only the slice instructions move into the Slice Processing Unit (SPU).

4.2.2 The Slice Processing Unit

The Slice Processing Unit (Figure 3) serves two primary functions:

1. It holds slice instructions, physical maps, and source data values for the slice in the Slice Data Buffer (SDB), and
2. It remaps slice instructions prior to re-introduction into the pipeline using the Slice Rename Filter and Slice Remapper.

Slice Data Buffer. Two actions on slice instructions determine the ordering information required in the SDB:

a) *Back-end renaming when registers of slice instructions are released and re-acquired*. If the slice remapper uses logical names for renaming, then the original program order is required. This is similar to a conventional front-end renamer. However, if physical names are used for renaming, then the data dependence order among slice instructions is sufficient.

b) *Branch misprediction recovery*. When a branch mispredicts, slice instructions after that branch must be identified and squashed. If a ROB-based misprediction recovery mechanism is used, then the slice buffer needs to maintain program order to allow correct identification of all instructions after the misprediction. However, if CPR-style map table checkpoints are used for misprediction recovery, then slice instructions after the mispredicted branch's checkpoint are easily identified and squashed using the checkpoint identifiers of the slice instructions.

As our baseline CPR uses checkpoint-based misprediction recovery and the slice remapper uses physical names, the SDB stores the slice instructions in data dependence order and not in program order. Since the SDB only stores slice instructions in data dependence order, it does not require tracking program order of the entire window. Thus, the SDB size is significantly smaller than the instruction window size.

Each entry in the FIFO SDB has the following fields shown in Figure 4: 1) instruction opcode, 2) one source register's data field to record values from a completed source register, 3) two source register mappings, 4) one destination register mapping, and 5) some control bits. These control bits manage re-insertion into the SDB and squashing of SDB entries in the event of branch mispredictions and other exceptions.

Opcode	Ready Source Data	Src1 Register Mapping	Src2 Register Mapping	Dest Register Mapping	Control Bits

Figure 4 SDB entry fields.

The SDB has sufficient bandwidth to allow writing and reading blocks of instructions. A block corresponds to the width of the machine. The single read-port and single write-port block organization for the SDB array is a simple, high-density, and effective implementation. SDB entries are allocated as slice instructions leave the scheduler. Blocks of slice instructions enter the SDB in scheduler order, and leave it in the same order.

The SDB can be implemented using a high density SRAM. One possible area-efficient design is a long-latency, high-bandwidth cache-like array structure with latency and bandwidth requirements similar to an L2 cache. Since memory latencies are high, operations on the SDB, including reading and writing, are not on the critical path. We model a 25 cycle latency for processing an instruction through the SPU (including insertion into the SDB, removal from the SDB, and processing through the remapper and filter), and observe no performance impact, since the additional latency is small compared to L2 cache miss latencies. An SDB entry corresponds to approximately 16 bytes, and our experiments presented in Section 5.2 suggest an SDB of few hundred entries is sufficient to achieve high performance.

Slice Remapper. Slice instructions leaving the SDB may require new registers. The slice remapper maps the physical registers in the slice to new physical registers. The slice remapper has access to the front-end mapper name space and a small number of reserved registers (see Section 4.1.3). The slice remapper has as many entries as the number of physical registers and records only map information. We believe CFP is the first proposal using a physical-to-physical remapper for slice instructions thus allowing back-end renaming and front-end renaming to coexist seamlessly.

Slice Rename Filter. Registers corresponding to live-outs must retain the original physical map. The Slice Rename Filter is used to avoid remapping these registers. The filter has an entry for each logical register and records the identifier of the instruction that last wrote the logical register. When an instruction is renamed in the front-end renamer, the rename filter is updated with the instruction identifier. If this instruction enters the SDB, on its subsequent reinsertion into the pipeline, the rename filter is looked up using the logical destination register of the instruction. If the instruction identifier returned by the lookup matches the current instruction being re-inserted into the pipeline, the register is considered still live as no later instruction has remapped it. The original physical mapping for this register must be retained so that any new front-end instructions will see the correct physical register. These registers do not pass through the slice remapper. If the identifier does not match, the register has been remapped and the slice instruction must now acquire a new register using the slice remapper. Physical registers corresponding to checkpoints are also handled similarly. The number of entries in the rename filter is *(number of logical registers) X (number of checkpoints).*

SDB and multiple independent loads. The SDB can have more than one slice in the event of multiple independent load misses. Since these slices are self-contained program slices (and include source operand values and dependences), they can be drained in any order. The only live-ins are the values of loads that missed. Load misses may complete out of order; a slice belonging to a later miss in the SDB may be ready prior to an earlier slice in the SDB. Three approaches to handle this are: 1) wait until oldest slice is ready and drain the SDB in a first-in first-out order, 2) drain SDB in a first-in first-out order when any miss in the SDB returns, and 3) drain the SDB sequentially from the miss serviced (may not be the oldest slice in the SDB). With (3), register mappings from an earlier waiting slice that source instructions in a later ready-to-execute slice are detected using the slice remapper. We use approach 3. Identifying the correct SDB entry corresponding to the load miss is easily achieved by using the SDB index stored in the outstanding miss cache fill buffer.

SDB and chains of dependent loads. A slice may have chains of dependent loads that may miss in the L2 cache. When a miss returns, the load's slice leaves the SDB and re-enters the pipeline. Subsequently, another load in the slice could immediately miss the L2 cache. Further, instructions dependent on this miss may be part of the slice of the earlier load miss. These instructions re-enter the SDB, thus preventing a stall in the event of a chained dependent load miss. However, these instructions must occupy their original position in the SDB along with their original physical map to maintain data dependence order of the original program. Thus, the release of an SDB entry occurs only if the slice instruction associated with it successfully executes and completes. Slice instructions re-entering the SDB discard any new mapping they obtained in the slice remapper and retain the original mapping already stored. They also read any newly produced source register values and store them when they re-enter the SDB. The head instruction in the slice always completes successfully thus guaranteeing forward progress. The SDB re-insertion may result in empty entries for instructions that completed and did not require re-insertion. This may reduce write bandwidth into the SDB because now, instead of blocks of entries, individual entries are written. We add a write-combining queue in front of the SDB to match the single port SDB design with the pipeline. Since these re-inserted instructions are needed again only after hundreds of cycles (L2 cache miss latency), any additional latency of the write-combining queue has no impact on performance.

4.3 CFP and base processor interactions

The two core CFP principles are: 1) draining out the long-latency-dependent slice (along with ready source values) while releasing scheduler entries and registers, and 2) re-acquiring these resources on re-insertion into the pipeline. While this paper has discussed these principles and their implementation in the context of a CPR processor, they are applicable as well to conventional processors such as ROB-based out-of-order and in-order processors. Conventional processors, independent of their architectural decisions find it increasingly difficult to tolerate very long latencies such as L2 cache misses to memory and thus can benefit from CFP (out-of-order execution and compiler techniques can often assist in tolerating shorter latencies such as L1 miss/L2 hit conditions, but not L2 cache misses).

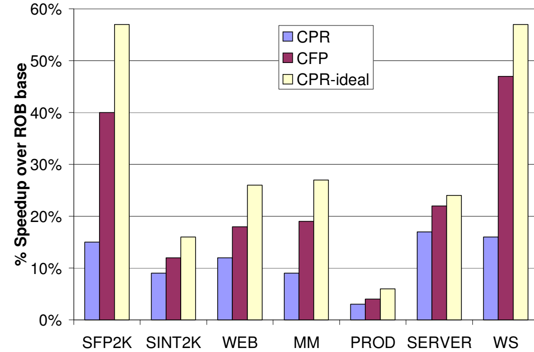


Figure 5 Performance comparison of ROB, CPR, and CFP.

Since CPR already provides checkpoints to enable coarse-grained scalable recovery and to re-generate architectural state whenever necessary, support for recovering to a point prior to the load miss for CFP (in the event of exceptions and mispredictions in the slice) already exists. For ROB-based and in-order processors, a form of roll-back support (although at a very coarse-grain) will be required in the form of a minimum single checkpoint to support CFP. The CPR-based CFP implementation provided in this paper is meant as a general solution and simplifications can be envisioned in the implementation if more restrictive baseline processors such as ROB-based or in-order models are employed. In either case, one only ensures the two core principles of CFP discussed above are maintained.

CPR is an attractive base processor model on which to build CFP since in addition to having support for coarse-grain checkpoints, CPR achieves higher performance than a ROB-based model and has benefits in increasing ILP by being efficient in managing resources and mechanisms even in the absence of long latency misses. CFP improves the resource efficiency of CPR in the presence of long latency operations and CPR's selective checkpoint property allows for a high-performance baseline. This synergistic interplay between CFP and CPR allows for design of truly scalable instruction window processors.

5. RESULTS AND ANALYSIS

Section 5.1 compares CFP performance to CPR and ROB baselines and Section 5.2 analyzes sources of performance. Section 5.3 discusses power implications and Section 5.4 compares the resource efficiency of CFP, CPR, and ROB processors across various configurations.

5.1 CFP performance

Figure 5 shows percent speedup of CPR, CFP, and an ideal CPR configuration over the baseline 256-entry ROB processor from Table 1. Both CFP and CPR configurations match the ROB baseline model parameters except for the use of 8 checkpoints instead of the 256-entry ROB and a hierarchical store queue organization.

The ideal CPR configuration has unlimited scheduler and register file entries. CFP outperforms both the ROB and CPR implementations across all benchmark suites. Moreover, CFP significantly bridges the gap between the baseline CPR and ideal CPR implementation and achieves 75% of the ideal CPR speedup.

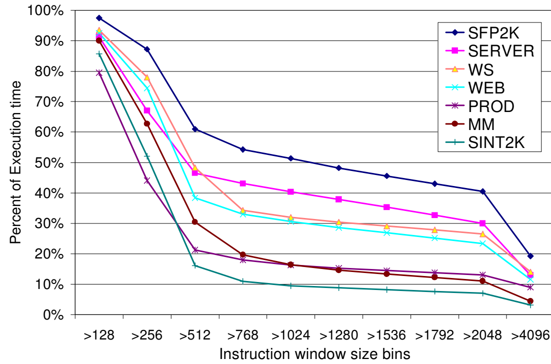


Figure 6 Instruction window size distribution.

Table 3 Latency Tolerance**:** Percent of independent instructions retired in the shadow of a L2 miss

SFP2K	SINT2K	WEB	PROD	MM	SERVER	WS
80%	86%	82%	85%	73%	90%	80%

The remaining gap between CFP and ideal CPR exists because the scheduler fills up with chains of multi-cycle latency instructions (loads that miss the L1 cache and hit the L2, multi-cycle floating point operations *etc.*) and their dependent instructions. While the individual instruction latencies are relatively small, the combined latency of an entire such chain may be long. Such chains can degrade performance since dependent instructions in the chain consume scheduler and register file entries. CFP does not target such situations since it aims at tolerating longer L2 cache misses. Schemes such as employed by the Pentium 4 for targeting L1 cache miss latencies, when used in conjunction with CFP, may eliminate these scheduler stalls.

5.2 CFP performance analysis

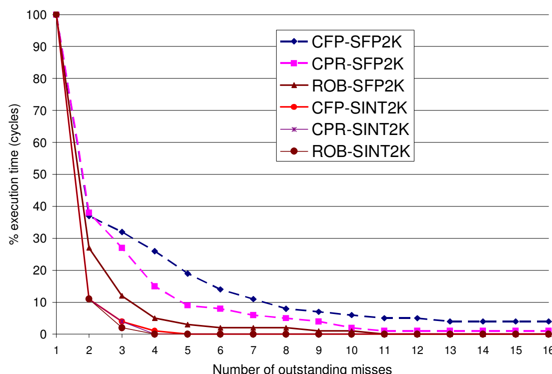
Figure 6 shows instruction-window size distribution with CFP. As we can see, a CFP processor is able to achieve an instruction window in the thousands for a significant fraction of the execution time. More than 2048 instructions are in the window for nearly 40% of the time for the floating-point benchmarks, and more than 20% of the time for the server, workstation, and internet benchmarks. Between 4096 and 8192 instructions are in the window 5% to 20% of execution time depending upon the suite.

CFP performance gains are due to two factors: 1) its ability to tolerate long memory latencies and 2) its ability to generate a cache miss early thus exposing memory level parallelism. CFP's ability to tolerate long memory latencies can be measured by the fraction of independent instructions that complete (and are eventually committed *i.e.*, were not on a misspeculated path) in the shadow of a long latency miss operation (shown in Table 3). As can be seen, a significant portion of the non-speculative instruction-window in the shadow of a long latency miss comprises miss-independent instructions for all benchmark suites. Others [13] have also observed similar results for the SINT2K suite. Thus, a significant amount of useful work can be completed while a miss is outstanding allowing the processor to tolerate such latencies. More importantly, CFP can tolerate even isolated cache misses—situations that cannot be handled by other techniques such as Runahead execution (discussed later).

In addition to tolerating cache miss latencies, CFP also increases memory level parallelism by getting to a cache miss sooner. Figure 7 shows the outstanding L2 cache miss distribution for four suites and the CPR, ROB, and CFP architectures. As can be seen, for WS and SFP2K, CFP increases the memory level parallelism over both CPR and ROB. For SINT2K and SERVER, the number of outstanding misses does not increase significantly. However, these suites still observe a performance gain from CFP (refer Figure 5) primarily because a large amount of independent work can be successfully completed (86% for SINT2K and 90% of the non-speculative instruction window for SERVER) for these suites even in the presence of isolated cache misses.

CFP can sustain a very large instruction window after a long-latency load miss because most mispredicted branches after the miss are independent of the miss and are therefore resolved quickly and in parallel with the miss. Only mispredicted branches in the window after the miss that depend upon the miss (and thus cannot be resolved until the miss completes) disrupt the usefulness of the instruction window. These branches are very infrequent (Table 4, Column 5).

To understand the amount of buffering for slices, we present statistics for the SDB. SDB occupancy information is presented in Table 4. The average SDB size is measured only during the presence of a long latency miss when the SDB is not empty. The average SDB size numbers indicate the SDB needs to sustain only



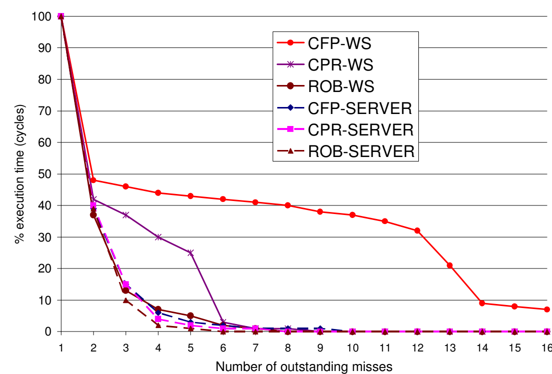
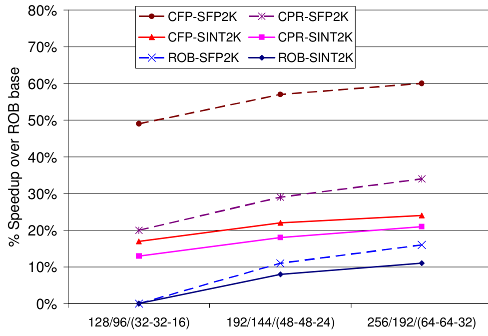


Figure 7 Memory level parallelism: Outstanding L2 miss distributions for CPR, CFP, and ROB. Higher curves do not imply more misses because these numbers are normalized to different execution times. Execution time for CFP is lowest.



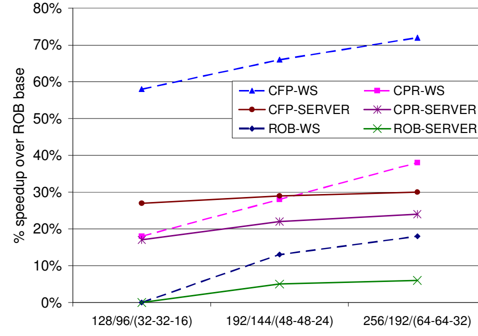


Figure 8 Equal cycle-critical resource comparison.

hundreds of instructions on average, to support an instruction window in the thousands. Further, only a small fraction of the total number of retired instructions (less than 10% on average) enter the SDB. Even though these slice instructions are small in number, they significantly limit performance in conventional ROB processors. SFP2K has a significant fraction (~16%) of retired instructions entering the SDB because of a large number of L2 cache misses. The small fraction of dependent instructions implies a small dense-SRAM SDB structure.

5.3 Implications on power

Since CFP executes past long latency operations and performs slice processing, power implications arise. If a branch dependent upon a long latency load is mispredicted, most of the execution in the shadow of the load may be wasted because it was on the mispredicted path. As a result the number of instructions executed on the wrong path increases with CFP. In the baseline CPR processor, 15% of instructions executed were misspeculated while with CFP, this number goes up to 30%. While this may not appear insignificant, this needs to be viewed keeping in mind the performance gains of CFP and the significant area and power savings in the cache hierarchy. Slice processing could also potentially increase power because of the additional work done in the SPU. However, since SPU activity is low (<10% of instructions enter the SPU), CFP compares favorably to conventional techniques requiring large highly active structures (such as scheduler and register file) to sustain a large instruction window.

Table 4 CFP/SDB statistics

Benchmark Suite	Avg. SDB size when occupied	% retired inst. into SDB	Branch misp. per 1000 uops	mpbrsdb[+] per 1000 shadow uops[*]	Avg. # of registers held by checkpoints	
					INT	FP
SFP2K	548	16.4%	0.64	< 0.001	33	28
SINT2K	190	0.8%	2.51	0.05	30	17
WEB	802	5.6%	1.66	0.11	30	17
MM	528	3.4%	2.49	0.02	29	21
PROD	374	1.1%	2.41	0.19	26	16
SERVER	339	4.7%	1.13	0.18	38	14
WS	736	7.1%	1.21	<0.001	24	16

[+]mpbrsdb: mispredicted branches in the SDB
*shadow uops: Total uops in the shadow of a miss

5.4 Equal cycle-critical resource comparison

To determine the resource efficiency of CFP, we compare the performance of CFP to a conventional ROB and a CPR processor using equal cycle-critical resources, over various configurations. For each configuration, the register file and scheduling window sizes remain the same for ROB, CPR, and CFP configurations. Figure 8 shows results for four benchmark suites. The *y*-axis is percent speedup over a 128-entry ROB baseline (the smallest ROB configuration in the figure). The *x*-axis label format *w/x/(i-f-m)* corresponds to a design for a *w*-entry reorder buffer, with *x* integer and *x* floating point registers, and distributed scheduler (*i*-entry Int., *f*-entry FP, and *m*-entry Mem.). The *w*-entry in the *x*-axis label is applicable only to the ROB-based machine. The CPR and CFP machines do not use a ROB and instead, use 8 map table checkpoints to achieve an adaptive instruction window. For space reasons, we present only the SINT2K, SFP2K, SERVER and WS benchmark suites; other suite results are similar.

Results show, for equal buffer sizes, a CFP processor outperforms both a ROB-based and a CPR processor. More interestingly, the 96/(32-32-16) CFP configuration significantly outperforms the 256/192/(64-64-32) ROB configuration for all benchmarks, by 28% on SFP2K, 20% on Server, 33% on WS and 5% on SINT2K.

To understand why a small register file is sufficient for CFP to outperform a much larger ROB configuration, Table 4 provides statistics for average number of registers held at any time by checkpoints. Under CFP, registers held by a checkpoint cannot be released until the checkpoint retires. The average number of physical registers tied down because of checkpoints is significantly lesser than total register file size. The total number of checkpoint registers tied down on average is smaller than the maximum such number (*i.e.*, *number of checkpoints X number of logical registers*) because often, the lifetimes of registers are long, and hence the same physical register belongs to multiple checkpoints.

6. CFP, RUNAHEAD, AND WIB

The inability of conventional processors to hide long memory latencies strongly ties in to the linear scaling of resources. This must occur to match demands placed by a very large instruction window, while maintaining high clock frequency and low design complexity. By providing mechanisms to process independent instructions continually in the presence of long memory latencies,

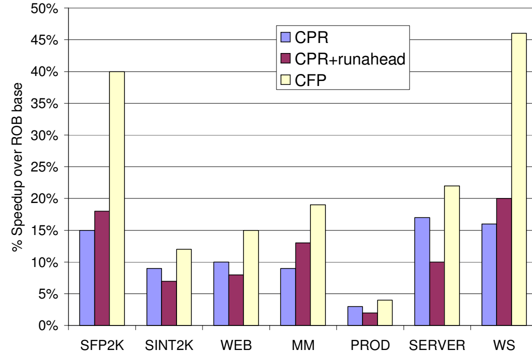


Figure 9 CFP, CPR and CPR+runahead.

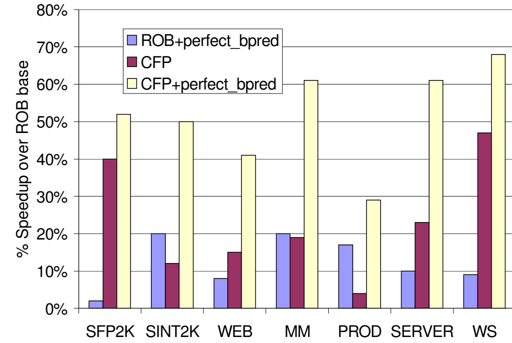


Figure 10 Performance trend with perfect branch prediction.

CFP can tolerate such latencies. We now compare CFP latency tolerance to two other proposals for memory latency tolerance: Runahead execution, and the Waiting Instruction Buffer.

6.1 CFP and Runahead

Runahead execution [11] in out-of-order processors has been proposed to tolerate memory latencies [18]. In runahead execution, the processor state is checkpointed at a long latency miss operation. Execution continues speculatively past the miss and prefetches data and possibly branch outcomes. When the miss returns, runahead execution terminates, the checkpoint is restored, and execution restarts from point of the load miss. Except for the prefetching effect of runahead, all work performed during runahead is discarded—all instructions past the load miss have to be re-fetched, and re-executed.

CFP subsumes runahead execution. Since CFP executes past long latency operations, it achieves benefits of runahead prefetch. Further, unlike runahead, CFP does not discard work. When the load data returns, the dependent instructions of the load complete execution, and allow the checkpoint to commit. Independent instructions past the miss have already completed and are retired without re-execution.

With increasing memory latencies, thousands of instructions could execute past a load miss. Not having to re-fetch and re-examine these instructions for committing them helps performance significantly. Further, runahead execution may result in cache pollution since sufficient cache capacity is required to hold prefetched instructions and data until they are re-examined and re-executed. CFP does not suffer from this limitation as re-examination or re-execution is not necessary.

Figure 9 shows the performance difference between CFP and CPR with runahead execution. The *y*-axis shows percent speedup over the baseline ROB configuration. Three bars are shown for each suite—CPR, CPR+runahead execution, and CFP. As can be seen, CFP outperforms runahead mode significantly for every benchmark suite (and not shown but also for every benchmark in the suites). CFP has significantly lower wasted speculative execution—only 30% for CFP compared to 70% additional uops for runahead execution.

6.2 CFP and WIB

The WIB [14] also drains load-miss-dependent instructions into a special buffer. However, significant differences exist between the WIB and CFP proposal. Two key differences are discussed below.

WIB requires a very large register file: CFP integrates a mechanism to achieve a non-blocking register file, while the WIB only provides a non-blocking scheduler. This is a substantial improvement over the WIB because it keeps the register file small, an important requirement for building memory-tolerant processors. Register files are very active, power-hungry cycle-critical structure and designers do not want these to be beyond a few hundreds.

WIB needs to allocate the entire window in its buffer. To recover from branch mispredictions, the WIB requires allocation of all instruction in the target window. This allows the WIB to record the program order of the slice, even though the WIB only actually stores the miss-dependent instructions. CFP does not suffer from such a restriction since it recovers by using a checkpoint.

The WIB proposal's assumption of very large register files, and a waiting buffer as large as the target window, results in it being inefficient in using resources.

7. IMPLICATIONS OF CFP

By allowing the instruction window to scale without requiring the cycle-critical register file and scheduler to scale, Continual Flow Pipelines have interesting implications for processor architecture directions. We discuss some of these implications below. By removing back-end pipeline stalls, CFP exposes branch prediction accuracy as the primary performance limiter. Section 7.1 discusses the interaction of CFP and branch prediction. High memory latency tolerance of CFP presents new opportunities for cache hierarchy organizations. Section 7.2 shows CFP with a small L2 cache outperforms a conventional ROB-based design with much larger caches. Multiple on-chip CFP cores can take better advantage of a given cache size as compared to multiple on-chip ROB-based cores, which would require the cache size to scale up. Sizing decisions for cycle-critical structures are simplified and discussed in Section 7.3.

7.1 Branch prediction remaining key limiter

Figure 10 shows performance potential with perfect branch prediction for a large instruction window processor (such as CFP) compared to a small instruction window processor (such as conventional ROB-based). The *y*-axis is percent speedup over the baseline ROB processor. Table 4 (Column 4) shows branch misprediction rates for the benchmarks. The key result is the difference in performance of perfect branch prediction for CFP as compared to ROB processors. The impact of branch prediction is substantially higher for CFP processors. In conventional ROB

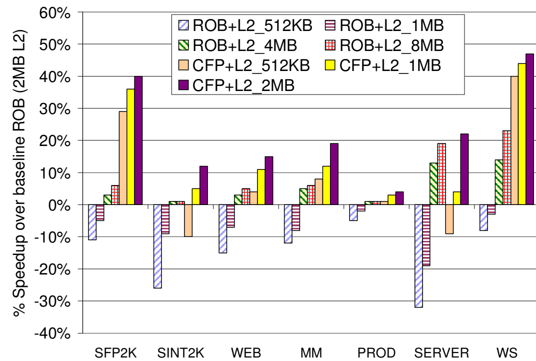


Figure 11 CFP vs. ROB for L2 cache sizes.

based processors, structural stalls in the presence of long latency operations hide branch misprediction impact. Thus, for ROB-based processors, performance gain due to perfect branch prediction is not substantial (less than 20% over a baseline ROB). However, for CFP, in the absence of such structural stalls, the gains are substantial even over a baseline CFP processor.

Since CFP allows instructions independent of long latency operations to execute and complete, any branches on this independent path also get resolved in parallel with the load miss. However, branches dependent on the long latency load cannot be resolved until the load data returns. Our experiments show that a very small fraction of mispredicted branches is dependent on a long-latency load miss.

Karkhanis and Smith [13] identified structural, data, and control-induced stalls as key performance limiters. While branch prediction accuracy is known to be an important performance issue, CFP processors effectively isolate the accuracy of branch prediction as the primary performance limiter, by eliminating structural and data stalls.

7.2 Increased cache efficiency and small dies

Caches account for a large fraction of the die size. Applications such as servers and workstations require very large caches, and for chip multiprocessors, pressure on the cache increases. Memory tolerance of CFP allows for high performance even with a small cache. Figure 11 shows the percent speedup (on the *y*-axis) over the ROB baseline (with a 2 MB L2 cache), as the L2 cache size is varied. For each suite, seven bars are shown—first four for a ROB-based processor, and the last three for a CFP processor.

As can be seen, CFP with a 2 MB cache outperforms ROB with an 8 MB cache. Further, a 512 KB L2 cache with CFP performs worse than a 2 MB L2 cache with a ROB for only two of the suites, primarily because a 512 KB cache results in thrashing of some benchmarks (*e.g.*, `vpr` and `twolf` for SINT2K) in these suites. However, CFP with 512 KB L2 cache still outperforms ROB processor with a 512 KB L2 cache. We assume all L2 caches have the latency of the baseline cache configuration. This results in a conservative performance estimate when we compare CFP configuration with a small L2 cache and a ROB configuration with a much larger L2 cache.

Achieving high performance with small cache sizes matches well current processor design goals of multiple cores on a chip. Since L2 caches occupy a significant fraction of die area, achieving comparable performance using much smaller caches allows multiple small cores to be placed on die, providing both high throughput and high single-thread performance. CFP thus forms an attractive building block core for future chip multiprocessors.

7.3 Simplified structure sizing

With CFP, the cycle-critical structures need to be designed only for a small active set of instructions. For example, we could start with the largest scheduler we could build. The L2 cache then is sized such that the scheduler can tolerate the L2 cache hit latency. Further, the register file can be sized to accommodate the needs of instructions in the scheduling window instead of the instruction window. Thus, instead of sizing key processor structures based on the target instruction-window size as is done in conventional processors, CFP allows these structures to be based on the much smaller scheduling window.

8. RELATED WORK

Section 3.3 discusses non-blocking schedulers [12, 14]. Various register file organizations have been proposed and include [3, 10]. The counter method used in CPR for reclaiming physical registers was first proposed [17] for a ROB-based processor. Virtual Physical Registers (VPR) [16] delay allocation of physical registers until just prior to instruction completion to reduce lifetimes of physical registers. They deal with dependent destination registers by not allocating destination physical registers to long latency operations and their dependent instructions until these operations are ready to execute. However, they do not reduce the lifetimes of completed source registers. Unlike VPR, CFP provides a mechanism to release both of the above types of registers.

In Dynamic Multithreading [1], instructions from the speculative thread execute in a multithreaded processor using data speculation, leave the pipeline freeing cycle-critical structures and wait in a separate buffer as large as the instruction window for subsequent validation. CFP does not require a multithreaded pipeline, and does not require a waiting buffer as large as the instruction window.

Proposals for resource efficient microarchitectures include Out-of-order commit processors [8] and Cherry [15]. Out-of-order commit processors [8] combine a checkpoint proposal [9] with the WIB [14] to address scheduler limitations for a checkpoint processor since the WIB focused on a ROB-based processor. Similar to the WIB, the paper also assumes a sufficiently sized register file. Cherry [15] uses the ROB and recycles physical registers and other resources once their associated instructions are branch-safe and memory-safe. Early resource reclamation is limited to a subset of the ROB. A checkpoint of the architected register file is used but only for recovering from exceptions and the ROB is used for retiring instructions.

Balasubramonian *et al*. [4] dynamically reserve physical registers for a future thread spawned when the main thread stalls due to a long latency operation. In addition to requiring partial support for two hardware contexts, partitioning resources between two threads prevents either thread from making full use of the machine's resources. The benefits of cache prefetching and branch computation are similar to that of Runahead execution.

Unlike CFP where execution continues in the presence of blocked operations (post-execution), thread-based pre-execution methods have been proposed where either auxiliary code [6, 22] or a small

subset of the program (*e.g.*, a backward slice of a cache miss) is pre-executed [19, 23] on idle contexts of a multithreaded processor prior to encountering the blocked operation.

In Datascalar architectures [5], multiple processors, each tightly coupled with part of the program's physical memory, asynchronously execute the same instructions on the same data, and the load results located in a processor's physical memory are broadcast to all other processors. This eliminates off-chip requests and reduces memory latency. However, such an approach is resource inefficient since multiple processors execute the same program.

Distributed large instruction window processing models have been proposed [20, 21]. These processing models significantly change the underlying processor and have different constraints and trade-offs over our conventional out-of-order processing model. CFP maintains a conventional processing model and is orthogonal to the above proposals.

9. CONCLUSIONS

Continual Flow Pipelines allow a processor core to sustain a very large and adaptive instruction window while keeping its scheduler and register file small. This exposes high ILP in the presence of long memory latencies. The memory latency tolerance results in the CFP core with a small L2 cache outperforming large ROB-based processors with very large caches. This improved cache efficiency, and resource decoupling has implications for future processor design. Look-ahead ability of CFP allows for high single thread performance in presence of long memory latencies, and its small resource core allows many of them to be placed on a single chip to address throughput-oriented applications.

Memory latency tolerance of CFP re-focuses the direction processor research must take to improve single thread performance. A large instruction window processor (such as CFP) now exposes branch prediction accuracy as the primary performance limiter for single threads. With Continual Flow Pipelines, the core pipeline does not stall due to resource limitations in the presence of long latency operations and only the rate at which the front-end feeds useful instructions to the back-end determines performance of such a pipeline. With structural and data limitations addressed (using the resource efficiency and memory latency tolerance of CFP), control flow now dominates performance limiting factors.

ACKNOWLEDGEMENTS

We thank Konrad Lai, John Shen, Jared Stark, and Chris Wilkerson for discussions and Saisanthosh Balakrishnan for comments on a draft of the paper. We also thank the reviewers for their constructive feedback.

REFERENCES

[1] H. Akkary and M. A. Driscoll. A Dynamic Multithreading Processor. In *Proceedings of the 31st International Symposium on Microarchitecture*, November 1998.

[2] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *Proceedings of the 36th International Symposium on Microarchitecture*, December 2003.

[3] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In *Proceedings of the 34th International Symposium on Microarchitecture*, December 2001, pp. 237--249.

[4] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi. Dynamically allocating processor resources between nearby and distant ILP. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, June 2001, pp. 26--37.

[5] D. Burger, S. Kaxiras, and J. R. Goodman. DataScalar Architectures. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997, pp. 338--349.

[6] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt. Simultaneous Subordinate Multithreading (SSMT). In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, May 1999.

[7] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, June 1998, pp. 142--153.

[8] A. Cristal, D. Ortega, J. Llosa, and M. Valero. Out-of-Order Commit Processors. In *Proceedings of the Tenth International Symposium on High-Performance Computer Architecture*, February 2004, pp. 48--59.

[9] A. Cristal, M. Valero, J.-L. Llosa, and A. Gonzalez. Large Virtual ROBs by Processor Checkpointing. *Technical Report UPC-DAC-2002-39, Universitat Politecnica de Catalunya*, July 2002.

[10] J.-L. Cruz, A. Gonzalez, M. Valero, and N. P. Topham. Multiple-banked register file architectures. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, June 200.

[11] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 1997 International Conference on Supercomputing*, 1997, pp. 68--75.

[12] 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*, February 2001.

[13] T. Karkhanis and J. E. Smith. A Day in the Life of a Data Cache Miss. In *Workshop on Memory Performance Issues*, June 2002.

[14] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A large, fast instruction window for tolerating cache misses. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002, pp. 59--70.

[15] J. F. Martinez, J. Renau, M. C. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors. In *Proceedings of the 35th International Symposium on Microarchitecture*, November 2002.

[16] T. Monreal, A. González, M. Valero, J. González, and V. Viñals. Dynamic Register Renaming Through Virtual-Physical Registers. In *Journal of Instruction Level Parallelism*, May 2000.

[17] M. Moudgill, K. Pingali, and S. Vassiliadis. Register Renaming and Dynamic Speculation: an alternative Approach. In *Proceedings of the 26th International Symposium on Microarchitecture*, December 1993.

[18] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors. In *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture*, February 2003.

[19] A. Roth and G. S. Sohi. Speculative Data-Driven Multi-Threading. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, January 2001.

[20] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, June 2003.

[21] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995, pp. 414--425.

[22] Y. Song and M. Dubois, Assisted Execution. University of Southern California, Technical Report #CENG 98-25, Department of EE-Systems, October 1998.

[23] C. B. Zilles and G. S. Sohi. Execution-based prediction using speculative slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, June 2001, pp. 2--13.