An Analysis of a Resource Efficient Checkpoint Architecture

HAITHAM AKKARY, RAVI RAJWAR, and SRIKANTH T. SRINIVASAN
Intel Corporation

Large instruction window processors achieve high performance by exposing large amounts of instruction level parallelism. However, accessing large hardware structures typically required to buffer and process such instruction window sizes significantly degrade the cycle time. This paper proposes a novel checkpoint processing and recovery (CPR) microarchitecture, and shows how to implement a large instruction window processor without requiring large structures thus permitting a high clock frequency.

We focus on four critical aspects of a microarchitecture: (1) scheduling instructions, (2) recovering from branch mispredicts, (3) buffering a large number of stores and forwarding data from stores to any dependent load, and (4) reclaiming physical registers. While scheduling window size is important, we show the performance of large instruction windows to be more sensitive to the other three design issues. Our CPR proposal incorporates novel microarchitectural schemes for addressing these design issues—a selective checkpoint mechanism for recovering from mispredicts, a hierarchical store queue organization for fast store-load forwarding, and an effective algorithm for aggressive physical register reclamation. Our proposals allow a processor to realize performance gains due to instruction windows of thousands of instructions without requiring large cycle-critical hardware structures.

Categories and Subject Descriptors: C.1 [**Processor Architectures**]:

General Terms: Computer systems

Additional Key Words and Phrases: Computer architecture, scalable architecture, checkpoint architecture, high-performance computing

1. INTRODUCTION

Achieving high performance in modern microprocessors requires a combination of exposing a large amount of instruction level parallelism (ILP) and processing instructions at a high clock frequency. Exposing maximum ILP in the presence of long latency operations such as load misses to memory, dependent chains of floating point operations and so on requires the processor to concurrently operate upon a large number of instructions, also known as an instruction

Authors' address: Intel Microarchitecture Research Lab (MRL), JF3-332, 2111 NE 25th Avenue, Hillsboro, OR 97124; email: {haitham.h.akkary;ravi.rajwar;srikanth.t.srinivasan}@intel.com.

window. An instruction window is defined as consisting of instructions renamed but not yet retired. For example, in reorder buffer based processors, every instruction that has a reorder buffer entry allocated is considered part of the instruction window. Hardware structures to buffer all instructions in a conventional, large instruction window must be sufficiently large. However, high clock frequencies require frequently accessed structures to be small and fast. With increasing clock frequencies, new designs that do not require large cycle-critical hardware structures become necessary for building large instruction window processors. This paper presents a detailed study of performance issues related to large instruction window processors and presents a novel and efficient checkpoint processing and recovery (CPR) microarchitecture for such processors.

Checkpoints in processors are typically used to repair architectural state to a known previous state. The use of checkpoints for out-of-order processors in the context of recovering from branch mispredictions and exceptions was proposed by Hwu and Patt [1987]. Some processors have since used checkpoints to recover architectural register state in the event of branch mispredictions [Leibholz and Razdan 1997; Yeager 1996]. Checkpoints were first used for tolerating long memory latencies in the virtual ROB proposal [Cristal et al. 2002].

In this paper, we analyze our checkpoint architecture proposal, checkpoint processing and recovery [Akkary et al. 2003], for sustaining a large number of in-flight instructions in a resource-efficient manner. Building a scalable checkpoint-based processor requires addressing various aspects such as register files, store queues and so on. This paper shows how to implement large instruction-window checkpoint architectures without requiring key cycle-critical structures to scale.

Four key aspects of a microprocessor critically affected by the demands placed by a large instruction window and the need for a high clock frequency are (1) the scheduling window, (2) branch misprediction recovery mechanism, (3) the store queue, and (4) the physical register file. The mechanisms, size, and access latency of structures associated with these aspects are potential key parameters in achieving high performance. In Section 3 we establish the effect on performance due to these aspects.

The scheduling window consists of instructions renamed but not yet issued to the execution units, and is examined each cycle to find instructions for issue. In Section 3.1, we show that, while scheduling windows are important, their size is not the most critical issue when building a high-performance large instruction window processor. The scheduling window size in an out-of-order processor needs to increase from current sizes only in the presence of long latency operations. Since only a small number of instructions are dependent on long latency operations and block the scheduler, the scheduling window need not to scale with instruction window size. This was originally observed by Karkhanis and Smith [2002] and verified for a large set of benchmarks in Section 5.4.

Branch mispredictions expose the long latency associated with high-frequency deep pipelines and are the single largest contributor to performance degradation as pipelines are stretched [Sprangle and Carmean 2002]. Branch

misprediction recovery requires redirecting fetch to the correct instruction and restoring the rename map table before new instructions can be renamed. The map table can be restored from a checkpoint [Leibholz and Razdan 1997; Yeager 1996], incrementally restored from a nonspeculative map table such as the retirement register alias table [Hinton et al. 2001], or incrementally restored from a history buffer that stores the speculative map table updates performed since the mispredicted branch was dispatched. In Section 3.2, we show practical implementations of these traditional recovery mechanisms to be either too costly or too slow, and in Section 4.1 we present a new policy of checkpointing the map table at low-confidence branches. This forms the basis of the CPR microarchitecture and allows a small number of checkpoints to be sufficient for a large instruction window.

The store queue is a critical component of out-of-order processors. Store queues serve three primary functions: disambiguating memory addresses, buffering stores (completed and otherwise) until retirement, and forwarding data to dependent load operations. The last operation, called store-to-load forwarding, directly impacts cycle time. Since a load may depend upon any store in the queue and multiple stores to the same address may simultaneously be present, the circuit to identify and forward data is complex and incurs long delays as store queue size increases. The store queue must provide the dependent load with data within the data cache access time to avoid complications of scheduling loads with variable latencies. Designing store queues with sizes much larger than currently feasible (Pentium® 4 has a 24 entry store queue) using conventional methods are highly unlikely, without making the forwarding circuit a critical path, thus increasing cycle time. We study the performance impact of the store queue in Section 3.3, and in Section 4.2 we propose a hierarchical store queue organization.

The physical register file (referred to from now on as register file) increases ILP by removing write-after-write and write-after-read dependences. Current mechanisms for allocating and freeing physical registers result in the lifetime of a physical register exceeding the lifetime of the allocating instruction. This requires the register file size to be of the same order as the instruction window size as shown in Section 3.4. For large instruction windows, naively increasing the register file size increases the register access time. Multicycle register file accesses degrade performance and also increases branch misprediction penalty. While different register file organizations have been proposed [Balasubramonian et al. 2001; Capitanio et al. 1992; Cruz et al. 2000], these proposals do not change the large register file requirement and hence can adversely impact area size and power. In this work, we demonstrate that register files need not be scaled with large instruction windows if an aggressive register reclamation policy such as the one proposed by Moudgill et al. [1993] is used. An aggressive register reclamation policy enables a register file to perform comparable to a larger conventional register file by significantly reducing the average lifetime of physical registers. In Section 4.3, we show how an aggressive register reclamation algorithm can be adapted to a checkpoint architecture.

Paper contributions: The paper makes the following contributions in analysis and design of high-performance large instruction window processors.

—*CPR: A new resource-efficient microarchitecture.* Our CPR microarchitecture significantly outperforms a ROB-based design with identical buffer resources, even for small window sizes. We argue that while the ROB itself can be made larger, the mechanisms associated with the ROB inhibit performance and need to be implemented differently. By off-loading all functionality of a ROB to other scalable mechanisms, we move away from a centralized, ROB-based processor design.

—*Confidence-based checkpoints.* We show branch recovery mechanisms to be critical for high performance and show a reorder buffer (ROB)-based recovery mechanism to be a performance limiter. Instead of using a ROB, we propose selectively creating checkpoints at low-confidence branches. Our selective checkpointing mechanism enables fast branch misprediction recovery and minimizes checkpoint overhead. We show eight such checkpoints are sufficient to achieve most of the performance of a 2048-entry instruction window processor with an ideal branch misprediction recovery mechanism.

—*Hierarchical store queues.* Our novel hierarchical store queue organization can buffer a large number of stores and can perform critical store-to-load forwarding without degrading cycle time.

—*Bulk retirement.* While not a key performance aspect in itself, we break the limit of in-order serialized retirement imposed by reorder buffers by providing the ability to retire hundreds of instructions per cycle.

The paper is organized as follows. We outline our simulation methodology in Section 2 and present a limit study analysis to identify key performance issues in Section 3. Section 4 presents and evaluates individual solutions to the key issues. Section 5 puts the individual solutions together into a single microarchitecture and evaluates the new CPR microarchitecture. Related work is discussed in Section 6, and we conclude in Section 7.

2. SIMULATION METHODOLOGY

We use a detailed execution-driven simulator working on top of a micro-operation (uop) level IA32 functional simulator for executing long instruction traces (LITs). A LIT is a snapshot of the processor architectural state including memory and is used to initialize the execution-driven performance simulator. A LIT includes all system interrupts needed to simulate system events such as DMA traffic and so on. The simulator executes both user and kernel instructions. Our baseline processor is based on a Pentium® 4 and, the parameters are shown in Table I. A detailed memory subsystem is also modeled. The simulated benchmark suite is listed in Table II. All performance numbers in this paper are reported as normalized micro-operations per cycle (uPC on the y-axis) where the normalization is with respect to the performance of the baseline parameters of Table I and marked as (base) in the graphs.

Table I. Baseline Processor Parameters

Processor frequency	3.8 GHz
Rename/issue/retire width	3/5/3
Branch mispred. penalty	30 cycles
Instruction window size	128
Scheduling window size	128
Register file	96 integer, 96 floating point
Load/store buffer sizes	48/32
Functional units	Pentium® 4 equivalent
Branch predictor	combining (64 K gshare, 16 K bimod)
Hardware data prefetcher	Stream based (16 streams)
Trace cache	32 K-uops, 8-way
L1 Data cache	16 KB, 4 cycle hit, 64 byte line
L2 Unified cache	1 MB, 8-way, 16 cycle hit, 64-byte line
L1/L2 Line size	64 bytes
Memory latency (load to use)	100 ns

Table II. Simulated Benchmark Suites

Suite	No. of Benchmarks	Descriptions/Examples
SPECFP2K (SFP2 K)	14	http://www.spec.org
SPECINT2K (SINT2 K)	12	http://www.spec.org
Internet (WEB)	12	SPECjbb, WebMark
Multimedia (MM)	10	MPEG, speech recog., photoshop
Productivity (PROD)	13	SYSmark2k, Winstone,
Server (SERVER)	4	TPC-C
Workstation (WS)	14	CAD, rendering

3. A LIMIT STUDY AND PERFORMANCE ANALYSIS

In this section, we analyze the following four key aspects affected by large instruction windows in detail: the scheduling window, branch misprediction recovery mechanism, the store queue, and the register file.

To bound performance gain due to large instruction windows, we first perform a limit study. In this study, the four key aspects we identified earlier are idealized and the instruction window size is varied from 128 up to 2048. In other words, for an instruction window size of 1024, the scheduling window size is 1024, the store queue and register file are sized ideally, and the rename map table is available instantaneously for branch misprediction recovery. For the studies in this section, we assume perfect memory disambiguation. All other parameters of the processor are similar to the baseline microarchitecture.

Figure 1 shows the performance variation with increasing instruction window sizes. The label *iwN* corresponds to an instruction window of size *N*. As seen from the graph, significant performance can be obtained with increasing instruction windows, up to 55% for 2048 entry instruction windows, if we idealize certain aspects of the processor.

To understand the sensitivity to performance of the individual key aspects mentioned earlier, we conduct further experiments. For each experiment, we vary the aspect parameter under study and idealize the other key aspects. Thus,

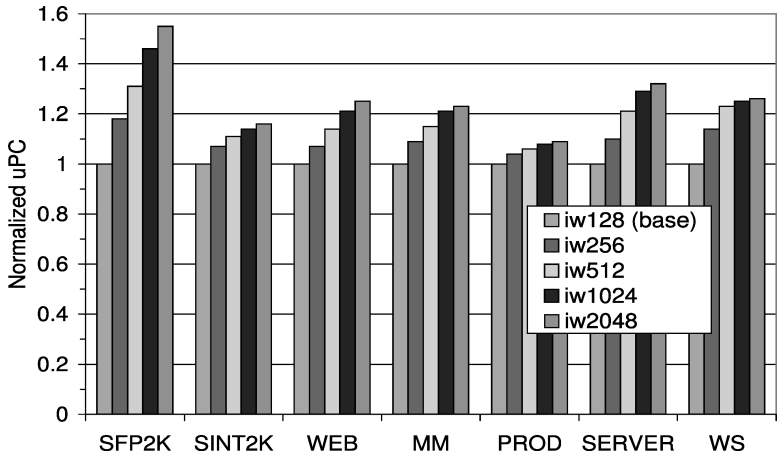

Fig. 1. Impact of instruction window size.

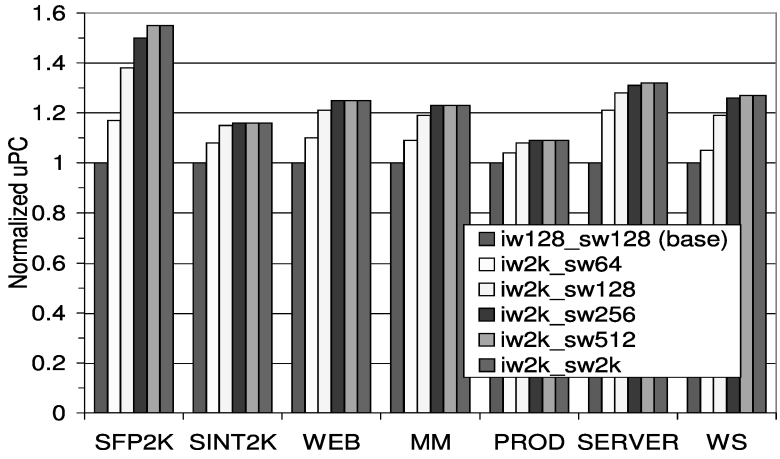

Fig. 2. Impact of scheduling window size.

three of the four parameters are kept idealized while one parameter is varied. This allows us to study one aspect in isolation without interference from other key parameters.

3.1 Impact of Scheduling Window Size

Figure 2 shows the impact of the scheduling window size (64 up to 2048) for a 2048-entry instruction window. The label *iwN_swM* corresponds to an instruction window of size *N* and a scheduling window of size *M*. We model an ideal store queue, a sufficiently large register file, and ideal misprediction recovery mechanism. As we can see, a 256-entry scheduling window achieves nearly the same performance as a 2048-entry scheduling window for an instruction window size of 2048. For most benchmarks, even a 128-entry scheduling window achieves a significant percentage of the ideal performance.

An out-of-order processor's scheduler fills and blocks only in the presence of long latency operations—primarily load misses to main memory. Since the scheduler can continue to issue independent instructions, only instructions dependent on the load miss occupy scheduler entries for a long period of time. Our results from Figure 2 show a relatively small scheduling window that is sufficient to support a large instruction window. This suggests only a small number of instructions in a large instruction window are waiting for cache miss data at any given time. Similar results have been reported for the integer benchmarks in another earlier study [Karkhanis and Smith 2002]. We verify these results for a large set of benchmarks and also show the scheduling window in the absence of cache misses needs to be only a small fraction of the instruction window in Section 5.4. Hence, we believe, while we need to investigate means of building a 128 to 256-entry scheduler, building bigger schedulers is not necessary to fully exploit large instruction windows, and assume a 128-entry scheduler for the rest of the paper, unless otherwise specified.

3.2 Impact of Misprediction Recovery Mechanism

Two main contributors to performance degradation due to branch mispredictions are cycles to resolve a branch, and cycles to recover after the branch misprediction is resolved. In this paper, we focus on misprediction recovery. Misprediction recovery involves restarting fetch and renaming instructions from the correct path. Fetch from the correct path may restart immediately after the branch misprediction is resolved. However, the correct path instructions cannot be renamed until the rename map table corresponding to the mispredicted branch is restored.

Common methods for restoring the map table include

(1) *Using map table checkpoints*. Map table checkpoints are created periodically either at every branch or every few cycles [Leibholz and Razdan 1997; Yeager 1996]. On a misprediction, the checkpoint corresponding to the mispredicted branch is restored. The number of checkpoints limits the number of unresolved branches allowed in the instruction window.

(2) *Using the retirement map table (RMAP)*. In this scheme, a retirement map table [Hinton et al. 2001] is used in addition to the frontend map table. Each ROB entry also has the rename map for its corresponding instruction. Once a misprediction is resolved, the mispredicted branch is allowed to reach the head of the ROB at which time the retirement map table will have the correct map table corresponding to the mispredicted branch. At this point, the retirement map table is copied to the frontend map table, after which renaming can start. Since all instructions prior to the mispredicted branch must be retired before renaming can start, this scheme can lead to significant delays if long latency operations prior to the mispredicted branch stall retirement.

(3) *Using the retirement map table and the ROB (RMAP+WALK)*. This scheme is an optimization on the scheme above. Instead of waiting for the mispredicted branch to reach the head of the ROB, we start with the current

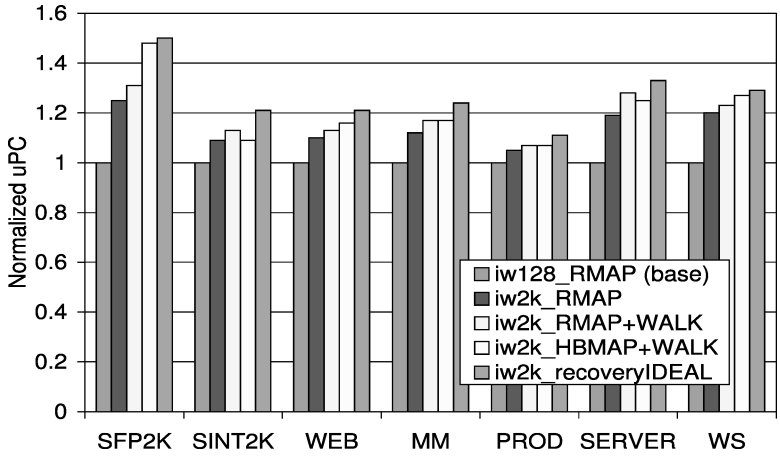

Fig. 3. Impact of misprediction recovery mechanism.

retirement map table and pro-actively walk from the head of the ROB toward the mispredicted branch, incorporating the rename information of each ROB entry. This allows renaming of correct path instructions to commence without waiting for all instructions prior to the mispredicted branch to retire.

(4) *Using the frontend map table and a history buffer (HBMAP+WALK)*. In this scheme, a history buffer is used to store overwritten maps of each instruction. On a branch misprediction, we start with the current frontend map table. We pro-actively walk from the current tail of the ROB (i.e., the most recently allocated instruction) toward the mispredicted branch, incorporating the overwritten maps of each instruction. Depending on whether the mispredicted branch is closer to the ROB head or ROB tail, RMAP + WALK, or HBMAP + WALK will perform better.

The periodic checkpoint method as described above, while quick, is impractical to implement and resource inefficient because hundreds or thousands of checkpoints may be required as instruction window sizes scale to the thousands. Further, only having a few checkpoints made at conditional branches performed worse than the other schemes we discuss. Hence we do not present results for the periodic checkpoint scheme. Sequentially restoring the map table, while implementable, could contribute to a significant increase in branch misprediction penalty since many instructions may be in the ROB or history buffer prior to the mispredicted branch and need to be serially processed. As instruction window size increases, the above methods are either costly or may become too slow.

We evaluate the three sequential map table restoration schemes for large instruction windows and compare them to an ideal misprediction recovery scheme. The scheduling window, register file, and store queue sizes are idealized. Figure 3 presents the results. No single scheme works for all workloads, and all show significant performance reduction compared to the ideal mechanism. The history buffer method (HBMAP+WALK) performs the best.

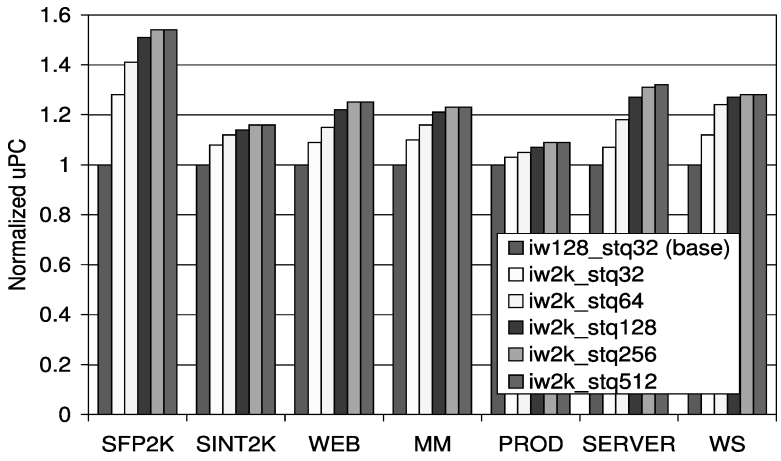

Fig. 4. Impact of store queue size.

Nevertheless, on benchmarks (e.g., SPECINT 2000) with frequent branch mispredictions the scheme suffers an 11% reduction in performance relative to the ideal model. The above results indicate the importance of having efficient misprediction recovery mechanisms to exploit performance of large instruction window processors.

3.3 Impact of the Store Queue

Large instruction windows place high pressure on the store queue because the store queue size is directly proportional to the number of stores in the instruction window. The store queue is a cycle-critical component of modern processors because it often needs to provide data to dependent load instructions in the same time it takes to access the data cache. A store queue with an access time larger than the data cache hit latency requires the scheduler to deal with an additional load latency and predict if a load hits the store queue or not. Mechanisms to recover from scheduler mispredicts introduce tremendous complexity to time-critical scheduling logic. Further, stores may stay in the store queue for long durations because, traditionally, stores are retired in order to allow for misprediction recovery, precise exceptions, and memory consistency requirements. A large fraction of loads (26% to 42%) hit stores in the store queue. Hence, store-to-load forwarding is critical to achieving high performance because it prevents dependent loads from stalling.

Figure 4 shows the impact on performance in a 2048-entry instruction window model as the store queue size varies from 32 to 512 entries. The label *iwN_stqM* corresponds to an instruction window of size *N* and a store queue of size *M*. The scheduling window size, misprediction recovery mechanism, and register file are idealized. A store queue size of at least 128 entries is required to achieve performance close to the ideal 2048-entry instruction window. Such size is a significant increase above current sizes (24–32) and will be quite a designing challenge in a naively scaled up implementation of a large instruction window processor.

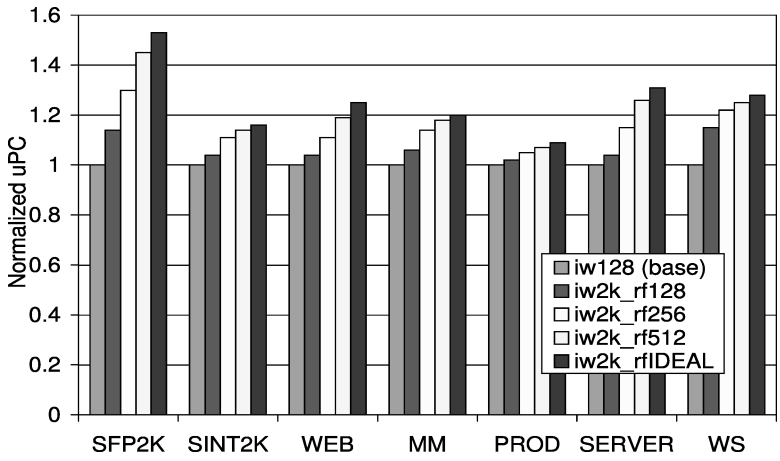

Fig. 5. Impact of register file size.

3.4 Impact of the Register File Size

Instructions with a destination register operand require a physical register to be allocated. In conventional processors, a physical register is allocated at the time the corresponding instruction is renamed, and is released when a subsequent instruction that overwrites the physical register's corresponding logical register is retired. Thus, register reclamation is tied to in-order instruction retirement and the lifetime of a physical register exceeds the lifetime of its allocating instruction. Since most instructions have a destination register operand, the register file size must scale with the instruction window size as shown in Figure 5. The label *iwN_rfM* corresponds to an instruction window of size N and a physical register file with M floating point and M integer registers.

Thus, large instruction windows place tremendous pressure on the register file. The register file is typically a highly ported structure. Building large highly ported register files to accommodate the renaming demands of large windows is difficult, introduces complexity, and increases cycle time. Therefore, we must investigate alternative mechanisms to design register files for large instruction window processors.

4. IMPLEMENTING LARGE INSTRUCTION WINDOWS

In the earlier section, we have shown performance of large instruction window processors to be most sensitive to the branch misprediction recovery mechanism and the size of the store queue and register file. In this section we address these three critical aspects. First, we present a new approach to recovering the rename map table in Section 4.1. Then, in Section 4.2, we present and evaluate a new store queue organization. Finally, in Section 4.3 we discuss an aggressive reclamation mechanism for the register file. These three new proposals form the key components for the CPR microarchitecture discussed in Section 5.

4.1 Selective Low-Confidence Branch Checkpoints

The misprediction recovery mechanism must be fast and have low overhead. As discussed in Section 3.2, except for the checkpoint mechanism, the other schemes recover maps serially. Thus, for high performance, the checkpoint mechanism is preferred. Our mechanism for recovering rename maps also uses checkpoints of the map table. However, we limit the number of such checkpoints to carefully selected points in the instruction window. Ideally, we would like to create checkpoints exactly on mispredicted branches. Hence, unlike earlier proposals where map table checkpoints are created either at every branch or every few instructions, we create map table checkpoints at branches with a high misprediction probability, selected using a confidence estimation scheme.

We rely on the same checkpoints created to handle branch misprediction recovery to implement precise interrupts and exceptions and to deal with architecture-specific serializing instructions. We now discuss the conditions under which checkpoints are created and then discuss checkpoint buffer management policies.

4.1.1 *Checkpoint Creation.* Since checkpoints are not created at every branch, a branch misprediction may result in execution restarting from the nearest checkpoint prior to the mispredicted branch. This causes the good instructions between the checkpoint instruction and the mispredicted branch to be re-executed. We call this re-execution overhead the checkpoint overhead (COVHD). A branch confidence estimator [Jacobsen et al. 1996] is used to minimize this overhead. The estimator uses a table of 4-bit saturating counters indexed using an *xor* of the branch address and global branch history. A correct prediction increments the counter, and a misprediction resets the counter to zero. A counter value of 15 signals high confidence while the remaining values signal low confidence. To minimize COVHD, in addition to creating checkpoints at low-confidence branches, a checkpoint is also created every 256 instructions.

While a checkpoint is made at low-confidence branches, a noncheckpointed branch may be mispredicted, forcing a recovery to a prior checkpoint. To prevent the same branch from mispredicting again and thus degrading performance, on a re-execution from a checkpoint, we use the branch outcome from the previous aborted execution itself rather than a prediction. This is done by storing the branch distance (in number of branches) from the checkpoint and the associated branch outcome. Furthermore, to guarantee forward progress, once a branch misprediction is resolved and re-execution begins from a prior checkpoint (C1), we force a new checkpoint (C2) at the first branch (irrespective of whether it is a low-confidence branch or not). This allows instructions between checkpoints C1 and C2 to be retired even in the pathological case where multiple branches get alternatively mispredicted. To handle other events such as exceptions and memory consistency events such as snoop invalidations, we use a similar mechanism and allow a checkpoint to be forced even on the very next instruction after a prior checkpoint.

4.1.2 *Checkpoint Buffer Management.* The checkpoint buffer keeps track of map table checkpoints. Checkpoints are allocated and reclaimed in a

first-in-first-out order. Each checkpoint buffer entry has a counter to determine when the corresponding checkpoint can be freed. The counter tracks completion of instructions associated with the checkpoint—the counter is incremented when an instruction is allocated and decremented when the instruction completes execution. Counter overflow is prevented by forcing a new checkpoint.

A checkpoint is allocated only if a free checkpoint is available. If a low confidence branch is fetched and a free checkpoint is not available, the processor ignores the low-confidence prediction of the branch and continues fetch, dispatch, and execution without creating any additional checkpoints as long as the last checkpoint's counter does not overflow. We find that not stalling on a checkpoint buffer full condition is important for high performance.

The oldest checkpoint is reclaimed when its associated counter has a value of 0 and the next checkpoint has been allocated. This means all instructions associated with the older checkpoint have been allocated and have completed execution.

Each instruction has an identifier associating it with a specific checkpoint. Instructions use this identifier to access the appropriate checkpoint buffer for incrementing and decrementing the counter. This identifier is also used to select instructions to be squashed or committed. As soon as the last instruction belonging to a checkpoint completes, all instructions in that checkpoint can be retired instantly and the associated checkpoint is reclaimed. This provides the ability to commit hundreds of instructions instantly thereby potentially removing the in-order retirement constraints enforced by the ROB. We will discuss this more in Section 5.2.

In our proposal, rather than using the ROB, we use a substantially smaller checkpoint buffer for misprediction recovery. Importantly, the size of the instruction window is not necessarily limited by the checkpoint buffer size because each checkpoint may correspond to a large number of instructions, and their numbers can vary across checkpoints. For example, if a checkpoint entry on the average corresponds to 300 instructions, eight checkpoints would be sufficient to support a 2048-entry instruction window.

4.1.3 *Selective Checkpointing Results.* Figure 6 compares four methods for restoring the map table against the ideal scheme. The scheduling window, store queue, and register file are sized ideally. Three of the methods are from Section 3.2 and the fourth is our selective checkpoint proposal. Our proposal uses eight checkpoints and employs a branch confidence estimator for deciding when to create a checkpoint. As can be seen from the graph, our proposal performs the best on average across all benchmarks, with the history buffer method coming second. Further, our proposal performs within 7% of ideal.

Table III presents key metrics to help better understand the behavior of our selective checkpoint scheme. CCOV denotes the average percentage of mispredicted branches that are predicted to be low confidence and checkpointed, while COVHD denotes the average number of good instructions re-executed because of rolling back to a prior checkpoint. We achieve a high CCOV and a very low COVHD because of the high mispredicted branch coverage achieved by the low-confidence estimator we use.

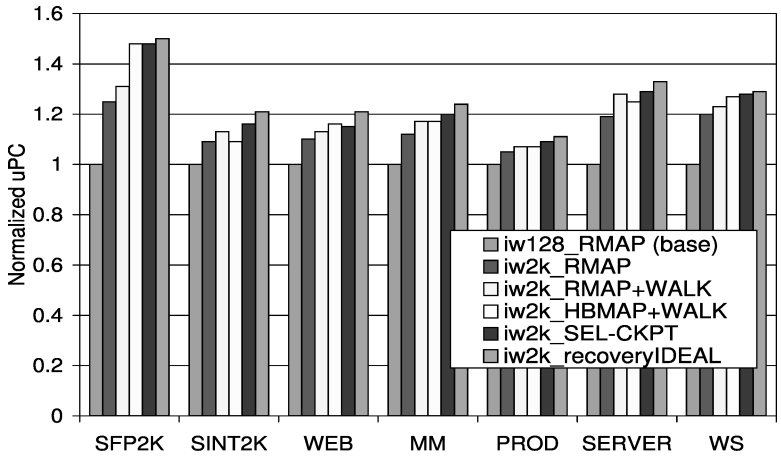

Fig. 6. Selective checkpoint performance.

Table III. Selective Checkpoint Related Statistics

Bench.	Misprediction Distance (MPD)	Checkpoint Coverage (CCOV) (%)	Checkpoint Overhead (COVHD)	
			per MP.	of Insn (%)
SFP2K	1265	75	20	1.6
SINT2K	312	81	12	3.8
WEB	575	73	31	5.4
MM	320	74	25	7.8
PROD	444	84	9	2
SERVER	478	74	15	3.1
WS	568	70	32	5.6

Selective checkpoints using a low-confidence estimator enable the processor's instruction window to adapt to an application's frequency of branch mispredictions. Consider SFP2K and SINT2k benchmark suites. The average distance between branch mispredictions (MPD) is significantly larger for SFP2K than SINT2K (1265 uops vs. 312 uops), while CCOV is about the same. The ratio of correctly predicted branches to mispredicted branches that are assigned low confidence by the confidence estimator is about 4 to 1 in general for all benchmark suites. Hence, the average distance between checkpoints is about 253 uops (1265/5) for SFP2K and 62 uops (or 312/5) for SINT2K. Using eight checkpoints, we can achieve close to a 2048-entry (253*8) and a 512-entry (62*8) instruction window for SFP2K and SINT2K, respectively. This is optimal since large instruction windows are less beneficial for performance when it is highly unlikely that fetch can proceed along the correct path for long due to frequent branch mispredictions.

4.2 Hierarchical Store Queue Organization

A store queue must have the capacity to buffer all stores within a large instruction window, typically on the order of hundreds and, more importantly, must forward data to any dependent load in the same time as the first-level

data cache hit latency. We propose and evaluate a hierarchical store queue: a fast and small first-level store queue backed by a much larger and slower second-level store queue. Since stores typically forward to nearby loads, most store-to-load forwarding occurs from the first-level store queue. Thus, the hierarchical organization works well.

4.2.1 *Level One Store Queue.* The fast-level one store queue (L1 STQ) is a small n-entry buffer holding the last n stores in the instruction window. This buffer is similar to store queues in current processors and is designed as a circular buffer with head and tail pointers. When a new store is inserted into the instruction window, an entry is allocated for the store at the tail of the L1 STQ. When a conventional store queue is full, instruction allocation stalls. However, when the L1 STQ is full, the oldest store is removed from the head of the queue to make space for the new store, and is moved into the backing level two store queue (L2 STQ). The L1 STQ has the necessary address matching and store select circuit to forward data to any dependent loads.

4.2.2 *Level Two Store Queue.* The level two store queue (L2 STQ) is much larger and slower than the L1 STQ and accepts stores forced out from the L1 STQ. Stores remain in the L2 STQ until retirement. In addition, the L2 STQ has a membership test buffer (MTB) associated with it. The MTB is similar to a Bloom filter [Bloom 1970] and aids in quickly determining whether a given load address matches a store entry in the L2 STQ.

The MTB is a direct-mapped nontagged array of counters indexed by a part of a load or store address. When a store is removed from the L1 STQ and is placed into the L2 STQ, the corresponding untagged MTB entry is incremented. When a store retires, updates the data cache and is removed from the L2 STQ, the corresponding untagged MTB entry is decremented. A nonzero count in the MTB entry potentially points to a matching store in the L2 STQ. On the other hand, a zero count in the MTB entry guarantees a matching store that does not exist in the L2 STQ. While tagging the MTB will prevent false matches, a nontagged and direct-mapped design makes the MTB access possible under the time to access the data cache and the L1 STQ—a critical requirement to prevent complexity in the scheduler. The MTB also contains the count of store entries in the L2 STQ with unknown address. The counter is used to quickly determine if there is a potential load conflict with unknown-address stores in the L2 STQ.

4.2.3 *Hierarchical Store Queue Design.* Figure 7 shows the hierarchical two level store queue organization. When a load is issued, and while the data cache is being read, the L1 STQ and the MTB are also accessed in parallel. If the load hits the L1 STQ, the store data is forwarded to the load. If the load misses the L1 STQ, and the MTB entry is zero (i.e., the L2 STQ also does not have a matching address), the data is forwarded to the load from the data cache. If the load misses the L1 STQ and the MTB indicates a potential match in the L2 STQ (i.e., the MTB entry is nonzero), the load is penalized a data cache miss penalty to allow sufficient time to access the L2 STQ and resolve the load-store dependency. If the load hits the L2 STQ, data is supplied to the load from the

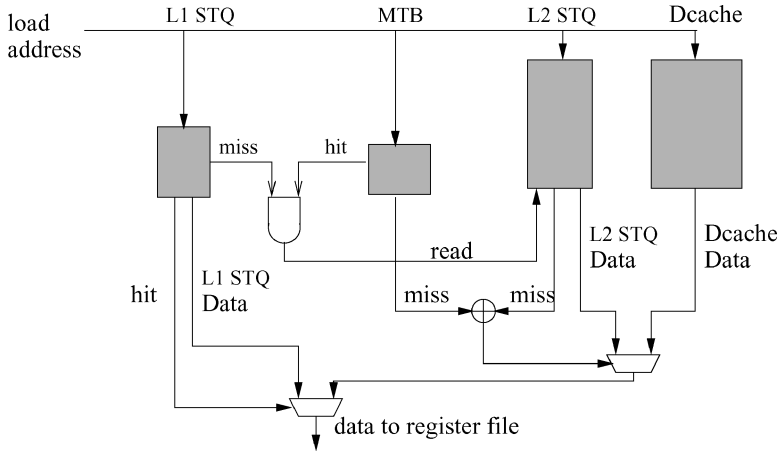

Fig. 7. Two-level hierarchical store algorithm.

L2 STQ, else the data is forwarded to the load from the data cache. Since the MTB is not tagged, a nonzero MTB entry count does not necessarily guarantee a matching store in the L2 STQ. However, the load has already suffered a delay equivalent to a data cache miss. Spurious hits in the MTB due to address aliasing therefore must be minimized by making the MTB as large as it can be, while keeping it accessible within the time to access the L1 STQ and the data cache.

4.2.4 *Memory Disambiguation.* Until now, we have assumed perfect memory disambiguation for our studies. Store-load dependences are highly predictable [Moshovos and Sohi 1997]. Hence, we use a memory-dependence predictor and let a load proceed if the predictor indicates that the load has no conflict with an unknown store address. On a store-load dependence violation, the processor needs to rollback to a prior checkpoint. Hence, our memory-dependence predictor focuses on minimizing load-store dependence violations and not necessarily achieving the best prediction accuracy. The predictor is based on the notion of a *store-distance* of a load computed as the number of store queue entries between the load and its forwarding store. To reduce aliasing and allow forwarding from different instances of the same store at varying distances from the load, up to four such distances are stored in a nontagged array indexed by the load instruction address. A load is stalled if the distance from a load to a current unresolved store address matches a distance value stored in the array.

We detect memory-dependence violations by letting stores snoop a set-associative load buffer. Because the load buffer is not on the critical path and does not forward any data, it is not a critical resource, so we do not focus on load buffers.

4.2.5 *Hierarchical Store Queue Performance.* Figure 8 shows the performance of the hierarchical store queue proposal, compared to the baseline and to an ideal 2048-entry instruction window. The scheduling window size, branch

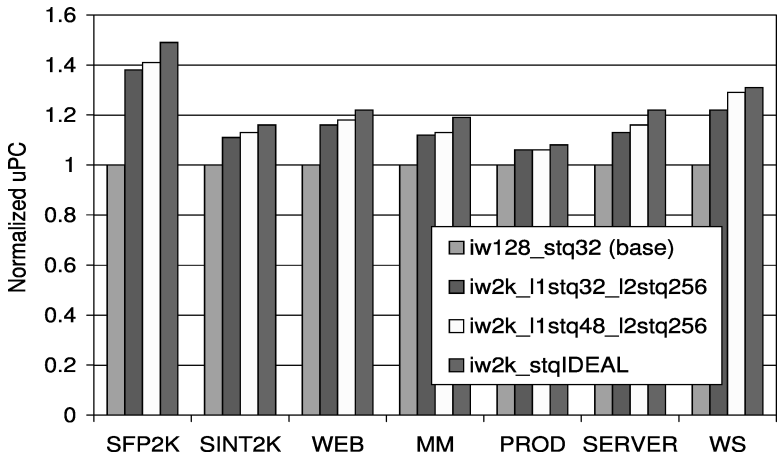

Fig. 8. Hierarchical store queue performance.

Table IV. Store-Load Forwarding Characteristics

	Percentage of all Loads	Percentage of Forwarded Loads That Hit	Percentage of all Loads That	
Bench.	Store-Forwarded	in L2 STQ	False Hit in MTB	Search L2 STQ
SFP2K	26	18	2	7
SINT2K	31	20	2	8
WEB	35	11	2	6
MM	27	13	2	6
PROD	42	13	2	9
SERVER	39	14	4	9
WS	34	17	3	9

recovery, and register file size are idealized. We show two L1 STQ configurations with 32 and 48 entries. L2 STQ and MTB are each 256 entries for both configurations (aliasing still occurs in the MTB because it is untagged). We see the iw2k_l1stq48_l2stq256 performance comes very close to the ideal 2048-entry instruction window model.

The hierarchical store queue design copes with store to load forwarding complexity without paying a performance hit due to increased forwarding latency. The design does not reduce the latency or the energy needed to access the large L2 STQ CAM array. For the hierarchical store queue design to perform well, forwarding from the L2 STQ has to be infrequent, in spite of the frequent store data forwarding on Pentium processors due to limited logical register space.

Table IV shows store-load forwarding characteristics with the hierarchical store queue. Column 2 in Table IV shows the percentage of loads that read data from either the L1 or L2 store queues. 26% to 42% of loads are forwarded data from the store queues in our CPR model, depending on the benchmark suite. However, the majority of forwarding is from the L1 STQ and less than 20% of the forwarded loads get their data from the L2 STQ, as shown in column 3. In addition to L2 STQ forwarding, up to 4% of all loads (column 4) are subject to an extra L2 STQ lookup latency due to false hits from address aliasing in the MTB. The

size of the MTB we use is 256 entries, and aliasing can be reduced further with larger MTB sizes. However, the MTB needs to be kept small enough to allow access latency within the time it takes to read the data cache and the L1 STQ.

Column 5 shows the percentage of all loads that search the L2 STQ (column 5 = (column 2 * column 3)/100 + column 4)). Less than 10% of all loads are subject to the L2 STQ lookup latency. The performance impact of the increased L2 STQ latency in our large window CPR design is minor, since our large window design is well suited for handling these relatively infrequent load operations that require L2 STQ lookup.

4.3 Physical Register Reclamation

Current processors use in-order instruction retirement to determine when a physical register may be freed, causing the lifetime of a physical register to be typically much larger than the lifetime of the instruction allocating that register. This artificially constricts supply of free physical registers and necessitates large register files.

However, most physical registers can be freed much earlier—as soon as all readers of the physical register have read the physical register, and the logical register corresponding to the physical register has been renamed again. Such an aggressive register reclamation scheme enables physical register usage to more closely match the true lifetimes of registers. Hence, rather than build large register files, we focus on efficiently reclaiming physical registers to provide the performance of a large register file without actually building one.

The aggressive register reclamation scheme can be implemented by associating a use counter and an unmapped flag with each physical register [Moudgill et al. 1993]. A physical register's use counter is incremented in the rename stage of the pipeline, when the input operand of an instruction (the reader) is mapped to the physical register. The use counter is decremented in the register read stage of the pipeline, when the reader actually reads the physical register. A physical register's unmapped flag is set when the logical register corresponding to the physical register is renamed again. A physical register can be reclaimed once its use counter value is 0, and its unmapped flag is set.

Using map table checkpoints makes the above aggressive register reclamation scheme easier to implement. Since a checkpoint provides the ability to restore the architecturally correct register state, physical registers belonging to a checkpoint (i.e., physical registers which are mapped to logical registers at the time of checkpoint creation) should not be released until the corresponding checkpoint is released. Hence, when a checkpoint is created, we increment the use counters for all physical registers belonging to the checkpoint. Similarly, when a checkpoint is released, we decrement the use counters of all physical registers belonging to the checkpoint. Using checkpoints as a reader guarantees physical registers are not released until all checkpoints to which they belong are released.

The unmapped flags are made part of the checkpoint. Hence, even if a misspeculated instruction overwrites a logical register, a checkpoint recovery results in these flags being restored to the correct values corresponding to the

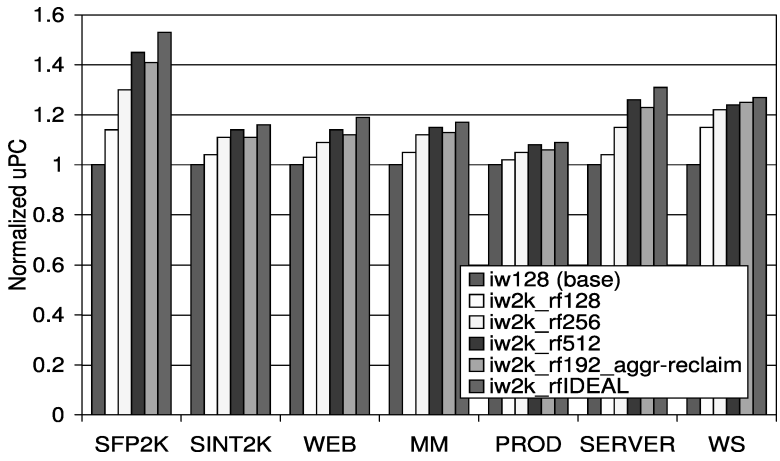

Fig. 9. Aggressive register reclamation performance.

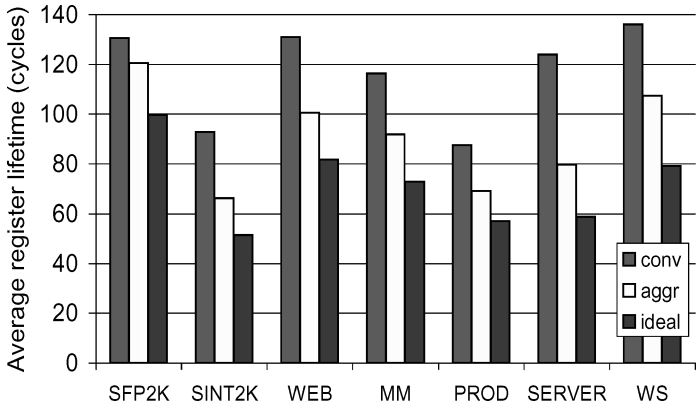

Fig. 10. Average register lifetimes for different register reclamation schemes.

checkpoint. Furthermore, all misspeculated instructions drain out of the pipe, as is done in current processors, and decrement any counters they incremented. Doing so is necessary for a processor with checkpoints to function correctly in the event of branch mispredictions, and also handle interrupts and exceptions precisely.

Figure 9 shows the results. The label *iwN_rfM* corresponds to an instruction window of size N and a register file with M integer and M floating point registers. The scheduling window size, branch recovery, and store queue size are idealized. Figure 5 in Section 3.4 shows a 512-entry register file (512 integer and 512 floating point registers) achieves performance close to an ideal configuration. Our results in Figure 9 show a 192-entry register file with aggressive reclamation achieves performance similar to that of a 512-entry register file with conventional reclamation.

To understand why the aggressive register reclamation scheme works, Figure 10 shows the average lifetime of physical registers mapped to logical

Table V. Checkpointed Register Statistics

Bench.	Per-cycle Average Number of		
	Checkpoints	Int Regs.	FP Regs.
SFP2K	3	25	13
SINT2K	4	27	—
WEB	3	22	—
MM	3	23	15
PROD	3	22	—
SERVER	4	26	—
WS	3	22	—

registers (including both integer and floating point registers) for three register reclamation schemes: a conventional ROB-based scheme (*conv*), the aggressive scheme used in CPR (*aggr*), and an ideal scheme (*ideal*) that uses a counter-based aggressive scheme similar to the one used in CPR but knows (using oracle information) the last reader of a physical register even if the logical register corresponding to the physical register has not been re-mapped.

From Figure 10, we see CPR's aggressive register reclamation scheme reduces register lifetimes by an average of 21% across all benchmarks. A further 16% reduction in lifetimes can be achieved using the ideal scheme. Practical implementations of the ideal scheme would require either predicting the last reader of registers, speculatively releasing the registers and recovering from mispredictions, or using the compiler to annotate the program with the last reader information, and is left as future work.

With CPR, the lifetimes of the checkpointed physical registers with the *aggr* and *ideal* schemes may increase compared to the *conv* scheme because the checkpointed physical registers cannot be released until the checkpoints to which they belong retire. Table V shows the average number of integer and floating point physical registers held by checkpoints, and the average number of checkpoints present in the window per cycle. There are between 3 and 4 checkpoints in the processor. The checkpoints hold onto between 22 and 27 integer physical registers. Since SFP2K and MM are the benchmark suites that frequently use the floating point physical registers, checkpointed floating point register results are shown only for these benchmarks. For these benchmarks, around 15 floating point physical registers are tied down by checkpoints at any point of execution.

The register lifetimes shown in Figure 10 include the lifetimes of both the checkpointed and noncheckpointed physical registers. In spite of the increase in register lifetimes of the checkpointed physical registers with CPR, Figure 10 shows a decrease in register lifetimes for the the *aggr* and *ideal* schemes. Hence, for both the *aggr* and *ideal* schemes, the reduction in lifetimes of the noncheckpointed physical registers is significantly higher than shown in Figure 10.

5. THE CPR MICROARCHITECTURE

Until now we have evaluated our individual proposals in isolation. In this section we evaluate the checkpoint processing and recovery microarchitecture that incorporates our earlier individual proposals for misprediction recovery,

aggressive register reclamation, and hierarchical store queues. Section 5.1 discusses some design aspects of CPR processors. Integrating the various mechanisms into one microarchitecture has implications for conventional ROB-based microarchitectures and we discuss them in Section 5.2. Section 5.3 presents performance results, and Section 5.4 presents CPR's large instruction window characteristics.

5.1 CPR Microarchitecture Design Implications

CPR uses numerous counters—from counters for tracking allocated instructions, counters for reclaiming registers, to counters in the store queues. To ease counter management, we allow all instructions to eventually decrement the counters, including squashed instructions that are merely drained out of the pipeline. Only when a counter becomes zero do we release its associated resource. For example, if an instruction is squashed due to a branch mispredict, the instruction will still decrement any related counters even as it is draining out of the pipeline without affecting any other architected state. Thus we do not require any global reset signals for counters in the various structures.

CPR allows a commit of hundreds of instructions instantaneously by simply manipulating a counter. These instructions may include many branches and store operations. A ROB-based architecture would retire stores in sequence and update any branch predictor serially as branches are retired. These functions traditionally associated with a single instruction commit, as occurs in a ROB-based design, need to be handled differently with a checkpoint-based architecture's bulk commit. We update the branch predictor once the branch executes rather than when the branch retires. We observe that speculatively updating a branch predictor does not degrade performance.

To enable the hierarchical store queue to handle bulk commits, each store's checkpoint identifier is associated with its corresponding entry in the store queue. This identifier is used to perform a bulk commit/squash of stores in the store queue. Manipulating similar bits in store queues is already done in modern out-of-order processors.

5.2 CPR and Reorder Buffers

Reorder buffers are used in modern processors to provide in-order semantics for instructions in the instruction window. This in-order semantic is currently needed for (1) recovering from branch mispredictions, (2) retiring state to registers and memory in program order, (3) providing precise interrupts, and (4) managing the register file, either in the form of providing storage for renamed registers or, if a separate physical register file is used, a mechanism for reclaiming physical registers after retirement.

As an instruction window size increases, using the ROB for misprediction recovery and register file management inhibits performance gains. While it may be possible to build very large ROBs (mainly FIFO structures), we argue that doing so does not provide performance because the real performance limiters are the mechanisms that use the ROB, and not the ROB itself. We have shown this in earlier sections where branch misprediction recovery mechanisms and

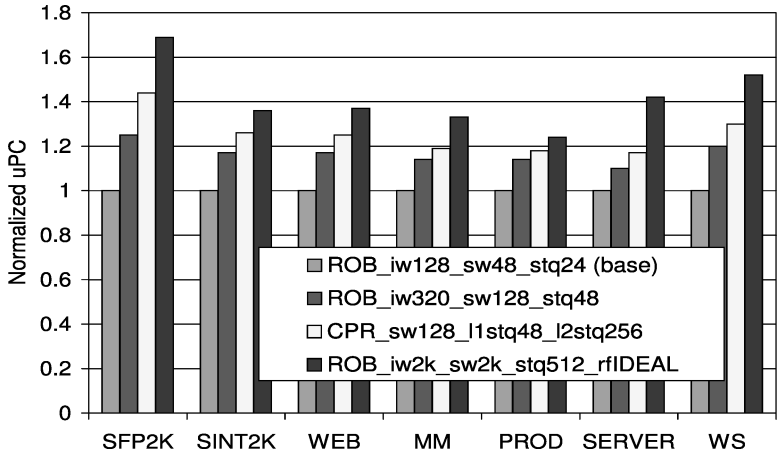

Fig. 11. CPR performance.

register reclamation severely limit performance as processors go to large instruction windows.

CPR uses confidence-based checkpoints and checkpoint counters for misprediction recovery where each checkpoint could correspond to a group of hundreds of instructions. A branch misprediction results in rollback to the closest checkpoint. Aggressive register reclamation occurs using counters whereby decoupling register reclamation from the in-order instruction retirement semantics provided by the ROB. Further, as discussed in Section 4.1, the same checkpoints above are also used for recovering from faults and in providing precise interrupts.

CPR replaces the functionality of the ROB by new and scalable mechanisms. We have thus developed a high performance ROB-free architecture. It may indeed be time to retire the ROB itself.

5.3 CPR Performance Analysis

Figure 11 presents CPR processor performance results. The CPR processor has a 128-entry scheduling window, 8 map table checkpoints made selectively at low-confidence branches, 192 integer and 192 floating point register with aggressive reclamation, a 48-entry L1 STQ, and a 256-entry L2 STQ. The CPR processor is compared to two conventional processors and an ideal 2048-entry instruction window. The first conventional configuration (also the baseline) has a 128-entry instruction window (iw128), a 48-entry scheduling window (sw48), and a 24-entry store queue (stq24) signified by iw128_sw48_stq24. The second conventional configuration is iw320_sw128_stq48. Both have appropriately sized register files. The remaining parameters for all configurations are similar to Table I except now we use a distance-based memory dependence predictor for all schemes.

The iw320_sw128_stq48 configuration has equal buffer capacity as the CPR processor for all timing critical parameters except our proposal replaces the 320-entry ROB of the conventional processor with an 8-entry checkpoint array.

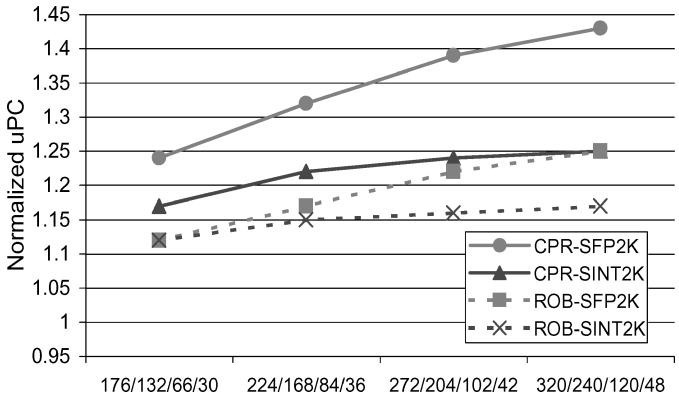

Fig. 12. Equal critical-resource comparison.

From the graph, we make two key observations:

1. *CPR processor with eight checkpoints outperforms a conventional* 320*-entry ROB processor*. SFP2000 benchmarks gain the most. These benchmarks frequently miss the cache and access memory. In such cases, large instruction windows are necessary to look far ahead and find independent instructions to execute. This performance gain is present even though the processor we model in all these configurations uses an aggressive 16-stream data prefetch hardware.
2. *CPR processor achieves between* 40% *and* 75% *of ideal performance for a* 2048*-entry instruction window*. Server benchmarks achieve the least performance gain. This is due to load stalls resulting from a significant number of predicted load-to-store dependences. Higher performance can be gained with better predictors and is left as future work.

Equal critical-resource comparison. To determine the resource efficiency of our microarchitecture, we compare the performance of a conventional ROB-based processor to a CPR processor that uses equal critical resources, for various processor configurations. For each configuration, the register file and the scheduling window are kept the same for both CPR and ROB-based processors. The timing critical L1 STQ size used in the CPR processor matches the store queue size used in the ROB-based processor. The CPR processor also uses a 256-entry L2 STQ not on the cycle critical path.

Figure 12 shows the results for the SFP2K and SINT2K benchmarks. The x-axis label format $w/x/y/z$ corresponds to a design for a w-entry instruction window, x integer and x floating point registers, a y-entry scheduling window, and a z-entry store queue. While the CPR model uses eight map checkpoints for each point instead of a ROB, the ROB model has a w-entry reorder buffer at each point. The remaining x, y, and z parameters are identical for the CPR and the ROB. The y-axis has normalized uPC with respect to the baseline configuration (128/96/48/24). The results show, for equal buffer sizes, a CPR processor outperforms a ROB-based processor even for small configurations. This highlights the resource utilization efficiency of the CPR processor.

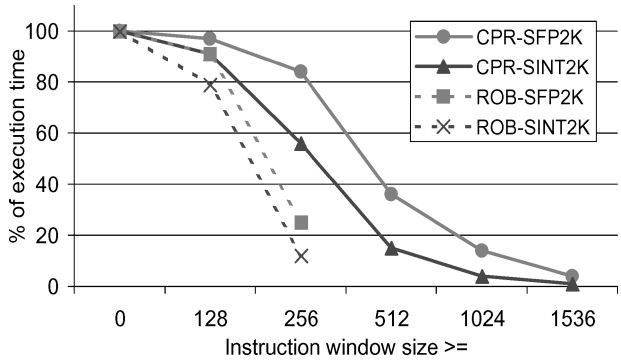

Fig. 13. ROB, CPR instruction window size distribution.

Table VI. Load Miss Characteristics

Bench.	Percentage of Retired Instructions in Load Miss Shadow	Percentage of Miss-dependent Instructions in Load Miss Shadow	Miss-dependent Instructions Per Load Miss	In Absence of Miss, Average Instructions in	
				Scheduler	ROB
SFP2K	13	23	10	263	1293
SINT2K	7	20	15	462	2051
WEB	7	24	12	242	819
MM	8	34	10	310	965
PROD	2	22	11	436	1049
SERVER	26	23	12	149	441
WS	19	25	5	133	1157

5.4 CPR Instruction Window Characteristics

In this section, we study the instruction window characteristics of CPR. Figure 13 shows the instruction window size distribution for a 256-entry ROB machine and an equal critical resource CPR machine for the SINT2K and SFP2K benchmarks. A point (x, y) on the graph indicates that for $y\%$ of cycles, the instruction window had greater than or equal to x instructions in the instruction window. The CPR machine can sustain more than 512 instructions in the instruction window for a significant fraction of the execution time.

A large instruction window is needed mostly in the presence of load misses that go to main memory. Column 2 of Table VI shows the average percentage of retired instructions that are in the instruction window in the shadow of a load miss (i.e., from the time a load miss is detected till the load data returns). The SFP2K, SERVER, and WS benchmarks have a higher fraction of instructions in the shadow of load misses than the other benchmarks. The performance benefit due to large instruction windows is also higher for these benchmarks.

Column 3 of Table VI shows that 20%–34% of instructions in the instruction window in the shadow of load misses depend on the miss data. With an average window size of 300–600 instructions for CPR, a 256-entry scheduler provides enough capacity to hold miss-dependent instructions without stalling, most of the time. The average number of miss-dependent instructions per load miss is

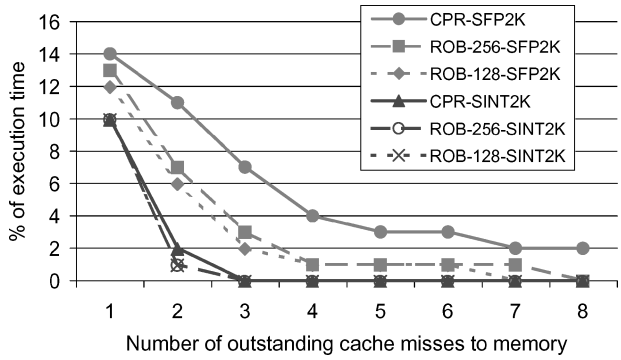

Fig. 14. ROB, CPR outstanding L2 cache miss distribution.

even smaller (column 4 in Table VI), since multiple loads often access the same cache miss block.

Even in the absence of cache misses, we see a small percentage of instructions waiting in the scheduler for execution compared to the total number of instructions in a large window processor. Columns 5 and 6 in Table VI show the average number of instructions in the scheduler and the average number of instructions in the ROB, respectively, when measured on an ideal 2K window/2K scheduler ROB-based machine with perfect cache and perfect branch prediction. Because scheduler entries are freed when their instructions are issued and ROB entries are freed when instructions are retired, out-of-order issue and in-order retirement cause instructions to occupy ROB entries a lot longer than scheduler entries. Therefore, scaling the scheduler is a lot less critical than scaling the total instruction window for high performance large window machines such as CPR, both in the presence and absence of cache misses.

One of the main benefits due to the large instruction windows achieved by CPR is the ability to look far ahead in the program and to overlap multiple independent misses to main memory (also referred to as memory level parallelism or MLP). Figure 14 shows the L2 cache miss distributions for the CPR machine and the 256-entry ROB machine for the SINT2K and SFP2K benchmarks. A point (x, y) on the graph indicates that for $y\%$ of cycles, there were at least x outstanding L2 cache misses in the instruction window. For the SFP2K benchmarks that have a high number of L2 cache misses, we see the CPR machine overlaps multiple misses for a higher fraction of the execution time. Since the execution times of the CPR machine is smaller than that of the ROB machine, the L2 cache miss distribution curves for the CPR machine are always higher than those for the ROB machine, in spite of having similar number of L2 cache misses as the ROB machine.

The results in this section show that CPR can sustain a large instruction window without requiring large cycle-critical resources, and CPR significantly outperforms a conventional ROB machine given equal resources. Hence, CPR is a promising microarchitecture for designing future high-performance microprocessors.

6. RELATED WORK

Checkpointing for repairing architectural state due to branch mispredictions and exceptions was proposed by Hwu and Patt [1987]. Their proposal also did not employ a ROB but rather used checkpoints at conditional branches to restore state. The Pentium 4 uses a retirement register alias table to track maps [Hinton et al. 2001], while the MIPS R10000 [Yeager 1996] and Alpha 21264 [Leibholz and Razdan 1997] use a checkpoint method to recover rename maps. The checkpoint method as used by the MIPS R10000 and Alpha 21264 do not scale as instruction windows become larger. History buffers have also been proposed to restore the register rename map table [Ranganathan et al. 1997; Smith and Pleszkun 1985].

Using checkpoints for tolerating long memory latencies was first proposed by Cristal et al. [2002]. In their initial proposal, the virtual ROB [Cristal et al. 2002], and in subsequent work [Cristal et al. 2004], checkpoints are created at long latency loads and periodic intervals and ROB entries of instructions following a long latency operation are released even before they complete. They call the latter mechanism out-of-order commit. This enables them to emulate a large virtual ROB while using a small physical ROB. As the ROB entries are released, their physical registers are also released early. Cristal et al. [2003] also studied the utilization of various critical resources in a processor that supports a large number of in-flight instructions.

Run-ahead execution [Dundas and Mudge 1997] uses checkpointing for prefetching in the shadow of a cache miss, while selectively checkpoints the map table at weakly predicted branches [Moshovos 2003] that have been proposed to reduce number of checkpoints and power in the rename unit.

The Cherry proposal [Martínez et al. 2002] uses the ROB and recycles physical registers and other resources once their associated instructions are branch-safe and memory-safe; that is, all branches prior to the instruction have completed and all loads have issued. Early resource reclamation is limited to a subset of the ROB. A checkpoint of the architected register file is also used but only for recovering from exceptions and the ROB is used for retiring instructions. They do not address the problem of branch misprediction recovery latency.

The IBM Power4 provides the effect of a larger ROB by assigning groups of up to six instructions to a ROB entry [Tendler et al. 2002]. Here, the ROB size still grows linearly with instructions. However, our selective checkpoint scheme allows much larger scaling without requiring a ROB.

Various register file organizations have been proposed [Balasubramonian et al. 2001; Capitanio et al. 1992; Cruz et al. 2000]. The counter method for reclaiming physical registers has been proposed earlier [Moudgill et al. 1993] for a ROB-based MIPS R10000-style processor.

Research in instruction scheduling has focused on logic design for large scheduling windows [Brown et al. 2001], and dependency-based scheduling queues [Palacharla et al. 1997]. Decentralized schedulers, where a collection of small schedulers are associated with functional units, have also been extensively studied [Hinton et al. 2001; Palacharla et al. 1997]. Large instructions

buffers have been proposed from where instructions dependent on other long latency instructions are reactivated for scheduling when the long latency instructions complete [Lebeck et al. 2002].

7. CONCLUDING REMARKS

In this paper, we show it is possible to implement a processor with a large instruction window without requiring large cycle-critical buffers. We do this by carefully examining several critical aspects of large instruction windows and studying the sensitivity to performance of each aspect in isolation with other aspects idealized. These studies reveal that while larger scheduling windows are important, other design aspects become more critical as an instruction window size increases. We use this insight to develop efficient mechanisms for handling branch misprediction recovery, forwarding data to dependent loads from large number of stores, and physical register reclamation. We combine these mechanisms to propose a new microarchitecture based on checkpoint processing and recovery (CPR).

CPR is a ROB-free architecture requiring only a small number of rename map checkpoints selectively created at low-confidence branches, while capable of supporting a large instruction window of the order of thousands of instructions. We argue that while large ROBs can be built, the mechanisms that depend upon the ROB, such as branch misprediction recovery and conventional physical register reclamation, are the real performance limiters. Thus, simply building a large ROB would not by itself provide performance gains. CPR decouples misprediction recovery and register reclamation from the ROB, and uses a scalable hierarchical store queue, thus allowing for scalable high-performance solutions for supporting very large instruction windows. While CPR scales easily to very large windows, it also outperforms a conventional ROB-based design even with the exact same cycle-critical buffer sizes.

ACKNOWLEDGMENTS

We thank Mike Fetterman, Stephan Jourdan, Konrad Lai, Eric Rotenberg, John Shen, Jim Smith, Jared Stark, and Mike Upton for discussions and comments.

REFERENCES

AKKARY, H., RAJWAR, R., AND SRINIVASAN, S. T. 2003. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proceedings of the 36th International Symposium on Microarchitecture*.

BALASUBRAMONIAN, R., DWARKADAS, S., AND ALBONESI, D. 2001. Reducing the complexity of the register file in dynamic superscalar processors. In *Proceedings of the 34th International Symposium on Microarchitecture*. 237–249.

BLOOM, B. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM 13*, 7 (July), 422–426.

BROWN, M. D., STARK, J., AND PATT, Y. N. 2001. Select-free instruction scheduling logic. In *Proceedings of the 34th International Symposium on Microarchitecture*. 204–213.

CAPITANIO, A., DUTT, N., AND NICOLAU, A. 1992. Partitioned register files for VLIWs: A preliminary analysis of tradeoffs. In *Proceedings of the 25th International Symposium on Microarchitecture*. 292–300.

CRISTAL, A., MARTINEZ, J. F., LLOSA, J., AND VALERO, M. 2003. A case for resource-conscious out-of-order processors. In *Computer Architecture Letters*.

CRISTAL, A., ORTEGA, D., LLOSA, J., AND VALERO, M. 2004. Out-of-order commit processors. In *Proceedings of the 10th International Symposium on High-Performance Computer Architecture*. 48–59.

CRISTAL, A., VALERO, M., LLOSA, J.-L., AND GONZALEZ, A. 2002. *Large Virtual ROBs by Processor Checkpointing*. Tech. Rep. UPC-DAC-2002-39, Department of Computer Science, Barcelona, Spain. July.

CRUZ, J.-L., GONZALEZ, A., VALERO, M., AND TOPHAM, N. P. 2000. Multiple-banked register file architectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*. ACM Press. 316–325.

DUNDAS, J. AND MUDGE, T. 1997. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 1997 International Conference on Supercomputing*. 68–75.

HINTON, G., SAGER, D., UPTON, M., BOGGS, D., CARMEAN, D., KYKER, A., AND ROUSSEL, P. 2001. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*.

HWU, W. W. AND PATT, Y. N. 1987. Checkpoint repair for out-of-order execution machines. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*. 18–26.

JACOBSEN, E., ROTENBERG, E., AND SMITH, J. E. 1996. Assigning confidence to conditional branch predictions. In *Proceedings of the 29th International Symposium on Microarchitecture*. 142–152.

KARKHANIS, T. AND SMITH, J. E. 2002. A day in the life of a data cache miss. In *Workshop on Memory Performance Issues*.

LEBECK, A. R., KOPPANALIL, J., LI, T., PATWARDHAN, J., AND ROTENBERG, E. 2002. A large, fast instruction window for tolerating cache misses. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*. 59–70.

LEIBHOLZ, D. AND RAZDAN, R. 1997. The Alpha 21264: A 500 MHz out-of-order execution microprocessor. In *Proceedings of the 42nd IEEE Computer Society International Conference (COMPCON)*. 28–36.

MARTÍNEZ, J. F., RENAU, J., HUANG, M. C., PRVULOVIC, M., AND TORRELLAS, J. 2002. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *Proceedings of the 35th International Symposium on Microarchitecture*.

MOSHOVOS, A. 2003. Checkpointing alternatives for high performance, power-aware processors. In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design*. ACM Press, New York, 318–321.

MOSHOVOS, A. AND SOHI, G. S. 1997. Streamlining inter-operation memory communication via data dependence prediction. In *Proceedings of the 30th International Symposium on Microarchitecture*. 235–245.

MOUDGILL, M., PINGALI, K., AND VASSILIADIS, S. 1993. Register renaming and dynamic speculation: an alternative approach. In *Proceedings of the 26th International Symposium on Microarchitecture*. 202–213.

PALACHARLA, S., JOUPPI, N. P., AND SMITH, J. E. 1997. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*. 206–218.

RANGANATHAN, P., PAI, V. S., AND ADVE, S. V. 1997. Using speculative retirement and larger instruction windows to narrow the performance gap between memory consistency models. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*. 199–210.

SMITH, J. E. AND PLESZKUN, A. R. 1985. Implementation of precise interrupts in pipelined processors. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*. 36–44.

SPRANGLE, E. AND CARMEAN, D. 2002. Increasing processor performance by implementing deeper pipelines. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*. 25–34.

TENDLER, J. M., DODSON, S., FIELDS, S., LE, H., AND SINHAROY, B. 2002. POWER4 system microarchitecture. *IBM J. Res. Dev. 46*, 1 (Jan.), 5–25.

YEAGER, K. 1996. The MIPS R10000 superscalar microprocessor. *IEEE Micro 16*, 2 (April), 28–40.

Received June 2004; revised October 2004; accepted October 2004