# Toward Kilo-instruction Processors

ADRIÁN CRISTAL, OLIVERIO J. SANTANA, and MATEO VALERO
Universitat Politècnica de Catalunya
and
JOSÉ F. MARTÍNEZ
Cornell University

The continuously increasing gap between processor and memory speeds is a serious limitation to the performance achievable by future microprocessors. Currently, processors tolerate long-latency memory operations largely by maintaining a high number of in-flight instructions. In the future, this may require supporting many hundreds, or even thousands, of in-flight instructions. Unfortunately, the traditional approach of scaling up critical processor structures to provide such support is impractical at these levels, due to area, power, and cycle time constraints.

In this paper we show that, in order to overcome this resource-scalability problem, the way in which critical processor resources are managed must be changed. Instead of simply upsizing the processor structures, we propose a smarter use of the available resources, supported by a selective checkpointing mechanism. This mechanism allows instructions to commit out of order, and makes a reorder buffer unnecessary. We present a set of techniques such as multilevel instruction queues, late allocation and early release of registers, and early release of load/store queue entries. All together, these techniques constitute what we call a *kilo-instruction processor*, an architecture that can support thousands of in-flight instructions, and thus may achieve high performance even in the presence of large memory access latencies.

Categories and Subject Descriptors: C.1.1 [**Processor Architectures**]: Single Data Stream Architectures—*RISC / CISC, VLIW architectures*; C.4 [**Performance of Systems**]: Design Studies

General Terms: Design, Performance

Additional Key Words and Phrases: Memory wall, instruction-level parallelism, multicheckpointing, kilo-instruction processors

## 1. INTRODUCTION

Microprocessor speed improves at a much higher rate than memory access latency. As a result each new processor generation requires a higher number of processor cycles to access main memory. A plethora of techniques have been

Authors' addresses: Adrián Cristal, Oliverio J. Santana, and Mateo Valero, Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, Edifici D6, Campus Nord, c/ Jordi Girona 1-3, 08034 Barcelona, Spain; email: {adrian,osantana,mateo}@ac.upc.edu; José F. Martínez, Computer Systems Laboratory, Cornell University, Frank H.T. Rhodes Hall, Ithaca, NY 14853, USA; email: martinez@csl.cornell.edu.

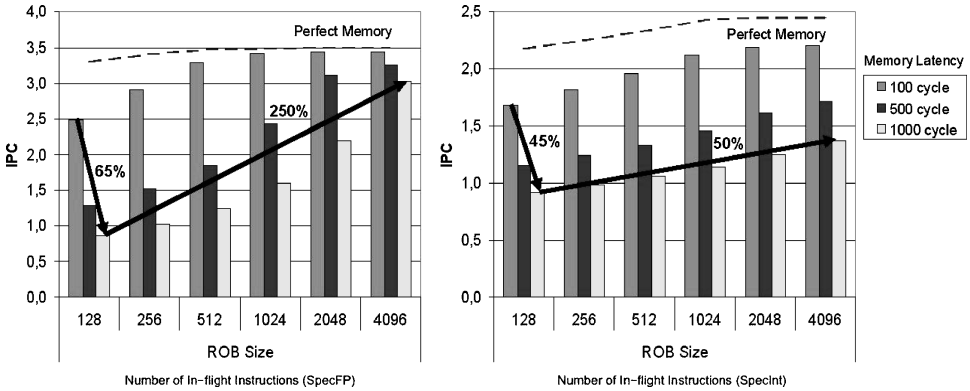


Fig. 1. Average performance of a four-issue out-of-order superscalar processor executing SPEC2000 floating-point and integer programs, for different memory latencies and number of supported in-flight instructions.

proposed to overcome this limitation, such as cache hierarchies [Smith 1982], hardware and software prefetching [Baer and Chen 1991; Joseph and Grunwald 1997; Klaiber and Levi 1991; Mowry et al. 1992], or helper threads [Chapell et al. 1999; Collins et al. 2001; Dubois and Song 1998; Roth and Sohi 2001; Sohilin et al. 2002; Zilles and Sohi 2001]. However, they do not completely solve the problem, which has become one of the most important limiting factors for the performance of high-frequency microprocessors.

A different approach to tolerating very long memory latencies is to substantially increase the number of in-flight instructions supported. A processor able to maintain thousands of in-flight instructions can overcome the latency of memory operations by overlapping memory accesses with the execution of independent instructions. Figure 1 shows the impact of increasing the number of in-flight instructions supported. When using the maximum number of in-flight instructions typical of today's processors (in the order of 128 instructions), an increase in memory latency from 100 to 1000 processor cycles causes a dramatic performance degradation for both integer and floating-point applications 45 and 65%, respectively. The main reason is that the reorder buffer is full 72% and 87% of the time, respectively, blocking the insertion of new instructions into the pipeline due to the in-order-commit nature of current out-of-order superscalar processors. As a consequence, such a processor does not have enough in-flight instructions to overlap the memory access latency with useful work.

The obvious solution is to support more in-flight instructions. Increasing the maximum number of in-flight instructions to 4096 provides a whopping 250% performance improvement for floating-point applications over the original 128-instruction, 1000-cycle-latency configuration, coming close to the performance of a perfect-memory configuration. For integer applications, hard-to-predict branches and pointer chains limit gain; nevertheless, even in this case, performance improves by 50%. With a great deal of research being devoted to improving branch prediction, integer programs should benefit more from a high number of in-flight instructions in the near future.

Unfortunately, supporting a high number of in-flight instructions typically involves scaling-up critical processor structures, such as reorder buffer, instruction queues, physical register file, or load/store queues. Because of area, power, and cycle time limitations, this is very difficult, if not impossible, to accomplish in current processor designs. Thus, the challenge is to design an architecture able to support a high number of in-flight instructions without depending as much on resource enlargement.

In this paper, we present quantitative evidence that critical processor structures are highly underutilized, and we exploit this fact to propose *kilo-instruction processors*, where we make intelligent use of the available resources instead of merely upsizing.

In particular, we discuss an out-of-order commit technique [Cristal et al. 2002a, 2002b, 2004a] that allows early release of reorder buffer (ROB) entries, and in fact allows the possibility of eliminating the ROB entirely. We also present mechanisms for smart use of the instruction queues [Cristal et al. 2004a] and aggressive management of the register file [Cristal et al. 2003c; Martínez et al. 2003]. Kilo-instruction processors can also accommodate recently proposed techniques to improve the management of load/store queues [Akkary et al. 2003; Cristal et al. 2002b; Martínez et al. 2002; Park et al. 2003; Sethumadhavan et al. 2003].

As a critical-enabling mechanism, kilo-instruction processors make use of selective checkpointing of the architectural state. This allows aggressive resource management at all levels, even at the expense of occasional corrupt processor state, which is resolved by rolling back to a saved checkpoint. All together, these techniques constitute an efficient way to deal with future memory latencies.

The remainder of this paper is organized as follows. Section 2 shows that critical processor structures are underutilized. Section 3 describes out-of-order instruction commit. Section 4 presents an efficient technique to manage the instruction queues. Section 5 discusses aggressive management of the physical register file. Section 6 describes techniques for managing large load/store queues. Section 7 evaluates the performance of kilo-instruction processors. We discuss our current research lines in Section 8. Finally, we conclude in Section 9.

## 2. CRITICAL RESOURCES

Current superscalar processors rely on in-order instruction commit to preserve original program semantics and support precise exceptions and interrupts. This imposes severe constraints on the use of the critical processor resources:

—Every decoded instruction requires a ROB entry until the instruction commits. Consequently, supporting thousands of in-flight instructions requires a ROB with thousands of entries.

—Every decoded instruction requires an entry in its corresponding instruction queue (IQ) until it is issued for execution. Consequently, in order to support thousands of in-flight instructions, IQs would require a very large number of entries, or else they could fill-up often with blocked instructions that depend on long-latency operations.

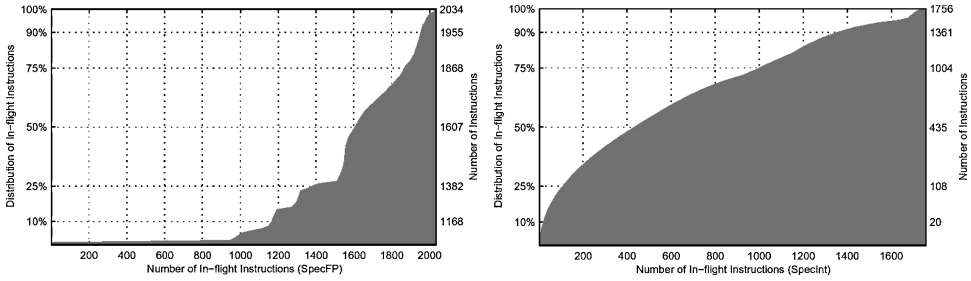


Fig. 2. Distribution function of in-flight instructions in SPEC FP and SPEC Int applications.

—Every renamed instruction that generates a result requires a physical register, which is assigned at the rename stage, and it is not released until the next instruction defining the same logical register commits. Consequently, supporting thousands of in-flight instructions requires an elevated number of physical registers.

—Every decoded memory instruction requires an entry in the load/store queues, which is not released until commit. Consequently, to support thousands of in-flight instructions, a large queue with complex disambiguation logic would be needed.

Unfortunately, scaling-up these structures makes the processor design impractical, not only due to area and power consumption limitations, but also because these structures will likely affect the processor cycle time [Palacharla et al. 1997]. However, a close look reveals that all these structures are highly underutilized, in part due to the in-order commit strategy. In this section, we analyze the usage of each one of these critical processor structures, showing that a significant fraction of such structures is indeed being wasted by blocked or executed instructions.

### 2.1 Reorder Buffer

Figure 2 shows the average cumulative distribution of in-flight instructions for SPEC FP and SPEC Int applications on a simulated four-issue out-of-order processor that supports up to 2048 in-flight instructions and has 500 cycle memory latency [Cristal et al. 2003a]. It can be seen that floating-point applications frequently exhibit a high amount of in-flight instructions: over 1800 instructions 75% of the time. On the other hand, integer applications show a relatively lower number of in-flight instructions, in part because branches are mispredicted more frequently, causing more instruction squashes. Nevertheless, the number of in-flight instructions is still high: over 400 instructions 50% of the time.

It becomes clear that, since each in-flight instruction needs a ROB entry until it commits, the ROB must be rather large to support all these in-flight instructions. This may pose a resource-scalability problem. In Section 3, we describe a mechanism that we call *out-of-order commit*, based on selective processor checkpointing, which preserves program semantics and supports precise exceptions and interrupts, all without the need for a traditional ROB structure.

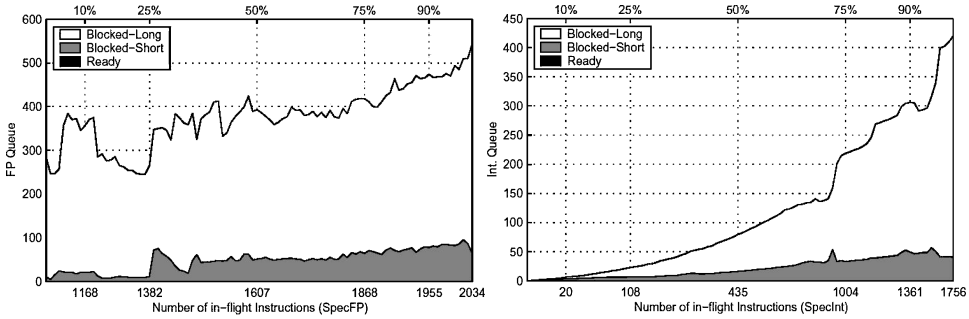


Fig. 3. Breakdown of allocated floating-point and integer instruction queue entries in SPEC FP and SPEC Int applications, respectively. The horizontal axis is adjusted according to the distribution function of in-flight instructions in each case (Figure 2).

## 2.2 Instruction Queues

Figure 3 shows the average number of allocated entries in the floating-point queue (for SPEC FP applications) and in the integer queue (for SPEC Int applications), plotted against the distribution of in-flight instructions from Figure 2. Floating-point and integer queues require 500 and 300 entries, respectively, in order to support in-flight instructions up to the 90th percentile. The complexity associated with such queues is almost certain to impact the processor's clock cycle significantly.

Fortunately, not all instructions behave in the same way. In the figure, instructions are divided into two groups: *blocked-short instructions*, if they are waiting for a functional unit or for results from short-latency operations, and *blocked-long instructions*, when they are waiting for some long-latency instruction to complete, such as a load instruction that misses in the L2 cache. The figure shows that this second group comprises by far the largest fraction of entries allocated in the instruction queues. In Section 4, we describe a smart IQ management technique that takes advantage of this fact.

## 2.3 Physical Register File

Figure 4 shows the average number of allocated floating-point and integer registers for SPEC FP and SPEC Int applications, respectively, against the distribution of in-flight instructions in Figure 2. To provide support up to the 90th percentile, nearly 1200 floating-point and 800 integer registers are required by SPEC FP and SPEC Int applications, respectively. Such size is impractical not only due to area and power limitations, but also because it requires a high access time, which is bound to impact the processor's cycle time.

However, a classification of allocated registers sheds new light. In the figure, live registers contain values currently in use. Blocked-short and blocked-long registers have been allocated during rename, but are blocked because the corresponding instructions are waiting for the execution of predecessor instructions. In particular, blocked-short registers are owned by instructions that will issue shortly, while blocked-long registers are owned by instructions that are blocked

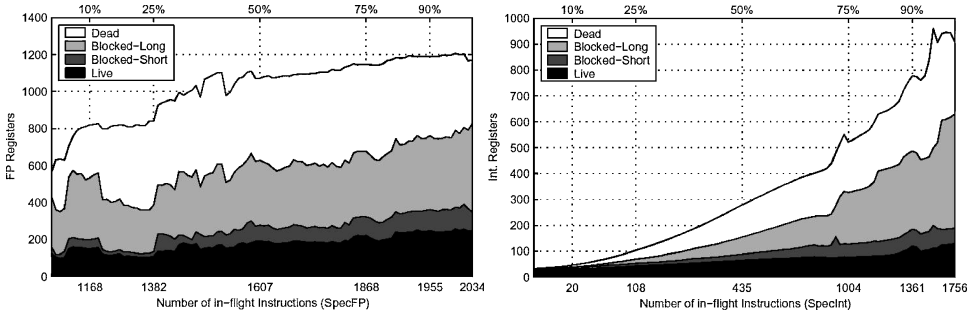


Fig. 4. Breakdown of allocated floating-point and integer registers in SPEC FP and SPEC Int applications, respectively. The horizontal axis is adjusted according to the distribution function of in-flight instructions in each case (Figure 2).

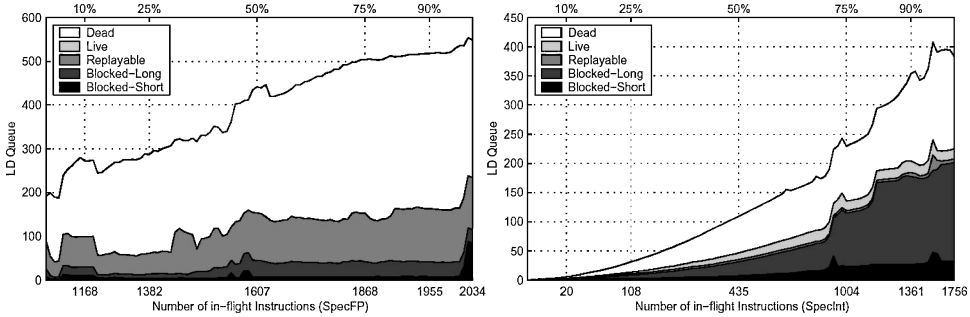


Fig. 5. Breakdown of allocated load queue entries in SPEC FP and SPEC Int applications. The horizontal axis is adjusted according to the distribution function of in-flight instructions in each case (Figure 2).

waiting for long-latency instructions. Finally, dead registers are no longer in use, but they are still allocated because the superseding producer instructions have not yet committed.

As it turns out, blocked-long and dead registers constitute the largest fraction of allocated registers. Using the appropriate techniques, dead registers can be released early, and blocked-long registers can be allocated late. In Section 5, we describe an aggressive register management mechanism that improves the efficiency of the physical register file by attacking both fronts simultaneously.

## 2.4 Load/Store Queue

Figure 5 shows the average number of allocated load queue entries, plotted against the distribution of in-flight instructions from Figure 2. To cover all cases up to the 90th percentile, more than 350 and 500 entries are required in the case of SPEC Int and SPEC FP applications, respectively. These sizes would not only increase the access time to the queues; they would also increase the complexity of the memory disambiguation logic. As before, the figure breaks down the allocated entries in different categories. Live entries correspond to loads that are being executed. Replayable entries represent loads that have executed out

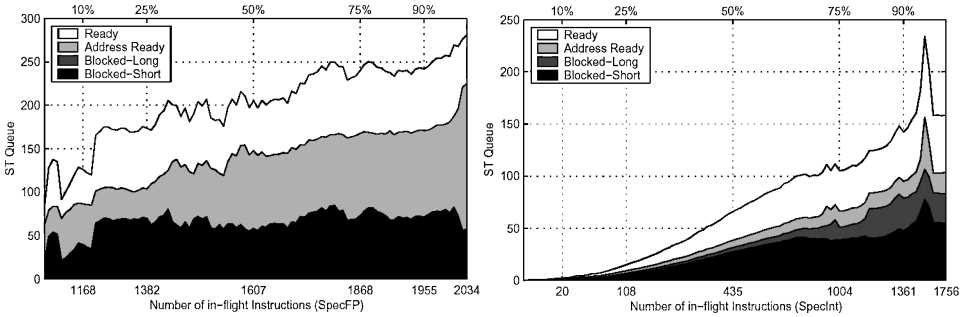


Fig. 6. Breakdown of allocated store queue entries in SPEC FP and SPEC Int applications. The horizontal axis is adjusted according to the distribution function of in-flight instructions in each case (Figure 2).

of program order with respect to some store instructions whose address remains unresolved. Blocked-short and blocked-long entries correspond to loads waiting for its address to be produced by a short-latency or a long-latency operation. Finally, dead entries correspond to loads that have been executed and are not subject to store-load replay traps, that is, the addresses of all previous store instructions have been resolved.

Integer applications have a significant fraction of blocked-long and dead-allocated entries. In the case of blocked-long entries, they are due to the fact that pointer chains are quite common in integer applications. These chains can be attacked using a variety of techniques such as prefetching, pre-execution, or value prediction. Dead entries are most abundant in floating-point applications. Traditional out-of-order processors keep these dead entries until their load retires, but aggressive implementations that recycle them earlier have been proposed [Cristal et al. 2002b; Martínez et al. 2002].

Figure 6 shows the average number of allocated store queue entries. Almost 150 and 250 entries are needed to provide support up to the 90th percentile in integer and floating-point applications, respectively. Once more, this high number of entries will be taxing on the clock cycle due to queue access time and complexity of the memory disambiguation logic. In the figure, store entries are broken down into four categories: Ready entries represent store instructions whose address and source operand are available, and are only waiting to reach the ROB head to execute. Address-ready entries correspond to stores whose address is ready, but are still waiting for the data. Blocked-long and blocked-short entries represent store instructions whose address depends on a long-latency or a short-latency instruction, respectively.

Under the right conditions, the ready and address-ready entries can be excluded from the memory disambiguation process before their store executes, reducing the complexity of the memory disambiguation logic [Martínez et al. 2002]. Furthermore, as with load instructions, an adequate aggressive implementation can solve the scalability problems caused by store instructions [Akkary et al. 2003; Park et al. 2003; Sethumadhavan et al. 2003], which are discussed in Section 6. Alternatively, stores can be temporarily buffered in the cache hierarchy [Martínez et al. 2002].

## 3. MULTICHECKPOINTING AND OUT-OF-ORDER COMMIT

The ROB can be understood as a history window of all the in-flight instructions. Instructions are inserted in-order into the ROB after they are fetched and decoded. Instructions are also removed in-order from the ROB when they commit, that is, when they finish executing and update the architectural state of the processor.

In-order commit ensures that program semantics is preserved, even as instruction execution may be speculative and takes place out of order. It also provides support for precise exceptions and interrupts. However, in-order commit is a serious problem in the presence of large memory access latencies. Let us suppose that a processor has a 128-entry ROB, and that the memory access latency is 500 cycles. If a load instruction accesses the main memory due to a second-level cache miss, it cannot be committed until its execution finishes 500 cycles later. When the load becomes the older instruction in the ROB, it blocks the in-order commit, and no later instruction can commit until the load finishes. The processor can spend part of this time doing useful work, but once the ROB becomes full, a processor stall soon follows.

To avoid this, the processor should be able to support a higher number of in-flight instructions whose execution would overlap with the load access latency. As shown in Section 2.1, several hundreds or even thousands of in-flight instructions would be required to achieve this. However, scaling-up the number of ROB entries to support this is impractical. A large ROB not only carries a higher implementation cost, but also an increase in the size of the associated control logic. As a result, accessing such a large hardware structure may impact the processor cycle time.

A kilo-instruction processor attacks this problem by using a checkpointing mechanism that enables out-of-order instruction commit [Cristal et al. 2002a, 2002b, 2004a]. The use of selective checkpointing allows kilo-instruction processors to preserve program semantics and supports precise exceptions and interrupts without requiring ROB support at all. This allows implementing the functionality of a large ROB without requiring an impractical centralized structure with thousands of entries.

### 3.1 Processor Checkpointing

Checkpointing is a well-established technique for restoring the correct architectural state of the processor after misspeculations or exceptions [Hwu and Patt 1987; Smith and Pleszkun 1985]. A checkpoint can be thought of as a snapshot of the state of the processor, which is taken at a specific instruction of the program being executed. This checkpoint contains all the information required to recover the architectural state and continues the execution from that point.

In order to describe a checkpointing mechanism, four design issues should be taken into account. These four points are interdependent, and thus they should be considered together as a whole.

—How many in-flight checkpoints should be maintained by the processor? A large number of checkpoints reduce the penalty of the recovery process, since

it is more likely that there is a checkpoint near an instruction that causes a misprediction or an exception. Nevertheless, a large number of checkpoints also increase the implementation cost.

—What kind of instructions should be checkpointed? It is possible to take a checkpoint at any instruction. However, some instructions are better candidates than others. For example, some current processors take checkpoints at branch instructions in order to minimize the branch misprediction penalty. Other mechanisms can take checkpoints at other types of instructions, according to their needs.

—How often should the processor take a checkpoint? In general, the more frequently a processor takes checkpoints, the higher the overhead and the cost are. On the other hand, taking checkpoints more frequently reduces the penalty associated with restoring a valid state. Several heuristics could be used to overcome this trade-off. Numerical applications mainly require taking into account the load instructions that miss in the second-level cache. On the other hand, integer applications require more complex heuristics, which combine checkpoints at load and branch instructions. Since taking a checkpoint at all branches involves a higher implementation cost, they can be taken only at hard-to-predict branches. With this approach, there may be several pending branches between checkpoints, increasing the average misprediction penalty. However, since only hard-to-predict branches are expected to be mispredicted, this heuristic should reduce the overhead of taking checkpoints without much increasing the overall misprediction penalty.

—How much information should be kept by each checkpoint? In general, it is only necessary to keep the information strictly required to recover the correct processor state. However, it could be beneficial to store additional information in order to improve the processor performance. For example, the approach described in this paper [Cristal et al. 2004a] requires a few bits to recover the state of the rename mappings. Other approaches require more information to recover the rename mapping [Cristal et al. 2002b; 2003c; Martínez et al. 2003]. The drawback is that more information involves a higher cost for the checkpoint storage and management.

### 3.2 Multicheckpointing in Kilo-instruction Processors

Figure 7 shows an example of a multicheckpointing process. In general, there always exists at least one checkpoint in the processor (timeline A). The processor fetches and issues instructions, taking new checkpoints as needed, for example, when the risk of a rollback is high. The processor maintains a total order of the checkpoints. Every decoded instruction is associated with the most recent checkpoint, and each checkpoint keeps a count of its instruction group. If misspeculation occurs, for example, a branch misprediction (timeline B), the processor flushes all instructions from the associated checkpoint on and resumes instruction fetch and processing. The in-flight instructions before that checkpoint are not affected (timeline C). On the other hand, when an instruction finishes, its checkpoint's instruction counter is decremented. If

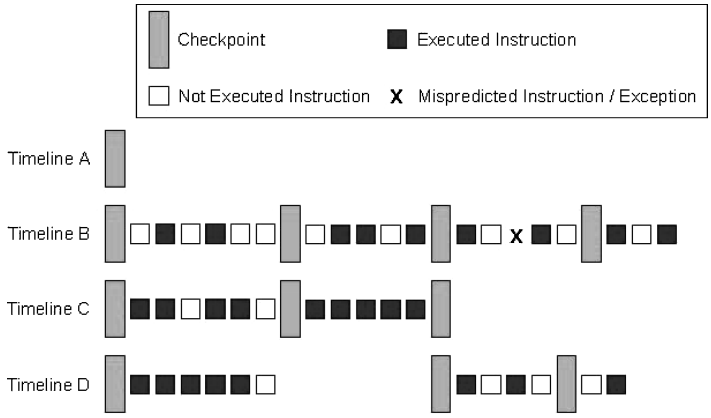


Fig. 7. An example of our multicheckpointing process.

the count reaches zero, all instructions associated with the checkpoint have executed (timeline C). At that point, the checkpoint is released (timeline D), and its instruction group is effectively merged with the preceding instruction group. (Since the instructions are all executed, the preceding checkpoint need not update its count.) If it is the oldest checkpoint's count that reaches zero, all instructions associated with it commit at once. This we call *gang-commit* [Martínez et al. 2003]. Finally, in the case of an exception, after restoring the associated checkpoint, our mechanism forces a checkpoint at the excepting instruction. This checkpoint reduces the execution penalty in the case exception appears again.

The use of selective checkpointing makes the traditional ROB unnecessary. In spite of this, the kilo-instruction processor keeps a small ROB-like structure. This structure, which we call pseudo-ROB [Cristal at al. 2004a], has the same functionality of a ROB. Decoded instructions are inserted in the pseudo-ROB in order. However, the instructions that reach the head of the pseudo-ROB are removed at a fixed rate, independent of their state. Since the processor state can be recovered from the pseudo-ROB, generating a checkpoint is only necessary when the instructions leave the pseudo-ROB. Delaying checkpoint generation is beneficial to alleviate the impact of branch mispredictions. Most branches are resolved while they are still in the pseudo-ROB [Cristal et al. 2002b], reducing the amount of pending branches between checkpoints. Indeed, over 90% of mispredictions are caused by branches that are still inside the pseudo-ROB. This means that most branch mispredictions do not need to roll back to the previous checkpoint for recovering the correct state, minimizing the misprediction penalty.

3.2.1 *Register Mapping.* Our register mapping uses a register alias table (RAT), which is implemented as a CAM memory having one entry per physical register. Figure 8 shows the traditional CAM structure of such a renaming mechanism, which is composed of a physical-logical register pair and a valid bit. There is also a register free list from where free registers are taken. For the sake of simplicity, we assume in this example that there are only seven

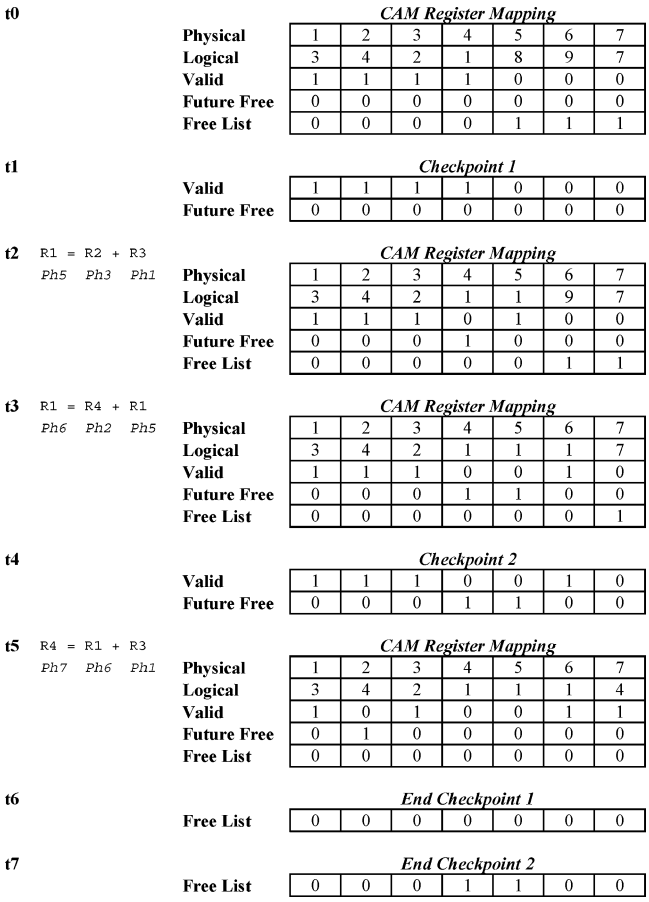


Fig. 8. Example of the proposed CAM register mapping.

physical registers and that the free list is implemented with a bit per physical register. For example, at instant $t0$, only four physical registers are mapped (the ones with the valid bit set to 1), while the other three registers are in the free list.

In a traditional renaming mechanism, physical registers are released when the instruction that redefines the logical register commits. Our proposal cannot use this technique because instructions are allowed to commit out-of-order. Therefore, we release physical registers when the processor removes the first checkpoint taken after the instruction that redefines the logical register. In order to achieve this, our register mapping contains an additional bit per physical register, which we call the *future free* bit. Future free bits are used to record which registers need to be freed between checkpoints. When a particular checkpoint commits, the future free bits are used to free the registers corresponding to the instructions associated with the previous checkpoint. This mechanism increases the pressure over the physical register file, delaying the register release, but this problem can be overcome with the techniques discussed in Section 5.

Let us suppose that the processor takes a checkpoint at instant *t1*. The processor does not need to keep the physical-logical mapping in the checkpoint. It is enough to keep the valid bits because the mapping is not going to change until the registers are freed after the commit of a later checkpoint. The multicheckpointing mechanism also needs to keep the current values of the future free bits, which are reset to compute the new values for the following checkpoint. Therefore, the cost of a checkpoint in our mechanism is the number of physical registers times 2 bits per register, which is very simple indeed.

Figure 8 also shows the state of the register mapping after a simple add instruction is renamed at instant *t2*. The physical register required by the add instruction is taken from the free list, that is, physical register 5 is assigned to logical register R1. The corresponding valid bit is set to 1 and the free list is updated accordingly. In addition, the processor also sets to 1 the future free bit associated with physical register 4, which was previously mapped to logical register R1, and thus it should be freed when the subsequent checkpoint commits. Figure 8 shows, at instant *t3*, the state of the register mapping after the next instruction with logical destination register R1 is renamed. It is important to note that there are two physical registers mapped with logical register R1, which will subsequently be freed at the same time.

Let us now suppose that a new checkpoint is taken at instant *t4* by keeping the valid and the future free bits. After the checkpoint is taken, all the future free bits are cleared, for them to be reused by the following checkpoint. The processor continues executing instructions, and at instant *t5*, a new instruction is renamed. The logical register R4 is mapped to the physical register 7. The valid bit associated with physical register 2 (the previous mapping of R4) is reset to 0 and the corresponding future bit is set to 1.

At instant *t6*, when all the instructions associated with the first checkpoint have finished (the instructions renamed at instants *t2* and *t3*), the checkpoint is removed, freeing all the corresponding physical registers. This is done using the future free bits. In this case, all future free bits are 0, so no physical registers are freed. When all instructions associated with the second checkpoint have finished, at instant *t7*, the processor also removes the second checkpoint. Now, the future free bits indicate that physical registers 4 and 5 should be freed. The corresponding logical registers were redefined by instructions executed between the first and the second checkpoints, and thus their contents will not be used again after the checkpoint is removed.

In the event of returning to a previous checkpoint, the fact that using CAMs in the renaming mechanism simplifies the computation of the free register list. To determine the registers in use, the processor performs a logical *or* between the valid vector bits and the future free vector bits of all active checkpoint maps and the current rename map.

3.2.2 *Taking Checkpoints.* Our mechanism systematically takes a new checkpoint if one of three thresholds is exceeded [Cristal et al. 2003b; 2004a]. (1) *At the first branch after* 64 *instructions*: Branches are good candidates for taking checkpoints because this allows minimizing the impact of branch mispredictions. (2) *After 64 store instructions*: Associating too many store instructions

with a particular checkpoint could degrade performance because they cannot release their resources until the corresponding checkpoint commits. (3) Finally, as a fallback case, after 512 instructions.

3.2.3 *Store Instruction Commit.* Store instructions should not send their data to memory until they commit. This is necessary to allow correct recovery of the architectural state in the case of a misprediction or an exception. Traditional processors keep the data in the store queue until the store instruction commits, at which point it is sent to memory. Our multicheckpointing mechanism also keeps the data in the store queue. However, the data belonging to all the store instructions associated with a checkpoint are not sent to memory until the checkpoint commits. The drawback of this technique is that it increases the pressure over the store queue. Nevertheless, several techniques for overcoming this problem have been proposed in the literature, as described in Section 6.

### 3.3 Related Work

To the best of our knowledge, the first multicheckpointing mechanism for overcoming the problems caused by maintaining thousands of in-flight instructions was proposed in January 2002 [Cristal et al. 2002a] and later developed by Cristal et al. [2002b]. The main novelty of this technique is that checkpointing enables an efficient way to control and manage the use of the critical processor structures. In this work, checkpoints are taken at load instructions. In particular, a checkpoint is taken when the ROB head is reached by a load that missed in the second-level cache. These checkpoints are useful to early, release instructions in the ROB, to release physical registers early, and to remove load instructions early from the load/store queue. The architecture presented in this paper is an evolution of this first approach to the multicheckpointing.

Cherry [Martínez et al. 2002] is another checkpointing scheme that was developed in parallel with Cristal et al. [2002b]. Instead of using a multicheckpointing mechanism, Cherry is based on a single checkpoint outside the ROB. This checkpoint uses a backup register file to keep all the architectural registers pointed to by the retirement mapping. The ROB is divided in two regions: the region occupied by speculative instructions and the region occupied by nonspeculative instructions. Cherry is able to release registers and load/store queue entries early in the ROB area not subject to misspeculation, using the checkpoint to provide precise exception handling. On the other hand, the instructions belonging to the region subject to misspeculation (such as speculative instructions after a nonresolved branch prediction) still depend on the ROB to recover the correct state in the case of misspeculation, and thus they are not able to release their corresponding resources.

Runahead execution uses checkpointing as well. Runahead processing was first proposed by Dundas and Mudge [1997] for in-order processors. This work generates highly accurate data prefetches by pre-executing future instructions under a cache miss. Mutlu et al. [2003] extend this proposal for out-of-order processors. They create a checkpoint of the architectural state when the head of the ROB is reached by a load that has missed in the second-level cache.

In addition, the processor starts executing instructions in a special mode that invalidates the result of the load and all the dependent instructions. When the load instruction actually completes, the processor returns to the normal mode, restoring the checkpoint. The results obtained during the first execution are not reused. However, this first execution provides useful knowledge, such as accurate data and instruction prefetches, which improves the performance during the second execution.

Finally, the checkpoint processing and recovery (CPR) architecture [Akkary et al. 2003] propose similar mechanisms to those described by Cristal et al. [2002b]. The main objective of the CPR architecture is to implement large instruction window processors. The CPR architecture uses a multicheckpointing mechanism, similar to ours, in order to checkpoint hard-to-predict branches. A checkpoint is taken when a hard-to-predict branch reaches the decode stage, enabling an early release of instructions and physical registers.

## 4. INSTRUCTION QUEUE MANAGEMENT

At the same time that instructions are inserted in the ROB, they are also inserted in their corresponding instruction queues (IQs). The instructions should wait in the IQs until they are issued for execution. However, as shown in Section 2.2, most instructions are blocked in the IQs, taking a long time to get issued for execution because they should wait for the results produced by long-latency instructions. Maintaining those blocked-long instructions in the IQs just takes away issue slots from other instructions that would be executed more quickly. This fact not only restricts the exploitable instruction-level parallelism, but also greatly increases the probability of stalling the processor due to a full IQ, which severely limits the achievable performance.

In the context of a processor able to support thousands of in-flight instructions, blocked-long instructions cause a serious scalability problem, since the IQs should have enough entries to keep all the instructions that are waiting for issue. The IQs are critical processor structures, and thus such a large increase in their size will definitely affect the processor cycle time [Palacharla et al. 1997]. This problem can be overcome by using multilevel IQs for taking advantage of the different waiting times of the instructions in the IQs. First, blocked-long instructions should be detected. Then, they are removed from the IQs, delegating their handling to slower, but larger and less complex structures. Later, when the blocked-long instructions become ready, they are moved back to the IQs.

### 4.1 Slow-Lane Instruction Queue

Kilo-instruction processors rely on a unified mechanism that combines multicheckpointing with techniques to efficiently manage the IQs and the physical register file. In particular, our multicheckpointing mechanism enables a smart multilevel IQ management technique [Cristal et al. 2004a], which is shown in Figure 9. Renamed instructions are inserted in both the ROB and the conventional IQs (step 1). Our mechanism detects the instructions that will take a long time to get issued for execution because they depend on a long-latency

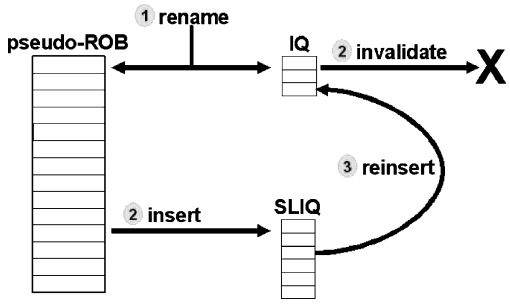


Fig. 9. The Slow-lane instruction queue.

memory operation. The long-latency memory operation detection is done by the hardware devoted to check the second-level cache tags, that is, there is no need to wait until the cache access is fully resolved.

The detected long-latency instructions are moved to a secondary buffer when they reach the head of the pseudo-ROB. We call this simple FIFO-like secondary buffer slow-lane instruction queue (SLIQ). Instructions stay in SLIQ until the corresponding long-latency memory operation is resolved. In order to simplify the implementation, this instruction movement is actually done invalidating the instructions in the IQs and inserting them in the SLIQ from the pseudo-ROB (step 2). Finally, when the long-latency operation that blocked the instructions in the SLIQ is resolved, the dependent instructions are moved from the SLIQ to the IQs (step 3). All physical registers have a bit associated with that states when the corresponding value will be ready in a short period of time. In order to detect which instructions should be moved from the SLIQ to the IQ, we just need to check these bits.

Since the SLIQ itself is a simple FIFO-like structure, it can be implemented as a bank-interleaved buffer. Moreover, the SLIQ is not on the critical path, and thus it can have thousands of entries without harming the processor cycle time. Therefore, our SLIQ mechanism allows us to effectively implementing the functionality of large IQs, making it possible to support a high number of in-flight instructions without scaling-up the IQs.

The presence of the pseudo-ROB is beneficial for detecting whether an instruction will be executed shortly or will consume resources for a long time. Our technique delays the decision of which instructions will wait during a long time until instructions are extracted from the pseudo-ROB. If a particular load instruction arrives at the end of the pseudo-ROB without hitting in the first- or second-level caches, we consider it a long-latency instruction. Thus, the decision is delayed until the waiting time of the instruction can be effectively known, increasing the detection accuracy.

Figure 10 shows an example of our SLIQ mechanism. The primary source of long-latency instructions is loads that miss in the second-level cache. Let us suppose that the load instruction at instant $t0$ is a long-latency instruction. Any instruction that depends on it will also be considered a long-latency instruction. Once taken out of the pseudo-ROB, a simple mechanism computes the dependencies on this load. Our mechanism uses a bit mask where each bit

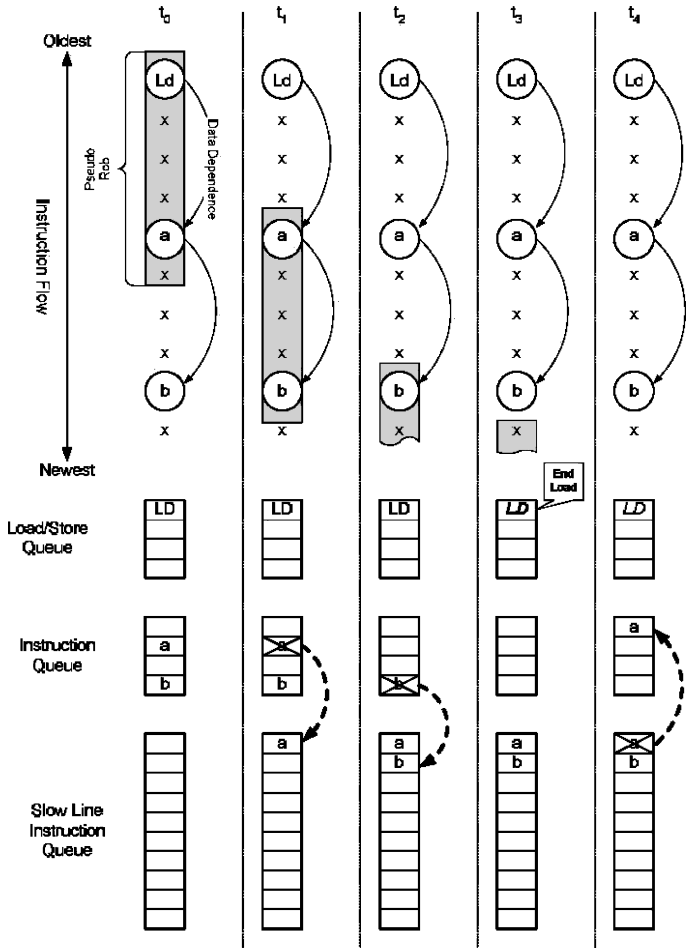


Fig. 10. Example of the SLIQ mechanism.

is associated with a logical register. This bit mask starts all cleared except for the bit corresponding to the long-latency load destination register. When an instruction is extracted from the pseudo-ROB, it incorporates its destination register into the bit mask if it consumes any register from it, since this instruction is dependent on the load and so will be any instruction that depends on it. On the other hand, a bit is cleared if a nondependent instruction redefines the corresponding register.

The dependent instructions computed in this way are invalidated in the general-purpose IQs and stored in order in the SLIQ. Although instructions are actually inserted in the SLIQ from the pseudo-ROB, we assume here that they are moved from the IQs to the SLIQ in order to simplify the example. This can be seen in Figure 10 at instants *t1* and *t2*, when instructions a and b, which depend on the long-latency load, are moved into the SLIQ. As a consequence, the entries associated with the long-latency instructions are freed, and thus can be used by short-latency instructions.

The instructions in the SLIQ wait for the resolution of the long-latency load miss. In order to simplify the wakening of these instructions, the destination register of the long-latency load is associated with the corresponding entry of the SLIQ. When this register gets its value, the load has been resolved, and thus the dependent instructions should be moved back to the IQs. These instructions are invalidated in the SLIQ and inserted into the corresponding IQs.

At instant *t3* in Figure 10 we can see that the long-latency load is resolved. The SLIQ is scanned in order from the entry corresponding to the load and the dependent instructions are inserted back into the window. However, it could happen that a second long-latency load is resolved while the instructions dependent on the first one are still being processed. On one hand, if the new load is younger than the instructions being processed, it will be found by the awakening mechanism. After that, the mechanism could continue, placing the instructions dependent on any of the two loads in the IQs. On the other hand, if the new load is older than the instructions being processed, it will not be found by the awakening mechanism. Thus, the awakening process is canceled and a new awakening process is started from the second load.

### 4.2 Related Work

Some previous studies have proposed multilevel IQs. The hierarchical scheduling window proposed by Brekelbaum et al. [2002] divides the IQ into a large but slow IQ and a small but fast IQ. Each one is independent and has its own wake-up and select logic. All the fetched and decoded instructions are inserted in the slow IQ. This slow IQ keeps all instructions whose operands become ready soon, which are considered latency tolerant. However, when the oldest instructions in the IQ are determined not to have nonready operands, they are supposed to be in the critical path and moved into the fast IQ. This heuristic ensures that the instructions in the fast IQ are highly interdependent and latency critical. On the other hand, the slow window contains latency-tolerant instructions, facilitating the implementation of a very large IQ and allowing the extraction of far-flung instruction-level parallelism. Unlike our SLIQ, this technique is not associated with any checkpoint mechanism. In addition, all the critical instructions are penalized, since they are introduced in the slow IQ before being detected. Both SLIQ and the WIB does not suffer from this problem, since the instructions are first inserted in the fast structure instead of in the slow one.

Lebeck et al. [2002] add a wait bit to each physical register. This bit indicates when the corresponding register is waiting for a long-latency memory operation to generate its contents. The bit is initially set by a cache miss, and is later propagated to the physical registers associated with the dependent instructions. The wake-up and select logic work as usual, but the instructions having at least one of their wait bits set are also considered to get issued. However, instead of being sent to a functional unit, they are placed in a waiting instruction buffer (WIB), freeing the IQ entry. Therefore, the WIB contains all the instructions directly or indirectly dependent on a cache miss, where they reside until the miss is resolved. A set of bit vectors are used to indicate which WIB entries depend

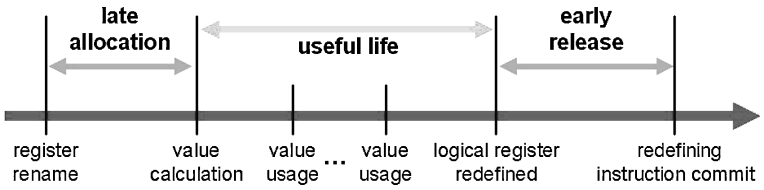


Fig. 11. Life cycle of a physical register.

on a specific cache miss. Each vector in this matrix is allocated when a load misses, and thus each load should keep an index to the corresponding vector. When a long-latency load finishes execution, the dependent instructions are reinserted in the IQs, sharing the same bandwidth with those new instructions that are decoded and dispatched. The dispatch logic prioritizes the instructions reinserted from the WIB in order to assure forward progress.

This mechanism is similar in spirit to our SLIQ. However, since the WIB only deals with the scalability of the general-purpose IQs, a direct performance comparison against our proposal is impossible. While the WIB requires very large and unfeasible ROB, physical register file, and load/store queue, our SLIQ is a component of a whole design focused on dealing with the scalability problems of all the critical processor structures. Recently, the continual flow pipelines (CFP) architecture [Srinivasan et al. 2004] has presented an efficient implementation, oriented to the Pentium 4 pipeline, of a two-level instruction queue called slice data buffer (SDB), which is similar to the WIB and the SLIQ. Like our work, the CFP architecture is not just focused on the scalability of the IQs, but on the scalability of all the processor structures.

## 5. EPHEMERAL REGISTERS

In order to reduce the number of physical registers needed, the kilo-instruction processor relies on the different behaviors observed during the life cycle of a physical register, which is shown in Figure 11. As we described in Section 2.3, a large portion of the allocated physical registers are blocked because the corresponding instructions are waiting for the execution of long-latency operations. This is due to the fact that physical registers are assigned early in the pipeline for keeping track of the register dependencies during the rename phase. Another high percentage of the allocated physical registers is dead, that is, they are no longer in use. They are still allocated because an instruction redefining the corresponding logical register has not yet committed, and thus, the processor cannot assure that the register contents will not be read again.

### 5.1 Improving Physical Register Utilization

Wasting physical registers due to long-time blocked instructions can be avoided by using techniques for late register allocation. Monreal et al. [1999] describe a technique that allows for a late allocation of physical registers. Instead of assigning a physical register to each renamed instruction, this technique assigns a virtual tag. These virtual registers are used to keep track of the rename dependencies, making unnecessary to assign a physical register to an instruction

until it starts execution. Therefore, this technique does not allocate a physical register until it is strictly necessary for storing the produced value. A similar technique has been included in the CFP architecture [Srinivasan et al. 2004]. CFP allows the release of destination registers of instructions dependent upon a miss, prior to these instructions entering the SDB. These instructions will subsequently re-acquire registers when they re-enter the pipeline. In addition, this technique is enhanced by making it possible to release completed source registers.

Dead registers can also be avoided by using techniques for early register release. Moudgill et al. [1993] propose to use a counter associated with each register for keeping track of the unexecuted instructions that read the register contents. An instruction that should read the value increments the counter when it is fetched and decrements the counter when it effectively reads the register. If there is an instruction that redefines the corresponding logical register, the processor does not need to wait until the instruction commits to releasing the physical register. It can be released as soon as the counter arrives to zero.

### 5.2 Implementing Ephemeral Registers

Kilo-instruction processors go one step further. They combine both techniques for early register release and late register allocation with our multicheckpointing mechanism, leading to an aggressive register recycling technique that we call ephemeral registers [Cristal et al. 2003c; Martínez et al. 2003]. As far as we know, this is the first proposal that simultaneously and in a coordinated manner supports the three techniques. This combination allows the processor to manage registers aggressively, effectively dissociating register release from instruction retirement, and register allocation from instruction renaming. As a result, our technique shrinks the lifetime of a physical register to its useful period in the pipeline, making it possible to support thousands of in-flight instructions without requiring an unfeasible large physical register file.

The implementation of ephemeral registers in kilo-instruction processors replaces the future free bits mechanism described in Section 3.2. Instead of assigning a physical register to each renamed instruction, our mechanism assigns a virtual tag. This technique implements late register allocation, since a physical register is only assigned when the instruction is issued for execution. In order to combine late register allocation with early register release, each virtual tag has an associated counter, along the lines of Moudgill et al. [1993]. This counter indicates how many instructions will read the contents of the virtual register. The counter is incremented by each renamed instruction that reads the virtual register and decremented when the reader instruction is issued for execution. After the corresponding logical register is redefined, the virtual register and its associated physical register (if there is one) can be released if the counter reaches zero and the contents of the register have already been written.

In the case of an exception or a misspeculation, the state of the counters should be recovered. The instructions already executed have no influence on the state of the counters because they have incremented the corresponding counters during rename and they have decremented them again after issue.

Therefore, it is only necessary to take into account the instructions in the IQs, since they have incremented the corresponding counters but they have not already decremented them.

Our architecture cannot use the pseudo-ROB to obtain the required information, since it is possible that an instruction is in the IQs but it has already been retired from the pseudo-ROB. In order to overcome this problem, we use a straightforward technique. After an exception or a misspeculation, the processor pipeline is flushed. However, the instructions in the IQs are not invalidated, but marked as no-operations. The instructions will be issued as usual, but they will not be executed, since their only purpose is to decrement the corresponding counters. This process is not time-critical because it can be performed just after the pipeline flush.

The instructions in the SLIQ cause the same problem, and we solve it in a similar way. An FIFO structure like the SLIQ has two pointers: the head pointer (HP), which points to the older instruction in the SLIQ, and the tail pointer (TP), which indicates where the long-latency instructions retired from the pseudo-ROB should be inserted. In the case of an exception or a misspeculation, the correct processor state is recovered using a checkpoint and the wrong instructions are flushed from the pipeline. In particular, the SLIQ is flushed by moving back the TP up to the point where the checkpoint was taken.

Now, the instructions between the old and the new positions of the TP should be reinserted as no-operations in the IQs. In order to do this, our technique uses two additional pointers: the old tail pointer (OTP), which points to the position of the TP just before the SLIQ flush, and the recovery pointer (RP), which points to the first instruction that should be recovered, that is, the current position of the TP. The RP is advanced each cycle, reinserting the no-operation instructions in the IQs. During the recovery, the TP cannot move forward the RP, since the new long-latency instructions inserted in the SLIQ must not reuse the entries assigned to old instructions that have not been already recovered. However, once again, this process is not time critical, since RP frees entries during the cycles in which the SLIQ cannot be filled because there are no instructions to be inserted after the pipeline flush. The recovery process ends when the RP reaches the OTP.

Finally, there are two possible sources of deadlock. The first is related to the concept of virtual-physical registers [Monreal et al. 1999]. We avoid this deadlock reserving a number of physical registers equal to the number of logical registers plus one to be used by the older instructions in the SLIQ. The other deadlock situation can happen if the instruction queue gets filled and therefore blocks reinsertion from the SLIQ. In order to avoid this, we make sure that at least one entry is always available for instructions to be reinserted from the SLIQ.

## 6. IMPLEMENTING LARGE LOAD/STORE QUEUES

Load and store instructions are inserted in the load/store queues at the same time they are inserted in the ROB. The main objective of these queues is to guarantee that all load and store instructions perform in agreement with program

order. This requires a complex memory disambiguation logic that compares the effective address of each memory operation with the addresses of all the previous in-flight memory operations. We model a memory disambiguation mechanism similar to the one implemented in the Power4 processor [Tendler et al. 2001]. This model calculates the effective address of a store instruction when the corresponding register is ready, that is, it does not wait for the data register. If a load instruction is issued and there is an older instruction that will write to the same memory location, the store result is forwarded to the load. If the result is still not ready, the load is rejected and reissued again, waiting for the result. Using this model, the amount of replay traps caused is relatively low for numerical applications (less than one per million of executed instructions).

However, as shown in Section 2.4, maintaining a high number of in-flight instructions involves an increase in the number of loads and stores that should be taken into account, which can make the load/store queue a true bottleneck both in latency and power. Several techniques have been proposed to overcome this problem. Although we do not propose to choose one or another, any of the mechanisms described in this section can be implemented in a kilo-instruction processor. The scalability problem of the load queue can be solved using techniques for early release of load instructions. These techniques were first proposed by Cristal et al. [2002b] and by Martínez et al. [2002]. In addition, several solutions have recently been proposed for dealing with the scalability problem of the store queue.

Akkary et al. [2003] propose a hierarchical store queue organization able to buffer a large number of store instructions and perform critical disambiguation processes without degrading cycle time. A fast and small first-level store queue (L1), similar to the store queues in current microprocessors, holds the last executed stores. When a new store appears, it is inserted in the L1. If it is full, the older store is removed, making space for the new store. The removed store is inserted into a backing second-level store queue (L2), where it remains until commit. As can be expected, the L2 is larger and slower than the L1. In addition, the L2 has a membership test buffer (MTB) associated with it. This MTB is an untagged table used to predict if a store instruction is buffered in the L2. When a load instruction is issued, the L1 and the MTB are accessed in parallel. If the load misses both the L1 and the MTB, the data is forwarded to the load from memory. If the load hits the L1, the data is forwarded to the load from the L1. Finally, if the load misses the L1, but hits the MTB, the L2 is accessed. If there is a hit, the data is forwarded to the load. However, if there is a miss, the load has been penalized unnecessarily, since the data should be forwarded from memory. Overall, this mechanism provides a performance close to a store queue with thousands of entries, while requiring a few hundreds.

Park et al. [2003] describe three techniques to scale the load/store queue. First, a store-set predictor [Chrysos and Emer 1998] is used to predict the matches between loads and stores. A load will search the store queue only when the predictor states that there is a potentially dependent store in the queue. This technique achieves an important reduction in the search bandwidth demand on the store queue. Second, in the context of shared-memory multiprocessors, a small load buffer is used to keep only the out-of-order issued load instructions.

When a load executes, it should search for out-of-order issued loads to assure memory consistency. However, instead of searching the load queue, it searches the load buffer, which is much smaller, achieving an important reduction in the search bandwidth demand on the load queue. Third, the load/store queue is segmented into multiple smaller queues, which are connected in a chain. These segments are treated as a pipeline, turning the load/store queue into a variable latency structure whose capacity can be increased without causing a negative impact on the access latency.

Sethumadhavan et al. [2003] improve the load/store queue scalability by applying approximate hardware hashing. This technique implements a hash table with Bloom filters [Bloom 1970] where the address of load and store instructions is hashed to a particular bit inside a table. If the bit is set, there is a likely address match with a previous load or store instruction and the load/store queue should be searched. In contrast, if the bit is not set, there cannot be an address match and the load/store queue does not need to be searched. In addition, multiple Bloom filters can be used to separate load/store queue partitions, reducing the number of partitions that should be looked up when a queue search is necessary. These techniques significantly reduce the access latency and power consumption required by the load/store queue, alleviating this scalability bottleneck.

Finally, Cain and Lipasti [2004] solve the load queue scalability problem by completely eliminating it. Data dependencies and memory consistency are enforced by re-executing load instructions in program order before commit, following a value-based memory-ordering approach. The set of loads that must be re-executed is filtered using several heuristics, which makes it possible to sacrifice only a negligible amount of performance due to the re-execution of loads.

## 7. PERFORMANCE EVALUATION

In this section, we evaluate the performance achievable by kilo-instruction processors. We focus on numerical applications. As shown in Section 1, increasing the number of in-flight instructions provides high performance improvement for these applications even in the presence of large memory access latencies.

Our data is obtained using a custom execution-driven simulator. Table I shows the setup of our simulated baseline architectures. Our baseline is an out-of-order superscalar processor able to maintain 128 in-flight instructions. We also present data for a limit baseline, an unfeasible processor able to maintain up to 4096 in-flight instructions. The benchmark suite used for all the experiments is the SPEC2000fp, which were compiled to take advantage of software prefetching. All benchmarks have been simulated 300 million representative instructions. To find the most representative execution segment, we have analyzed the distribution of basic blocks as described by Sherwood et al. [2001].

Figure 12 shows the performance of a processor using our out-of-order commit and SLIQ techniques. In order to analyze the behavior of these two techniques, we assume unbounded physical register file and load/store queue. The

Table I. Experimental Setup

| Element | Baseline 128 | Baseline 4096 |
|---|---|---|
| Simulation Strategy | Execution-driven | Execution-driven |
| Issue policy | Out-of-order | Out-of-order |
| Fetch/commit width | 4/4 | 4/4 |
| Branch predictor | 16 K-entry GShare | 16 K-entry GShare |
| Branch penalty | 10 cycles | 10 cycles |
| I-L1 size | 32 Kb 4-way, 32 byte line | 32 Kb 4-way, 32 byte line |
| I-L1 latency | 2 cycles | 2 cycles |
| D-L1 size | 32 Kb 4-way, 32 bytes line | 32 Kb 4-way, 32 bytes line |
| D-L1 latency | 2 cycles | 2 cycles |
| L2 size | 512 Kb 4-way, 64 bytes line | 512 Kb 4-way, 64 bytes line |
| L2 latency | 10 cycles | 10 cycles |
| TLB | 64 entries, 4 way, 8 KB page 30 cycles | 64 entries, 4 way, 8KB page 30 cycles |
| Memory latency | 1000 cycles | 1000 cycles |
| Memory ports | 2 | 2 |
| Reorder buffer | 128 entries | 4096 entries |
| Load/store queue | 128 entries | 4096 entries |
| Integer queue | 128 entries | 4096 entries |
| FP queue | 128 entries | 4096 entries |
| Integer register | 128 | 4096 |
| FP registers | 128 | 4096 |

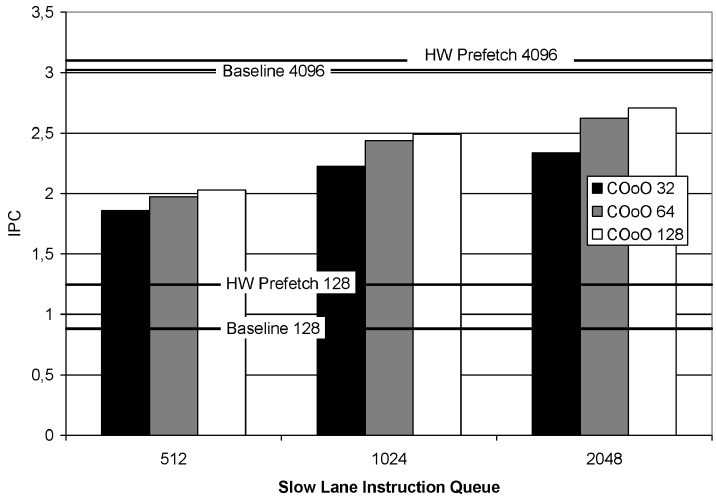


Fig. 12. Performance achieved when our out-of-order commit is combined with the SLIQ.

figure presents three groups of three bars each, as well as four reference lines across the figure. The bars COoO 32, 64, and 128 correspond to our processor organization having a pseudo-ROB and IQs of 32, 64, and 128 entries respectively. The reference lines correspond to our baseline processor, which has 128-entry ROB and IQs, and to our unrealistic limit baseline processor, which has 4096-entry ROB and IQs. Both baselines are presented not using and using hardware stride prefetch (labeled *HW Prefetch* in the figures). We have implemented an

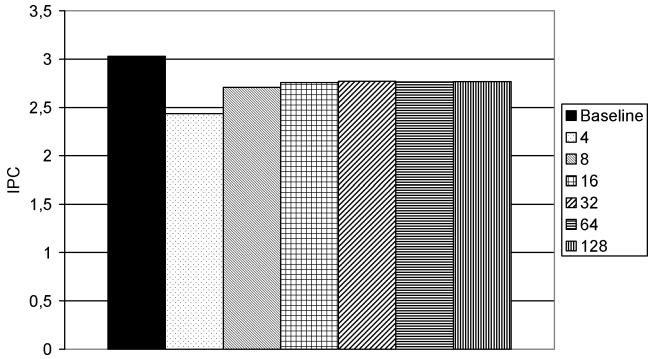


Fig. 13. Sensitivity of commit mechanism to the amount of available checkpoints (2048 entry IQ and 2048 physical registers).

aggressive prefetcher, placed between the second-level cache and main memory. It supports up to 16 streams containing 16 lines each. Every stream is able to take into account up to four different strides during the learning period. In contrast, our kilo-instruction processor does not use any hardware prefetching mechanism.

Each set of bars groups simulations according to the amount of SLIQ entries. Through simulation, we have found that the percentage of executed instructions that depend on long-latency memory operations ranges from 20% to 30%. Therefore, only this percentage of executed instructions needs to be moved to the SLIQ and back. As a result, even the simplest setup, having 32-entry pseudo-ROB and IQs, as well as 512-entry SLIQ, outperforms by more than 50% the lower-line baseline processor using prefetch, which has larger critical structures (128-entry ROB and IQs). If we consider more complex setups, the difference in performance grows up to more than 100%.

Note that our mechanism always suffers a penalty with respect to the limit baseline. Nevertheless, our proposal is significantly close to this unrealistic baseline with a fraction of the cost. Although having IQs with thousands of entries is impossible due to the current technology, having a 2048-entry SLIQ is possible because it does not need complex wake-up logic and its selection logic is very simple. Instructions are reinserted in the IQs at a pace of four instructions per cycle (the processor width). The instructions provided by the SLIQ are prioritized over the new instructions coming from renaming. Moreover, the SLIQ is not on the critical path and its access latency can be tolerated. We have analyzed the sensitivity of the SLIQ with respect to this latency and we have found that even 12-cycle latency only produces a negligible performance slowdown.

In all the evaluated cases, the processor is able to maintain a maximum of eight in-flight checkpoints. We have found that eight checkpoints are enough to achieve almost all the potential performance. Figure 13 shows the performance obtained varying the total number of checkpoints. The limit bar represents the performance obtained when considering a 4096-entry ROB, which is microarchitecturally unfeasible. As can be seen, having eight checkpoints produces just

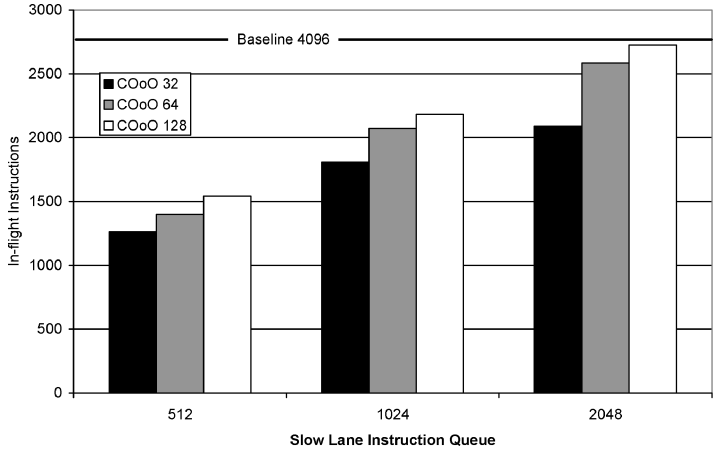


Fig. 14. Average number of in-flight instructions.

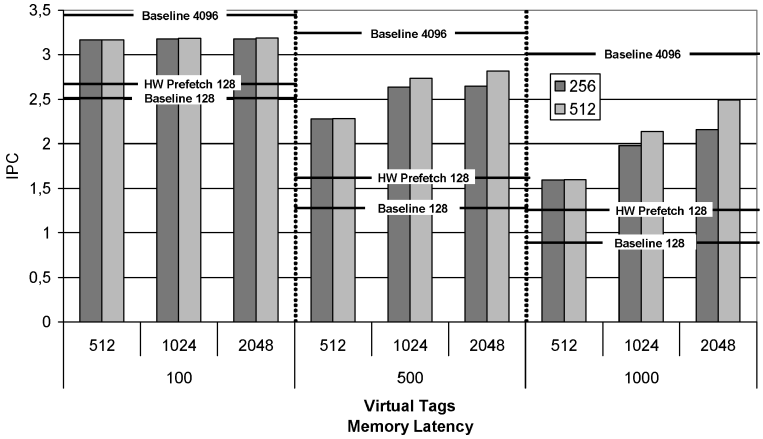


Fig. 15. Kilo-instruction processor performance.

a 9% performance slowdown against the unrealistic limit machine. A higher number of checkpoints does not provide significant benefit.

Figure 14 shows the average number of in-flight instructions for both our proposal and the limit baseline. The bars and the reference line correspond to the same setup as Figure 12. It becomes clear that our mechanism is effectively allowing for a very big amount of in-flight instructions. The more aggressive setups are close to the unfeasible limit machine, which is able to support up to 4096 in-flight instructions.

Finally, Figure 15 shows the performance of our out-of-order commit and SLIQ mechanisms when combined with the ephemeral registers technique. This data provides insight about the performance achievable by kilo-instruction processors. The figure is divided into three zones, each of them comprising the results for 100, 500, and 1000 cycles of main memory access latency. Each zone is composed of three groups of two bars, corresponding to 512, 1024, and 2048

virtual registers or tags [Monreal et al. 1999]. The two bars of each group represent the performance using 256 or 512 physical registers. In addition, each zone of the figure has three lines that represent the performance of the lower-line and upper-line baseline setups (we have not included the limit line using prefetching because it is very close to the limit line not using it).

The main observation is that kilo-instruction processors provide important performance improvements over the lower-line baseline processor, even when it uses an aggressive hardware stride prefetcher. Using 2048 virtual tags, the kilo-instruction processor is almost twice faster than the lower-line baseline using prefetch when the memory access latency is 500 cycles or higher. The performance variation ranges from no improvement in *172.mgrid* or just 18% speedup in *177.mesa* to 4× improvement in *188.ammp*, 5× improvement in *173.applu*, or even 10× improvement in *189.lucas*.

In addition, our results show that the kilo-instruction processor is an effective way of approaching the unimplementable upper-line baseline processor in an affordable way. However, there is still room for improvement. The distance between the kilo-instruction processor performance and the upper-line baseline is higher for larger memory access latencies. This fact causes that, although the performance results of the more aggressive setups nearly saturate for a memory access latency of 100 or 500 cycles, the growing trend is far from saturating when the memory access latency is 1000 cycles. This trend suggests that a more aggressive machine, able to support a higher number of in-flight instructions, will provide even a better performance.

## 8. CONCLUSIONS

Maintaining a high amount of in-flight instructions is an effective mean for overcoming the memory wall problem. However, increasing the number of in-flight instructions requires upsizing the critical processor structures, which is impractical due to area, power consumption, and cycle time limitations.

Kilo-instruction processors overcome these limitations by smartly using the available resources. In this paper, we have provided quantitative evidence that the critical processor structures are severely underutilized. We propose a set of efficient techniques that allow changing the way current processors use the critical structures. Although these techniques increase the complexity of the processor design, they make it possible to scale up the total number of in-flight instructions. Having thousands of in-flight instructions, a kilo-instruction processor is able to provide high performance improvements in the presence of large memory access latencies, which compensates the added complexity.

Therefore, we strongly believe that kilo-instruction processors are an efficient mechanism for dealing with future memory latencies. In this paper, we have shown that the design of a kilo-instruction processor is feasible. We are currently refining the design in order to reduce the complexity and the power consumption required by our approach. Moreover, we are analyzing the synergy between kilo-instruction processors and other architectures, such as multithreaded processors. We are also working on kilo-instruction multiprocessors [Galluzzi et al. 2004]. Kilo-instruction processors constitute a flexible paradigm

that can be combined with other architectures to improve their capabilities and boost the processor performance, creating a great amount of new and appealing ideas for future research.

## ACKNOWLEDGMENTS

This research has been supported in part by CICYT grant TIN-2004-07739-C02-01 (Valero), NSF grant CCF-0429922 (Martínez), the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HIPEAC), and CEPBA. We would like to thank Srikanth Srinivasan, Ravi Rajwar, and Haitham Akkary for their worthwhile comments to this work. Thanks also go to Francisco Cazorla, Ayose Falcón, Marco Galluzzi, Robén González, Josep Llosa, Daniel Ortega, Alex Pajuelo, Miquel Pericas, and Tanausú Ramírez for their contribution to the kilo-instruction processors.

## REFERENCES

AKKARY, H., RAJWAR, R., AND SRINIVASAN, S. T. 2003a. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proceedings of the 36th International Symposium on Microarchitecture* (San Diego, CA). 423–434.

AKKARY, H., RAJWAR, R., AND SRINIVASAN, S. T. 2003b. Checkpoint processing and recovery: An efficient, scalable alternative to reorder buffers. *IEEE Micro 23*, 6, 11–19.

BAER, J.-L. AND CHEN, T.-F. 1991. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of Supercomputing'91* (Albuquerque, NM). 176–186.

BREKELBAUM, E., RUPLEY, J., WILKERSON, C., AND BLACK, B. 2002. Hierarchical scheduling windows. In *Proceedings of the 35th International Symposium on Microarchitecture* (Istanbul, Turkey). 27–36.

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

CAIN, H. W. AND LIPASTI, M. H. 2004. Memory ordering: A value-based approach. In *Proceedings of the 31st International Symposium on Computer Architecture* (Munich, Germany). 90–101.

COLLINS, J. D., TULLSEN, D. M., WANG, H., AND SHEN, J. P. 2001. Dynamic speculative precomputation. In *Proceedings of the 34th International Symposium on Microarchitecture* (Austin, TX). 306–317.

CHAPELL, R., STARK, J., KIM, S., REINHARDT, S., AND PATT, Y. N. 1999. Simultaneous subordinate microthreading (SSMT). In *Proceedings of the 26th International Symposium on Computer Architecture* (Atlanta, GA). 186–195.

CHRYSOS, G. Z. AND EMER, J. S. 1998. Memory dependence prediction using store sets. In *Proceedings of the 25th International Symposium on Computer Architecture* (Barcelona, Spain). 142–153.

CRISTAL, A., VALERO, M., GONZALEZ, A., AND LLOSA, J. 2002a. White paper: Grant proposal to Intel-MRL in January 2002. Universitat Politècnica de Catalunya, Barcelona, Spain.

CRISTAL, A., VALERO, M., GONZALEZ, A., AND LLOSA, J. 2002b. *Large Virtual ROBs by Processor Checkpointing*. Technical Report UPC-DAC-2002-39, Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, Barcelona, Spain. Also submitted to the *35th International Symposium on Microarchitecture*, in 2002.

CRISTAL, A., MARTÍNEZ, J. F., LLOSA, J., AND VALERO, M. 2003a. A case for resource-conscious out-of-order processors. *IEEE TCCA Computer Architecture Letters 2*.

CRISTAL, A., ORTEGA, D., LLOSA, J., AND VALERO, M. 2003b. Kilo-instruction processors. In *Proceedings of the 5th International Symposium on High-Performance Computing* (Tokyo, Japan). 10–25. Keynote paper.

CRISTAL, A., MARTÍNEZ, J. F., LLOSA, J., AND VALERO, M. 2003c. *Ephemeral Registers with Multicheckpointing*. Technical Report UPC-DAC-2003-51, Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, Barcelona, Spain.

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

CRISTAL, A., SANTANA, O. J., AND VALERO, M. 2004b. Maintaining thousands of in-flight instructions. In *Proceedings of the Euro-Par Conference* (Pisa, Italy). Keynote paper.

DUBOIS, M. AND SONG, Y. 1998. *Assisted Execution*. Technical Report CENG 98–25, Department of EE-Systems, University of Southern California, Los Angeles, CA.

DUNDAS, J. AND MUDGE, T. 1997. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 9th International Conference on Supercomputing* (Vienna, Austria). 68–75.

GALLUZZI, M., PUENTE, V., CRISTAL, A., BEIVIDE, R., GREGORIO, J. A., AND VALERO, M. 2004. A first glance at kilo-instruction based multiprocessors. In *Proceedings of the 1st Conference on Computer Frontiers* (Ischia, Italy).

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

JOSEPH, D. AND GRUNWALD, D. 1997. Prefetching using Markov predictors. In *Proceedings of the 24th International Symposium on Computer Architecture* (Denver, CO). 252–263.

KLAIBER, A. AND LEVI, H. 1991. An architecture for software-controlled data prefetching. In *Proceedings of the 18th International Symposium on Computer Architecture* (Toronto, Canada). 53–53.

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

MARTÍNEZ, J. F., RENAU, J., HUANG, M., PRVULOVIC, M., AND TORRELLAS, J. 2002. Checkpointed early resource recycling in out-of-order microprocessors. In *Proceedings of the 35th International Symposium on Microarchitecture* (Istanbul, Turkey). 3–14.

MARTÍNEZ, J. F., CRISTAL, A., VALERO, M., AND LLOSA, J. 2003. *Ephemeral Registers*. Technical Report CSL-TR-2003-1035, Cornell Computer Systems Lab, Ithaca, NY.

MONREAL, T., GONZALEZ, A., VALERO, M., GONZALEZ, J., AND VIÑALS, V. 1999. Delaying physical register allocation through virtual-physical registers. In *Proceedings of the 32nd International Symposium on Microarchitecture* (Haifa, Israel). 186–192.

MOWRY, T., LAM, M., AND GUPTA, A. 1992. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, MA). 62–73.

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* (Austin, TX). 202–213.

MUTLU, O., STARK, J., WILKERSON, C., AND PATT, Y. N. 2003a. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture* (Anaheim, CA). 129–140.

MUTLU, O., STARK, J., WILKERSON, C., AND PATT, Y. N. 2003b. Runahead execution: An effective alternative to large instruction windows. *IEEE Micro 23*, 6, 20–25.

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

PARK, I., OOI, C., AND VIJAYKUMAR, T. 2003. Reducing design complexity of the load/store queue. In *Proceedings of the 36th International Symposium on Microarchitecture* (San Diego, CA). 411–422.

ROTH, A. AND SOHI, G. S. 2001. Speculative data-driven multithreading. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture* (Nuevo Leone, Mexico). 37–48.

SETHUMADHAVAN, S., DESIKAN, R., BURGER, D., MOORE, C., AND KECKLER, S. 2003. Scalable hardware memory disambiguation for high ILP processors. In *Proceedings of the 36th International Symposium on Microarchitecture* (San Diego, CA). 399–410.

SHERWOOD, T., PERELMAN, E., AND CALDER, B. 2001. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (Barcelona, Spain). 3–14.

SMITH, A. 1982. Cache memories. *Computing Surveys 14*, 3, 473–530.

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

SOHILIN, Y., LEE, J., AND TORRELLAS, J. 2002. Using a user-level memory thread for correlation prefetching. In *Proceedings of the 29th International Symposium on Computer Architecture* (Anchorage, AK). 171–182.

SRINIVASAN, S. T., RAJWAR, R., AKKARY, H., GANDHI, A., AND UPTON, M. 2004. Continual flow pipelines. In *Proceedings of the 11st International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, MA). 107–119.

TENDLER, J. M., DODSON, S., FIELDS, S., LE, H., AND SINHAROY, B. 2001. IBM @server POWER4 System Microarchitecture. IBM Technical White Paper. IBM Server Group.

ZILLES, C. AND SOHI, G. S. 2001. Execution-based prediction using speculative slices. In *Proceedings of the 28th International Symposium on Computer Architecture* (Göteborg, Sweden). 2–13.

Received September 2004; revised November 2004; accepted November 2004