CHECKPOINT PROCESSING AND RECOVERY: AN EFFICIENT, SCALABLE ALTERNATIVE TO REORDER BUFFERS

PROCESSORS REQUIRE A COMBINATION OF LARGE INSTRUCTION WINDOWS AND HIGH CLOCK FREQUENCY TO ACHIEVE HIGH PERFORMANCE. TRADITIONAL PROCESSORS USE REORDER BUFFERS, BUT THESE STRUCTURES DO NOT SCALE EFFICIENTLY AS WINDOW SIZE INCREASES. A NEW TECHNIQUE, CHECKPOINT PROCESSING AND RECOVERY, OFFERS AN EFFICIENT MEANS OF INCREASING THE INSTRUCTION WINDOW SIZE WITHOUT REQUIRING LARGE, CYCLE-CRITICAL STRUCTURES, AND PROVIDES A PROMISING MICROARCHITECTURE FOR FUTURE HIGH-PERFORMANCE PROCESSORS.

Haitham Akkary
Portland State University

Ravi Rajwar
Srikanth T. Srinivasan
Intel Microarchitecture Research Lab

• • • • • • Achieving high performance in modern microprocessors requires a combination of exposing large amounts of instruction level parallelism (ILP) and processing instructions at a high clock frequency. Exposing maximum ILP requires the processor to operate concurrently on large numbers of instructions, also known as the instruction window; a high-frequency design requires frequently accessed structures to be small and fast. These requirements are somewhat at odds with each other—hardware structures must be sufficiently large to buffer and process all instructions in a large instruction window, yet must remain fast enough to support high frequencies. Thus, new designs that focus on eliminating large, cycle-critical hardware structures become necessary to achieve high performance.

We have proposed Checkpoint Processing and Recovery (CPR) as an efficient microarchitecture capable of sustaining large instruction windows without requiring large critical structures.[1] In our original work, we presented a detailed study of performance issues for large instruction windows in addition to a CPR design. Here, we focus on an out-of-order processor with a reorder buffer (ROB).

In conventional designs, each instruction in the instruction window must have an entry in the ROB. We argue that to build scalable, large instruction windows, future processors must move away from a ROB-centric design

0272-1732/03/$17.00 © 2003 IEEE Published by the IEEE computer Society

of key processor mechanisms and toward a checkpoint-oriented processing model. We show how a ROB, although conceptually simple, limits the microarchitecture's scalability. We then demonstrate how a CPR microarchitecture overcomes the inherent limitations of the ROB while truly allowing scalability.

Reorder buffers

Modern out-of-order processors typically consist of several pipeline stages, which perform several operations on an instruction: fetch; decode and rename; issue and execute out of order; reorder to return the instructions to the original program order; and retire.

In the front-end of the pipeline, the processor fetches and inserts instructions into the pipeline. If a branch instruction redirects control flow to a new location, the processor must redirect instruction fetch to that location. The processor cannot determine the new location with certainty until the branch completes execution; waiting for the branch to execute before redirecting fetch could introduce significant delays. Hence, the processor employs branch prediction and speculatively processes instructions after the branch.

The decode and rename stage examines fetched instructions and constructs dependences. This process detects both true data dependences (read-after-write hazards) and false data dependences (write-after-write and write-after-read hazards). Although the processor retains true data dependences, it removes false data dependences by renaming the logical registers to a larger pool of microarchitecture-visible physical registers and recording the renaming using a rename map table.

The processor issues an instruction for execution once its operands are ready. If it is a memory instruction, the processor takes appropriate steps to ensure the maintenance of dependences via memory locations, through loads and stores. As long as the processor maintains both register and memory dependences, instruction issue can occur in any order. This is the out-of-order aspect of modern processors.

Once these instructions complete execution, the processor reinserts them back into the original program order to implement precise interrupts and exceptions,[2] typically using the ROB. The processor allocates a ROB entry in program order for every instruction dispatched by the pipeline's front end. This entry serves as a holding area for the instruction until it retires.

ROB-based instruction retirement

When a completed instruction reaches the head of the ROB, it retires. Instruction retirement typically consists of committing the instruction's register or memory data to architectural state and freeing its ROB entry. Thus, the ROB enforces an in-order retirement of instructions, which simplifies register reclamation and rename map table recovery in the event of branch mispredicts, interrupts, and exceptions.

ROB-based register reclamation

A conventional scheme reclaims a physical register when the instruction that remaps the logical register corresponding to the physical registers retires. Such a scheme uses the in-order retirement of instructions to guarantee a reclaimed physical register is not needed to be a part of the processor architectural state later.

ROB-based rename map table recovery

If the front end mispredicts a branch, instructions from the incorrect path can update the rename map table. Once the mispredicted branch completes execution, the processor must recover the correct rename map table and undo the incorrect-path instruction updates. This recovery must occur before renaming correct-path instructions that follows the mispredicted branch.

Three ROB-based techniques for this map table recovery exist. In the first, the processor maintains an additional rename map table at retirement and updates it only with correct-path instructions. Once the mispredicted branch reaches the ROB's head, the processor copies this table to the front-end map table. The second technique is similar to the first, but instead of waiting for the mispredicted branch to reach the ROB's head, the processor proactively walks the ROB from the head toward the mispredicted branch, incorporating rename information at each ROB entry. In the third technique, the processor walks the ROB from its tail (the most recently allocated instruction) toward the mispredicted branch, undoing the changes to the rename map table made by each incorrect path instruction. This scheme requires each ROB entry to store the rename map that its corresponding instruction overwrote.

The processor delays renaming the front end until it recovers the rename map table; this delay varies, depending on the policy employed. Handling interrupts and exceptions also requires recovery of the rename map table, and the processor can employ any of these techniques.

Thus, the ROB is essential for the processor to implement mechanisms for instruction retirement, physical-register reclamation, and rename map table recovery. Although this arrangement simplifies conceptual reasoning about out-of-order processors, it significantly limits scalability, as we discuss next.

Reorder buffer limitations for large-instruction-window processors

An uncompleted instruction cannot release its ROB entry. If such an instruction reaches the ROB's head, it blocks the ROB until it completes and retires. To prevent pipeline stalls because of the resulting lack of ROB entries, the ROB must be sized in proportion to the instruction window. Since the ROB is a simple first-in, first-out structure, a large ROB can be built. However, as the ROB size increases, the mechanisms associated with the ROB do not scale well.

With ROB-based map table recovery schemes, the front end cannot start renaming new correct-path instructions until either the mispredicted branch reaches the ROB's head, or the processor finishes serially walking the ROB to reconstruct the map table corresponding to the mispredicted branch. This delay in restarting the renaming becomes longer for larger ROBs and limits performance.

The ROB-based physical-register reclamation scheme causes the lifetime of a physical register to far exceed the lifetime of the instruction to which it is allocated. Because most instructions, except stores and branches, require the allocation of a destination physical register, such a ROB-based physical-register reclamation scheme forces the register file size to be of the same order as the instruction window size. Naively increasing the register file size degrades performance by increasing register access time. Sophisticated and large register file organizations, such as hierarchical register files, are complex and adversely impact die size and power.

An alternative: Checkpoint processing and recovery

We present a novel and efficient CPR microarchitecture for building large-instruction-window processors. CPR decouples key mechanisms from the ROB and provides a scalable alternative to current ROB-based processors. The key idea behind CPR involves the notion that sometimes reconstructing architectural state is more efficient than explicitly storing state, as long as sufficient information is available for reconstruction. Thus, unlike ROB-based approaches that record state at every instruction, CPR records state only at carefully selected points of an execution and will regenerate state for individual instructions only if necessary. Such an approach is inherently scalable and more efficient than ROB-based approaches.

By means of such selective state tracking, CPR efficiently handles the key processor mechanisms of rename map table recovery, instruction retirement, and physical-register reclamation. Here, we briefly summarize how CPR provides these mechanisms; our original work includes additional details and studies.[1]

CPR rename map table recovery

The recovery mechanism for the rename map table must be fast and have low overhead. Previous work had proposed periodically creating checkpoints to recover processor state at every branch.[3] However, for large instruction windows, such periodic checkpoints incur high storage overhead and are inherently not scalable. CPR recognizes the performance advantage of checkpoints and significantly reduces the overhead of periodic checkpoints by limiting the number of such checkpoints to carefully selected points in the instruction window.

Because branch mispredicts are the most frequent cause of rename map table recovery, ideally, we would like to create checkpoints to recover processor state exactly at mispredicted branches. So CPR creates map table checkpoints at branches with a high mispredict probability, selecting these branches by using a branch confidence estimator.[4]

Because CPR does not create checkpoints at every branch, once the processor resolves a non-checkpointed mispredicted branch, it restarts execution from the checkpoint just before the mispredicted branch. This can cause reexecution of the nonspeculative instructions between the checkpoint instruction and the mispredicted branch. The branch confidence estimator works well in minimizing this checkpoint overhead. We also use the same checkpoints created at low-confidence branches to implement precise interrupts and exceptions, and to deal with architecture-specific serializing instructions.

The selective placement of checkpoints together with reexecution to restore architectural state achieves an effective combination of fast recovery time and scalability. Further, the number of checkpoints does not limit the instruction window size because each checkpoint can correspond to many instructions, and the number of instructions per checkpoint can vary across checkpoints. These characteristics let CPR support an adaptive instruction window size that meets applications' needs—a small instruction window for applications with frequent branch mispredicts and a large instruction window for applications with infrequent branch mispredicts.

Mispredicting a non-checkpointed branch will force a recovery to a prior checkpoint. CPR prevents repeated misprediction of the same branch by using the branch outcome from the previously aborted execution itself rather than a prediction. Furthermore, once the processor resolves a mispredicted branch and begins reexecuting from a prior checkpoint (say, C1), CPR forces a new checkpoint (say, C2) at the first branch, independent of whether the branch is low-confidence or not. This lets instructions between checkpoints C1 and C2 retire under all situations, including any pathological cases where multiple branches are alternatively mispredicted. A similar mechanism that forces a checkpoint on the instruction immediately after a prior checkpoint, handles exceptions and memory consistency events, such as snoop invalidations.

CPR instruction retirement

CPR allocates and reclaims checkpoints in a first-in-first-out order, and a checkpoint buffer keeps track of map table checkpoints. Each checkpoint buffer entry has a counter to determine if the processor can free the corresponding checkpoint. The counter tracks the completion of instructions associated with the checkpoint; it increments when the processor allocates an instruction and decrements when the instruction completes execution. CPR prevents counter overflow by forcing a new checkpoint.

CPR allocates a checkpoint only if a free checkpoint is available. If the processor fetches a low-confidence branch and no checkpoints are available, it 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. Not stalling execution even if the checkpoint buffer is full is important for high performance. CPR reclaims the oldest checkpoint when its associated counter has a value of zero, and the next checkpoint has been allocated. This means all the instructions associated with the oldest checkpoint have been allocated and have completed execution.

Each instruction has an identifier associating it to a specific checkpoint. The processor uses this identifier to access the appropriate checkpoint buffer for incrementing and decrementing the counter, and for selecting instructions to squash or commit. When the last instruction belonging to a checkpoint completes, the processor can instantly retire all the instructions in that checkpoint and reclaim the associated checkpoint. This enables CPR to commit hundreds of instructions instantly, thereby removing the serialized retirement constraints enforced by a ROB.

CPR physical-register reclamation

Instead of waiting for the in-order instruction retirement of the ROB to determine register reclamation, CPR employs an aggressive register reclamation scheme. This scheme reclaims a physical register as soon as all its readers have completed their reads, a strategy that becomes possible if the original register state is recoverable when necessary. To ensure recoverability of such state in CPR, physical registers mapped to logical registers at the time of checkpoint creation are not reclaimed until CPR releases the corresponding checkpoint. All other registers can be reclaimed as soon as possible without waiting for instruction retirement. This enables physical-register usage to match the lifetimes of registers more closely than ROB-based reclamation schemes. CPR can thus achieve the performance of a large register file without requiring one.

CPR's register reclamation mechanism associates a use counter and an unmapped flag with each physical register.[5] In the pipeline's rename stage, when the processor maps the input operand of an instruction (the reader) to a physical register, it increments the physical register's use counter. In the pipeline's register read stage, when the reader actually reads the physical register, it decrements the physical register's use counter. When the processor remaps the logical register corresponding to the physical register, it sets the physical register's unmapped flag. Reclamation of a physical register can occur when the register's use counter value is 0, and its unmapped flag is set.

To prevent the release of a checkpoint's physical register before the checkpoint's release, we increment the use counters for all physical registers belonging to a checkpoint at the time of checkpoint creation. Similarly, when the processor releases a checkpoint, it decrements the use counters of all physical registers belonging to the checkpoint. Treating checkpoints as readers guarantees that physical registers are not reclaimed until after the release of all checkpoints to which they belong.

The unmapped flags are part of the checkpoint. Hence, even if a misspeculated instruction overwrites a logical register, checkpoint recovery restores these flags to the correct values corresponding to the checkpoint. Furthermore, all mispredicted instructions drain out of the pipe, as typical in current processors, and decrement any counters they incremented. Doing so allows a processor with checkpoints to handle branch mispredicts, interrupts, and exceptions.

Other limits to scalability

So far, we have discussed how CPR uses selective checkpoints to address a ROB's limitations when the ROB is used to manage key processor mechanisms of scalable map table recovery, instruction retirement, and register reclamation. However, checkpoints are only a step in the direction of large instruction windows; other issues unrelated to ROBs still limit scalability. We now briefly discuss two of these issues: the instruction scheduler and the processor store queue.

Instruction scheduler

The scheduler holds renamed instructions not yet issued for execution and examines these instructions each cycle to find ready instructions for issue. In out-of-order processors, a typical scheduler fills and blocks only in the presence of long-latency operations, such as loads that miss the cache and go to main memory. The scheduler can continue to issue independent instructions, and only instructions that depend on a load missing the cache will occupy scheduler entries for a long time. Our experimental results show the set of instructions dependent on loads that miss the cache is small. Hence a relatively small scheduler size is enough to support a large instruction window: a 128- to 256-entry scheduler is sufficient for a 2,048-entry instruction window. Thus, the scheduler does not have to scale with the instruction window, hence we do not focus on schedulers. However, building a 128- to 256-entry scheduler is not an easy task, and further reducing the scheduler's size is an important area for research.

Store queue

A store instruction occupies a store queue entry from the time it is renamed until it retires. Traditionally, stores are retired in order to satisfy branch misprediction, precise exception, and memory consistency requirements and thus typically stay in the store queue for long durations. Further, the store queue size is directly proportional to the number of stores in the instruction window. The large number of stores in a large instruction window combined with the fact that each store stays in the store queue for a long duration requires large store queues for large instruction windows.

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 affects cycle time. It will be difficult to increase conventional store queue sizes (32 to 48 entries) without making the forwarding circuit a critical path, thus increasing cycle time.

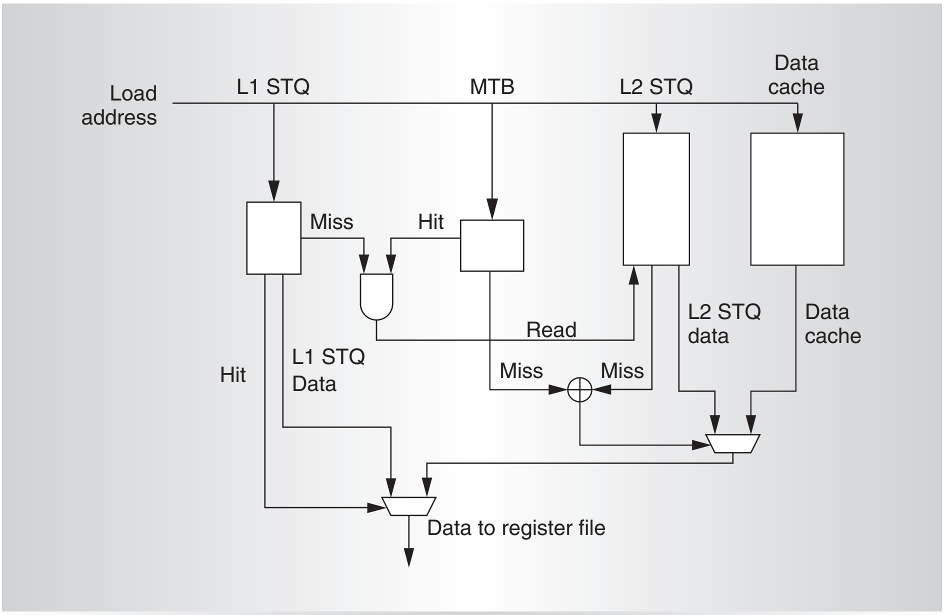

Figure 1. Hierarchical store queue.

To address the limitations of conventional store queues, CPR uses a novel hierarchical store-queue organization, consisting of a small (32 to 48 entries) and fast (level-one data cache latency) level-one store queue (L1 STQ), backed by a much larger (approximately 256 entries) and slower (level-two data cache latency) level-two store queue (L2 STQ). Figure 1 shows the hierarchical two-level store queue's organization. The L1 STQ holds the most recent stores while the L2 STQ holds older stores displaced from the L1 STQ. Because stores typically forward to nearby loads, most store-to-load forwarding occurs from the L1 STQ without affecting cycle time. The L2 STQ provides the capacity to store several stores and infrequently handles store-to-load forwarding.

Store-to-load forwarding. If a load hits in the L1 STQ, it receives the data at the level-one data cache latency. When a load misses in the L1 STQ, it could get its data either from the L2 STQ or from the memory hierarchy (caches and main memory). Before accessing the memory hierarchy, we could wait for the full L2 STQ access latency to determine if the load hit or miss in the L2 STQ. However, we would then incur the L2 STQ access latency even if the data misses in the L2 STQ and hits in the level-one data cache. Because this is a common case and the delay decreases performance, we must quickly determine if a load is going to hit or miss the L2 STQ. For this purpose, we use a Membership Test Buffer (MTB).

The MTB is similar to a bloom filter [6] and has the property that if it indicates a load hit in the L2 STQ, the load has a high chance of hitting in the L2 STQ. On the other hand, if the MTB indicates a load will miss in the L2 STQ, it will certainly miss in the L2 STQ. The MTB is a direct-mapped, non-tagged array of counters indexed by a part of a load or store address. When a store is removed from the L1 STQ and placed into the L2 STQ it increments the corresponding untagged MTB entry. When a store retires, updates the data cache, and is removed from the L2 STQ, it decrements the corresponding untagged MTB entry. A nonzero count in the MTB entry potentially indicates a matching store in the L2 STQ. On the other hand, a zero count in the MTB entry guarantees a matching store does not exist in the L2 STQ.

When a load issues and reads the data cache, the load also accesses the L1 STQ and the MTB in parallel. If the load hits the L1 STQ, it forwards the store data to the load. If the load misses the L1 STQ, and the MTB indicates a L2 STQ miss, the data cache forwards the data to the load. If the load misses the L1 STQ and the MTB indicates a potential hit in the L2 STQ, the load waits to allow sufficient time to access the L2 STQ and resolve the load-store dependence. If the load hits the L2 STQ, it supplies the data to the load. If the load misses the L2 STQ, the data cache forwards the data to the load. However, the load would have already suffered a delay equivalent to an L2 STQ access latency. To minimize spurious hits in the MTB from address aliasing, the MTB should be as large as possible, yet have an access time within those of the L1 STQ and the data cache.

Store address unknowns and the load buffer. With a large instruction window, there is a high chance that when the processor issues loads, the store queues will contain an unknown store address. Although conventional means can be used to detect the presence of unknown store addresses in the L1 STQ, for the L2 STQ, we use the MTB to indicate whether it potentially has an unknown store address. Stalling the issue of a load in the presence of an unknown store address has a significant negative impact on performance. Store-load dependences are highly predictable.[7] 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. CPR detects 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. A memory dependence violation incurs a penalty equivalent to flushing the instruction window and rolling back execution to a previous checkpoint. Because the coverage of our memory dependence predictor is high, CPR seldom incurs this penalty for dependence violations.

Bulk commit/squash of stores. Since CPR requires the capability to bulk commit/squash all stores belonging to a checkpoint, CPR associates each store's checkpoint identifier with its corresponding entry in the store queue. This identifier is used to perform a bulk commit/squash of stores in the hierarchical store queue.

Thus, CPR comprehensively addresses all critical design aspects of large instruction windows and provides a unified mechanism for building large instruction window processors.

ROB versus CPR

Our original paper presents a comprehensive set of results to measure the suitability of CPR for building large instruction windows.[1] Here, we compare the performance of CPR with that of a ROB-based machine and show the ability of CPR to support large instruction windows.

Equal critical-resource comparison

To determine the resource efficiency of CPR, we compare the performance of a conventional ROB machine to a CPR machine that uses equal critical resources for various processor configurations. For each configuration, we keep the critical resources—the register file and the scheduling window—the same for CPR and ROB machines. The timing-critical L1 STQ size used in the CPR machine matches the store queue size used in the ROB machine. The CPR machine also uses a 256-entry L2 STQ that is not on the cycle-critical path.

Figure 2 shows average results for SPEC2000 integer (SINT2K), floating-point (SFP2K), server, and workstation (WS) benchmark suites; other benchmark suite results are similar. The *y* axis is the particular configuration's percentage speedup over the performance of a 128-entry ROB baseline (the smallest ROB configuration in the figure). The *x* axis indicates the various processor configurations. The first number for each configuration is the ROB size and applies only to the ROB machine. The CPR machine does not use a ROB and instead uses eight map table checkpoints to achieve an adaptive instruction window. The second number is the physical register file size, a value of *p* indicates *p* integer and *p* floating-point registers. The third set of numbers (*i*-*f*-*m*) is the scheduler size, where *i* is the number of entries for the integer instructions; *f*, for the floating-point instructions; and *m*, for the memory instructions.

The results show that for equal buffer sizes a CPR machine consistently and significantly outperforms a ROB machine. More interestingly, the 128/96/(32-32-16) CPR configuration has only half the number of critical resources as the 256/192/(64-64-32) ROB configuration, yet outperforms the larger ROB

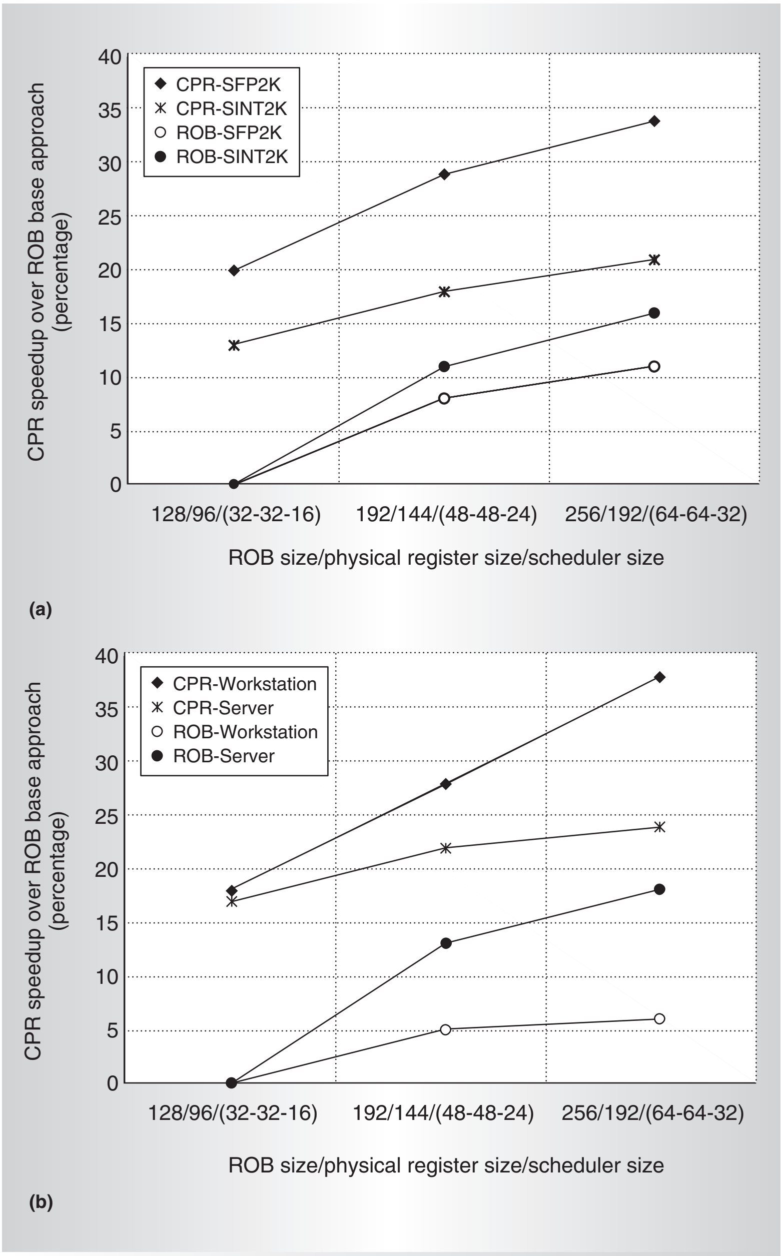

Figure 2. Comparison of ROB and CPR machines having an equal number of critical resources: Integer and floating-point (a); and workstation and server (b) benchmarks. ROB size is in number of entries, and physical-register size indicates the number of integer and floating-point registers. Scheduler size has three numbers: the number of entries for the integer, floating-point, and memory instructions.

machine for all benchmarks. This highlights the resource utilization efficiency of the CPR machine.

Instruction window size comparison

Figure 3 shows the average instruction window size achieved by a 256-entry ROB machine and a CPR machine with equal critical resources. The average instruction window size for a benchmark is the instruction window size per cycle computed over the entire run of the benchmark. As these results show, the resource efficiency of the CPR machine allows it to sustain average instruction window sizes two or four times larger than a ROB-based machine with equal critical resources. Further, we find CPR sustains thousands of instructions in the window for significant periods of execution time. This shows CPR can sustain large instruction windows without requiring large cycle-critical resources; hence, it is well suited to building processors with large instruction windows.

Achieving high performance requires large instruction windows operating at high frequencies. Although conventional processors designs rely on a centralized ROB, the key mechanisms traditionally associated with the ROB are either too slow or too costly to support the large ROBs required for large instruction windows. By decoupling key mechanisms from the ROB, CPR opens up new opportunities for processor optimizations. Schemes that might have been ineffective in the presence of a serializing structure such as a ROB now might achieve their true potential. Our novel CPR microarchitecture offers a scalable and efficient alternative to ROB-based processors and provides a promising microarchitecture for designing future high-performance processors. MICRO

References

1. H. Akkary, R. Rajwar, and S.T. Srinivasan, “Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors,” *Proc. 36th Ann. Int’l Symp. Microarchitecture*, ACM Press, 2003, p. 423-434.
2. J.E. Smith and A.R. Pleszkun, “Implementation of Precise Interrupts in Pipelined Processors,” *Proc. 12th Ann. Int’l Symp. Computer Architecture*, ACM Press, 1985, pp. 36-44.
3. W.W. Hwu and Y.N. Patt, “Checkpoint Repair For Out-Of-Order Execution Machines,” *Proc. 14th Ann. Int’l Symp. Computer Architecture*,

l Press, 1987, pp. 18-26.
4. E. Jacobson, E. Rotenberg, and J.E. Smith, "Assigning Confidence to Conditional Branch Predictions," *Proc. 29th Int'l Symp. Microarchitecture*, ACM Press, 1996, pp. 142-152.
5. M. Moudgill, K. Pingali, and S. Vassiliadis, "Register Renaming and Dynamic Speculation: an Alternative Approach," *Proc. 26th Int'l Symp. Microarchitecture*, ACM Press, 1993, pp. 202-213.
6. B. Bloom, "Space/Time Tradeoffs in Hash Coding with Allowable Errors," *Comm. ACM*, vol. 13, no. 7, July 1970, pp. 422-426.
7. A. Moshovos and G. Sohi, "Streamlining Inter-Operation Memory Communication via Data Dependence Prediction," *Proc. 30th Int'l Symp. Microarchitecture*, ACM Press, 1997, pp. 235-245.

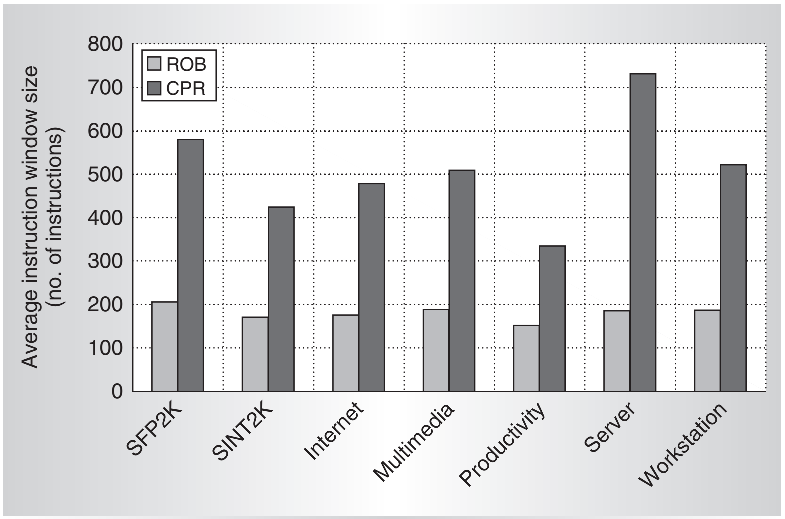

Figure 3. Comparison of average instruction window sizes for ROB and CPR machines.

Haitham Akkary is an assistant professor in the Electrical and Computer Engineering Department at Portland State University. His main research interest is microprocessor architecture. Akkary has a BS and MS in from Louisiana State University, and a PhD from Portland State University, all in electrical and computer engineering. He is a member of the IEEE and the ACM.

Ravi Rajwar is with the Intel Microarchitecture Research Lab. His research interests include the theoretical and practical aspects of computer architecture. Rajwar received a PhD degree in computer science from the University of Wisconsin-Madison.

Srikanth T. Srinivasan is a senior researcher in the Intel Microarchitecture Research Lab. His current research focuses on scalable and efficient microarchitectures. Srinivasan received a BE in computer science from the Birla Institute of Technology and Science, Pilani, India; and a PhD in computer science from Duke University.

Direct questions and comments about this article to Srikanth T. Srinivasan, Microarchitecture Research Lab, Intel Corp., 2111 NE 25th Avenue, Hillsboro, OR 97124; srikanth.t.srinivasan@intel.com.