# Early Register Release for Out-of-Order Processors with Register Windows

Eduardo Quinones
Departament d'Arquitectura de Computadors
Universitat Politecnica de Catalunya
equinone@ac.upc.edu

Joan-Manuel Parcerisa
Departament d'Arquitectura de Computadors
Universitat Politecnica de Catalunya
equinone@ac.upc.edu

Antonio Gonzalez
Departament d'Arquitectura de Computadors
Universitat Politecnica de Catalunya
Intel Barcelona Research Center
Intel Labs
antonio.gonzalez@intel.com

## Abstract

*Register windows is an architectural technique that reduces memory operations required to save and restore registers across procedure calls. Its effectiveness depends on the size of the register file. Such register requirements are normally increased for out-of-order execution because it requires registers for the in-flight instructions, in addition to the architectural ones. However, a large register file has an important cost in terms of area and power and may even affect the cycle time. In this paper we propose two early register release techniques that leverages register windows to drastically reduce the register requirements, and hence reduce the register file cost. Contrary to the common belief that out-of-order processors with register windows would need a large physical register file, this paper shows that the physical register file size may be reduced to the bare minimum by using this novel microarchitecture. Moreover, our proposal has much lower hardware complexity than previous approaches, and requires minimal changes to a conventional register window scheme. Performance studies show that the proposed technique can reduce the number of physical registers to the same number as logical registers plus one (minimum number to guarantee forward progress) and still achieve almost the same performance as un unbounded register file.*

## 1 Introduction

Register windows is an architectural technique that reduces the loads and stores required to save and restore registers across procedure calls by storing the local variables of multiple procedure contexts in a large architectural register file. When a procedure is called, it maps its context to a set of consecutive new architected registers, called a register window. Through a simple runtime mechanism, the compiler-defined local variables are then renamed to these windowed registers.

If there are not enough architectural registers to allocate new contexts, local variables from caller procedures are saved to memory and their associated registers are freed for the new contexts. When the saved contexts are needed they are restored in the register file. These operations are typically referred to as *spill* and *fill*. SPARC [4] and Itanium [8] are two commercial architectures that use register windows.

The effectiveness of the register windows technique depends on the size of the architectural register file because the more registers it has, the less number of spills and fills are required [18]. Analogously, in an out-of-order processor, the effectiveness of the register windows depends on the size of the map table, which in turn determines the minimum number of physical registers.

To extract high levels of parallelism, out-of-order processors require many more physical registers than architected ones, to store the uncommitted values of a large number of instructions in flight [6]. Therefore, an out-of-order processor with register windows requires a large amount of physical registers because of a twofold reason: to hold multiple contexts and to support a large instruction window. Unfortunately, the size of the register file has a strong impact on its access time [6], which may stay in the critical path that sets the cycle time. It has also an important cost in terms of area and power. There exist many proposals that address

16th International Conference on
Parallel Architecture and Compilation Techniques (PACT 2007)
0-7695-2944-5/07 $25.00 

this problem through different approaches.

One approach consists of pipelining the register file access [9]. However, a multi-cycle register file requires a complex multiple-level bypassing, and increases the branch misprediction penalty. Other approaches improve the register file access time, area and power by modifying the internal organization, through register caching [20] or register banking [3]. Alternative approaches have focused on reducing the physical register file size by reducing the register requirements through more aggressive reservation policies: late allocation [7] and early release [13] [15] [2].

In this paper we propose a novel early register release technique for out-of-order processors that builds upon the information provided by a register windows mechanism to achieve an impressive level of register savings. On conventional processors, a physical register remains allocated until the next instruction writing the same architectural register commits. However, the procedure call and return semantics enables more aggressive conditions for register release:

1. When a procedure finishes and its closing return instruction commits, all physical registers defined by this procedure can be safely released. The values defined in the closed context are *dead values* that will never be used again.

2. When the rename stage runs out of physical registers, mappings defined by instructions that are not in-flight, which belong to caller procedures, can also be released. However, unlike the previous case, these values may be used in the future, after returning from the procedure, so they must be saved to memory before they are released.

By exploiting these early release opportunities, the proposed scheme achieves a drastic reduction in physical register requirements. It reduces the number of physical registers to the same number of architectural plus one (128 in our experiments for IPF binaries) and still achieves almost the same performance as an unbounded register file.

Contrary to the common belief that out-of-order processors with register windows would need a large physical register file, this paper shows that register windows, together with the proposed techniques, can significantly reduce the physical register file pressure to the bare minimum. In other words, we show that our proposed scheme achieves a synergistic effect between register windows and out-of-order execution, resulting in an extremely cost-effective implementation.

Besides, our scheme requires much lower hardware complexity than previous related approaches [15], and it requires minimal changes to a conventional register windows scheme.

As state above, a register windows mechanism works by translating compiler-defined local variables to architected registers prior to renaming them to physical registers. The information required for this translation is kept as a part of the processor state and must be recovered in case of branch misprediction or exception. Our scheme also provides an effective recovery mechanism for window information that is suitable for out-of-order execution.

The rest of this paper is organized as follows. Section 2 discusses the state of the art on early register release techniques. Section 3 describes our proposal in detail. Section 4 presents and discusses the experimental results. Section 5 analyze several cost and complexity issues of the proposed solution and previous approaches. Finally, the main conclusions are summarized in section 6.

## 2 Related Work

In this section we will shortly review techniques that make early releasing of physical registers, to reduce the average number of required registers.

Jones et al [11] uses the compiler to identify registers that will only be read once and rename them to different logical registers. Upon issuing and instruction with one of these logical registers as a source, the processor can release the register through checkpointing.

Moudgrill et al [14] suggested releasing physical registers eagerly, as soon as the last instruction that uses a physical register commits. The last-use tracking is based on counters which record the number of pending reads for every physical register. This initial proposal did not support precise exceptions since counters were not correctly recovered when instructions were squashed. More recently, Akkary et al. [2] proposed to improve the Moudgrill scheme by adding an *unmapped* flag for each physical register, which is set when a subsequent instruction redefines the logical register it has associated. Then, a physical register can be released once its usage counter is 0 and its *unmapped* flag is set. Moreover, for proper exception recovery of the reference counters, when a checkpoint is created the counters of all physical register belonging to the checkpoint are incremented. Similarly, when a checkpoint is released, the counters of all physical registers belonging to the checkpoint are decremented.

Monreal et.al. [13] proposed a scheme where registers are released as soon as the processor knows that there will be no further use of them. Conventional renaming forces a physical register to be idle from the commit of its *Last-Use* (LU) instruction until the commit of the first *Next-Version* (NV) instruction. The idea is to shift the responsibility from NV instruction to LU instruction. Each time a NV instruction is renamed, its corresponding LU instruction pair is marked. Marked LU instructions reaching the commit stage will release registers instead of keeping them idle until the commit of the NV instruction.


0-7695-2944-5/07 $25.00 

Ergin et al. [5] introduced the checkpointed register file to implement early register release. This scheme can release a physical register before the redefining instruction is known to be non-speculative. This is done by copying its value into the shadow bit-cells of the register where it can be accessed easily if a branch misprediction occur.

Oehmke et.al. [15] have recently proposed the *virtual context architecture*(VCA), which maps logical registers holding local variables to a large memory address space and manages the physical register file as a cache that keeps the most recently used values. Logical register identifiers are converted to memory addresses and then mapped to physical registers by using a tagged set-associative rename map table. Unlike the conventional renaming approach, the VCA rename table look up may miss, i.e. there may be no physical register mapped to a given logical register. When the renaming of a source register causes a table miss, the value is restored from memory and a free physical register is allocated and mapped onto the table. If there are no free physical registers or table entries for a new mapping, then a replacement occurs: a valid entry is chosen by LRU, the value is saved to memory and its physical register is released. Although the VCA scheme does not properly define register windows as such, in practice it produces similar effects: multiple procedure contexts are maintained in registers, and the available register space is transparently managed without explicit saves and restores. However, unlike our approach, their tagged set-associative map table adds substantial complexity to the rename stage, which might have implications on the cycle time. Furthermore, when a map table entry is replaced, its associated value is always saved to memory, regardless of whether it is actually a *dead value*. This occurs because VCA is not able to distinguish *dead values* from *live values*.

Some current commercial architectures implement register windows to reduce procedure call/return overhead: SPARC [4] and Itanium [8]. In the former case, where register windows are of fixed sized, overflows and underflows are handled by trapping to the operative system. In the latter case, where register windows are of variable size, overflows and underflows are solved by a hardware mechanism called *Register Stack Engine*.

# 3 Early Register Release with Register Windows

This section describes our early release register technique base on register windows. Our scheme assumes an ISA with full register window support, such as IA64 [10]. Along this paper we assume a conventional out-of-order processor with a typical register renaming mechanism that uses a Map Table to associate architected to physical registers. In IA64, the 128 architected integer registers are divided into two groups: 32 static registers and 96 windowed registers. The static registers are available to all procedures while the windowed registers are allocated on demand to each procedure. Our technique is only applied to windowed registers. Both, static and windowed architected registers map to a unified physical register file.

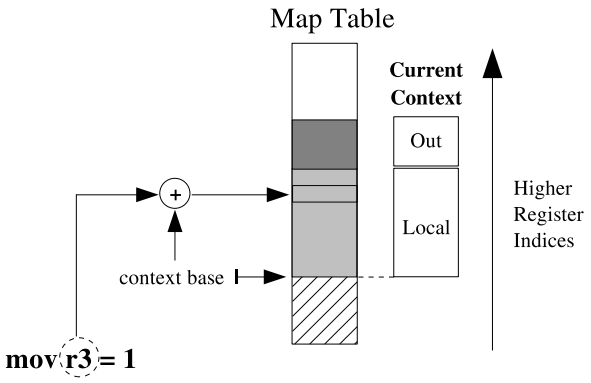


**Figure 1. Dynamic translation from virtual register *r3* to its corresponding architectural windowed register.**

A basic out-of-order register window mechanism is explained first. Then, we expose two early register release opportunities and a mechanism to exploit them.

## 3.1 Baseline Register Window

Register windows is a technique that helps to reduce the loads and stores required to save registers across procedure calls by storing the local variables of multiple procedure contexts in a large register file [18]. Throughout this paper we will use also the term procedure context to refer to register windows.

From the the ISA's perspective, all procedure contexts use the same 'virtual' register name space. However, when a procedure is called, it is responsible for dynamically allocating a separate set of consecutive architected registers, a register window, by specifying a *context base pointer* and a window *size*. Each virtual register name is then dynamically translated to an architected register by simply adding the base pointer to it (see Figure 1). Notice that register windows grow towards higher register indices.

Every register window is divided into two regions: the *local region*, which include both input parameters and local variables, and the *out region*, where it passes parameters to its callee procedures. By overlapping register windows, parameters are passed through procedures, so that registers holding the output parameters of the caller procedure become the local parameters of the callee. The overlap is il-


0-7695-2944-5/07 $25.00 

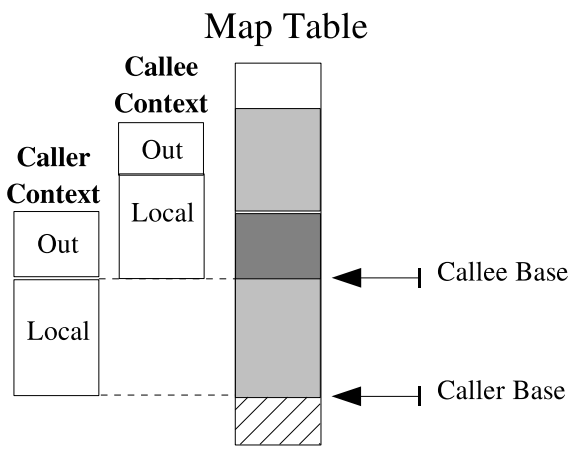


**Figure 2. Overlapping Register Windows.**

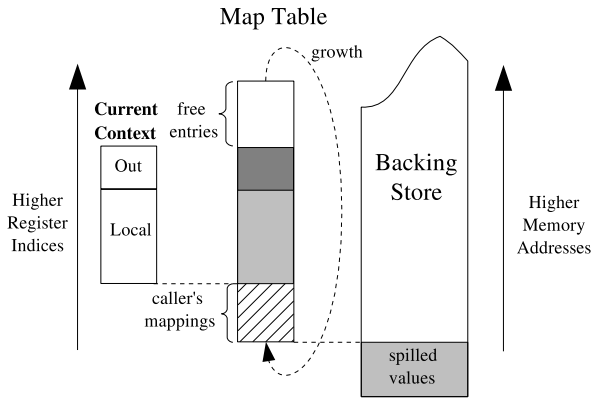


**Figure 3. Relationship between Map Table and Backing Store memory.**

lustrated in Figure 2.

Hence, every register window is fully defined by a *context descriptor* having three parameters: the *context base pointer*, which sets the beginning of the register window; the *context size*, which includes the local and out regions; and the *output parameters*, which defines the size of the out region.

The register windows are managed by software, with three specific instructions: *br.call, alloc* and *br.ret*. Br.call creates a new context, by setting a context descriptor with its base pointing to the first register in the out region of the caller context, and its context size equal to the out region's size. Alloc modifies the current context descriptor by setting a new context size and output parameters. Finally, br.ret returns to the caller procedure and makes its context the current context. The compiler is responsible for saving, on each procedure call, the caller context descriptor in a local register and restoring it later on return.

It may happen that an alloc instruction increases the window size but there is no room available to allocate the new context size in the architected registers, i.e. at the top of the map table. In such case, the contents of some physical registers at the bottom of the map table (which belong to callers procedure contexts) are sent to a special region of memory called *backing store*, and then they are released. Such operation is called a *spill*. Note that spills produce both free physical registers and free entries in the map table. These entries can then be assigned to new contexts to create the illusion of an infinite-sized register stack [1].

When a procedure returns, it expects to find its context in the map table, and the corresponding values stored in physical registers. However, if some of these registers were previously spilled to memory, the context is not completely present. For each missing context register, a physical register is allocated and renamed in the map table. Then, the value it had before spilling is reloaded from the backing store. Such operation is called a *fill*. Subsequent instructions that use this value are made data dependent on the result of the fill so they are not stalled at renaming until the fill completion.

The relationship between map table and backing store is shown in Figure 3. Notice that the map table is managed as a circular buffer. Spill and fill operations will be discussed in more detail in section 3.5.

### 3.2 Early Register Release Techniques

On a conventional out-of-order processor, a physical register is release when the instruction that redefines it commits. To help reducing physical register pressure, we have identified two opportunities for early releasing registers, and we propose two techniques to exploit them: *Context Release* and *Register Release Spill*. Our techniques are motivated by the observation that if none of the instructions of a procedure are currently in-flight, not all mappings of the procedure context need to stay in the map table. We refer to those mappings as *not-active*. Not-active mappings can be released as well as their associated physical registers. Some examples will be given below.

The Context Release technique is illustrated in Figure 4. As shown in Figure 4a, when br.ret commits, there are not in-flight instructions whose operands belong to context 1, so context 1 mappings that do not overlap with context 2 are *not-active* and can be released. In fact, none of these registers will never be used, so they can be released without having to wait until a subsequent redefinition commits [2].

[1] Each mapping has a unique associated backing store address.

[2] Note that the alloc instruction can either enlarge or shrink the size of a context, thus it may allocate or release registers. If a context size is shrunk,


0-7695-2944-5/07 $25.00 

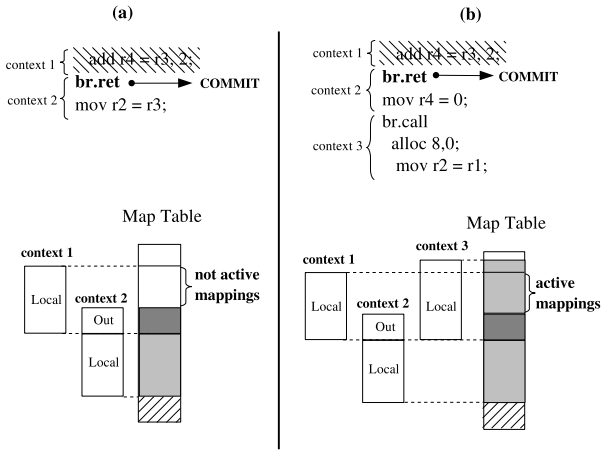


**Figure 4.** ***Context-Release*** **technique. (a) Context 1 mappings are *not-active* so they can be released. (b) Context 1 mappings are *active* (they have been redefined by context 3) so they can not be released.**

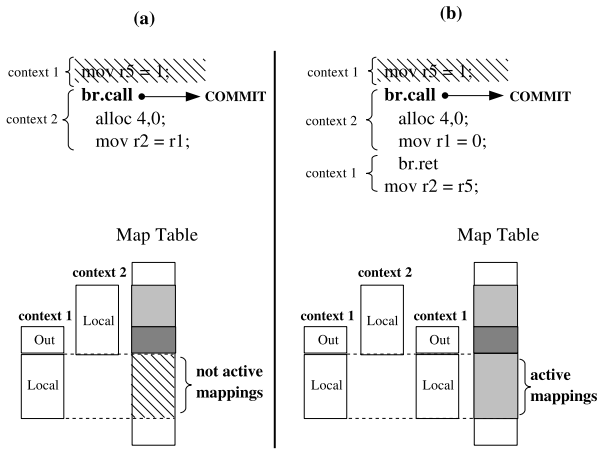


**Figure 5.** ***Register-Release Spill*** **technique (a) Context 1 mappings are *not-active* so they can be early released. (b) Context 1 mappings are *active* (after returning from the procedure) so they can not be released.**

However, the Context Release technique is not always possible when a br.ret commits. As shown in Figure 4b, although context 1 is not active, its mappings can not be released because the map table entries have been reassigned to context 3. Since there are in-flight instructions that use context 3, their mappings are active.

The second early register release technique, the *Register Release Spill*, is illustrated in Figure 5. As shown in Figure 5a, when *br.call* commits, context 1 mappings become not-active. This is not the case in Figure 5b, where the context 1 procedure has already returned before br.call commits, so their mappings have become active. Hence, only not-active mappings from Figure 5a can be early released. However, since such mappings belong to caller procedures, they must first spill the content of their associated physical registers. We will refer to this operation as Register Release Spill.

Though beneficial for register pressure, this technique increases the amount of spill/fill operations. Our experiments have shown that a blind application of the Register Release Spill to the baseline register window scheme increases the amount of spill/fill operations from 1% to 7% of the total number of commited instructions. Since spilling registers has an associated cost, it could reduce the benefits brought by register windows. Hence, Register Release Spill is only triggered when the released registers are actually required for the execution of the current context, i.e. if the renaming runs out of physical registers. Our experiments have show that this policy increases only spill/fill operations to 2% of the total number of commited instructions, and it outperforms a blind Register Release Spill by 3%. These experiments apply only Register Release Spill techniques and not Context Release techniques.

Notice that there is a slight difference between the Register Release Spill and the conventional spill used in the baseline (see section 3.1). A conventional spill is triggered by a lack of mappings when the window size is enlarged. What Register Release spill actually does is "to steal" physical registers from callers procedures to ensure an optimum execution of the current active zone.

### 3.3 Implementation

To implement register windows in an out-of-order processor, we propose the *Active Context Descriptor Table* (ACDT). The ACDT tracks all uncommitted context states to accomplish two main purposes: to identify the active mappings at any one point in time, which is required by our early register techniques; and to allow precise state recovery of context descriptor information in case of branch mispredictions and other exceptions.

The ACDT acts as a checkpoint repository that buffers in a *fifo* way, the successive states of the current context descriptor. As such, a new entry is queued for each context modification (at rename stage in program order), and it is removed when the instruction that has inserted the checkpoint is committed. Hence, ACDT maintains only active context descriptors. With these information, the ACDT allows precise state recovery of context descriptor informa-

the Context Release technique can be also applied when the alloc commits.


0-7695-2944-5/07 $25.00 

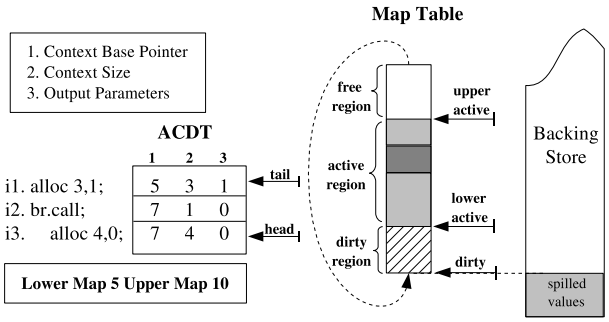


**Figure 6. The pointers *upper active, lower active* and *dirty* divide the map table in three regions: *free, active* and *dirty*. *Upper active* and *lower active* pointers are computed using ACDT information. *Dirty* pointer points to the first non-spilled mapping.**

tion: in case of branch mispredictions and other exceptions, when all younger instructions are squashed, the subsequent context states are removed from the ACDT, too.

As they are define, active mappings stay in a continuous region on the map table, called *active region*. The active region is bounded by two pointers that are maintained within the ACDT: the *lower active* and the *upper active*. Both pointers are updated at each context modification using the information present in the ACDT: the *lower active* equals to the minimum context base pointer present in the ACDT, and the *upper active* equals to the maximum of the *top pointers* (actually the sum of base + size) present in the ACDT.

The mappings staying below the *lower active* pointer belong to not-active contexts of callers procedures. These mappings may eventually become free if their values are spilled to memory, or may become active again if a br.ret restores that context. They form the so called *dirty region* of the map table, and it lays between the *lower active* pointer and a third pointer called *dirty*. The *dirty* pointer equals to the first mapping that will be spilled if the renaming engine requires it. Actually, the associated backing store address of the *dirty* pointer corresponds to the top of the backing store.

Finally, the *free-region* lays between the *upper active* pointer and the *dirty* pointer (note that the map table is managed as a circular buffer), and it contains invalid mappings. The three regions are shown in Figure 6.

In conclusion, when a procedure finishes and its closing br.ret commits, we can apply the Context Release technique which frees the not-active mappings of the closed context by adjusting the *upper active* pointer and releasing the physical registers associated to the mappings that fall above the new *upper active* pointer. These mappings contain dead values and become part of the *free region*. Furthermore, when the renaming runs out of physical registers, a Register Release Spill technique is triggered and it frees as many not-active mappings as needed from the *dirty region* starting at the *dirty* pointer.

### 3.4 Delayed Spill/Fill Operations

A closer look to the register window scheme for out-of-order processors described in the previous sections, reveals that there is still some room for improvement.

First, when an alloc is executed and there is not enough space int the map table, spill operations are generated to free the required mappings. Assuming a realistic scenario, there may exist a limit on the number of spills generated per cycle. So a massive spill generation may stall the renaming for several cycles. Hence, we propose to defer each spill until the map table entry is actually reassigned by a subsequent instruction.

Second, when a br.ret instruction is executed and the restored context is not present in the map table, fill operations are generated until all its mappings are restored. As it happens with the previous case, a massive fill generation may stall the renaming for several cycles. Moreover, it may happen that some of these mappings will never be used. Actually, it is quite often that a br.ret is followed by another br.ret, so none of the restored mappings are used. Hence, we propose to defer each fill and its corresponding physical register allocation until the mapping is actually used by a subsequent instruction. By delaying fill operations we achieve a lower memory traffic and we reduce the physical register pressure.

Hence, when an alloc is executed and there is not enough space for the new context, the required mappings from the dirty region are appended to the active region, and marked with a *pending spill* bit, so the alloc is not stalled. When an subsequent instruction redefines a mapping with its pending spill bit set, the current mapping is first spilled to the backing store. In a similar way, when a br.ret is executed and the restored context is not present in the map table, the required mappings from the free region are appended to the active region, and marked with a *pending fill* bit, so the br.ret instruction is not stalled. When a subsequent dependent instruction wants to use a mapping with the pending fill bit set, a fill operation is generated and a new physical register is reserved.

It might happen that a mapping with the pending spill bit set is not redefined until a subsequent nested procedure, and the map table has wrapped around several times. In that case, it would be costly to determine its associated backing store address. Thus, it is less complex to have an engine that autonomously clear the pending spills that were left behind. This problem does not occur with pending fills be-

cause these are actually invalid mappings that may be safely reused.

### 3.5 Spill-Fills operations

This section explains some implementation considerations about the spill and fill operations.

A naive implementation of spills and fills could insert them into the pipeline as standard store and load instructions. However, spills and fills have simpler requirements that enable more efficient implementations. First, their effective addresses are known at rename time, and the data that the spills store to memory are committed values. Since all their source operands are ready at renaming, they can be scheduled on a simpler hardware. Second, a given implementation could be further optimized by making that the system guarantees that spills and fills do not require memory disambiguation with respect to program stores and loads. In this case, spill and fill operations could be scheduled independently from all other program instructions by using a simple *fifo* buffer [15].

## 4 Evaluation

This section evaluates the two early register release techniques, Context Release and Register Release Spill, and the Delay Spill/Fill technique presented in previous sections.

### 4.1 Experimental Setup

All the experiments presented in this paper use a cycle-accurate, execution-driven simulator that runs IA64 ISA binaries. It has been built from scratch using the Liberty Simulation Environment (LSE) [19]. LSE is a simulator construction system, based on module definitions and module communications, that also provides a complete IA64 functional emulator that maintains the correct machine state.

We have simulated eleven integer benchmark programs from Spec2000 [1] using the Minne Spec [12] input set. Floating-point benchmarks have not been evaluated because the IPF register window mechanism is only implemented on integer registers. All benchmarks have been compiled with IA64 Intel's compiler (Electron v.8.1) using maximum optimization levels and profile information. For each benchmark, 100 million committed instructions are simulated. To obtain representative portions of code to simulate, we have used the Pinpoint tool [16].

The simulator models in detail an eight-stage out-of-order processor. It pays special attention to the implementation of the rename stage and models many IA64 peculiarities that are involved in the renaming, such as the register rotation and the application registers. The register stack engine have been replaced by our scheme. The microarchitecture features predicate prediction [17]. Load-store queues, as well as the data and control speculation mechanisms defined in IA64, are also modeled and integrated in the memory disambiguation subsystem. The main architectural parameters are shown in Table 1.

| Architectural Parameters | |
|---|---|
| Fetch Width | Up to 2 bundles (6 instructions) |
| Issue Queues | Integer Issue Queue: 80 entries<br>Floating-point Issue Queue: 80 entries<br>Branch Issue Queue: 32 entries<br>Load-Store Queue: 2 queues 64 entries |
| Reorder Buffer | 256 entries |
| L1D | 64KB, 4way, 64B block, 2 cycle latency<br>Non-blocking, 12 primary misses,<br>4 secondary misses, 16 write-buffer entries<br>2 load, 2 store ports |
| L1I | 32KB, 4 way, 64B block, 1 cycle latency |
| L2 unified | 1MB, 16 way, 128B block, 8 cycle latency<br>Non-blocking, 12 primary misses<br>8 write-buffer entries |
| DTLB | 512 entries. 10 cycles miss penalty |
| ITLB | 512 entries. 10 cycles miss penalty |
| Main Memory | 120 cycles of latency |
| Multilevel Branch Predictor | First-level: Gshare 14-bit GHR<br>Total size: 4 KB. 1-cycle access<br>Sec-level: Perceptron. 30b GHR, 10b LHR<br>Total size :148 KB. 3-cycle access<br>10 cycles for misprediction recovery |
| Predicate Predictor | Perceptron. 30b GHR. 10b LHR<br>Total size :148 KB. 3-cycle access<br>10 cycles for misprediction recovery |
| Integer Map Table | 96 local entries, 32 global entries |
| Integer Physical Register File | 129 physical registers |

**Table 1. Main architectural parameters used.**

The simulator models in detail the baseline register windows, as well as our early register release techniques and the ACDT mechanism, as described in the previous section. The simulator also models the VCA register windows technique, featuring a four-way set associative cache for logical-physical register mapping, with 11-bit tags, as described in [15], although it was adapted to the IA64 ISA.

### 4.2 Early Register Release Performance

This section evaluates the performance of our proposals (the Context Release, the Register Release Spill and the Delayed Spill/Fill techniques), and compares it to a configuration that uses the VCA register windows scheme. For each configuration, performance speedups are normalized IPCs relative to a configuration that uses the baseline register windows scheme with 160 physical registers. Although this gives an advantage of 32 registers to the baseline, we

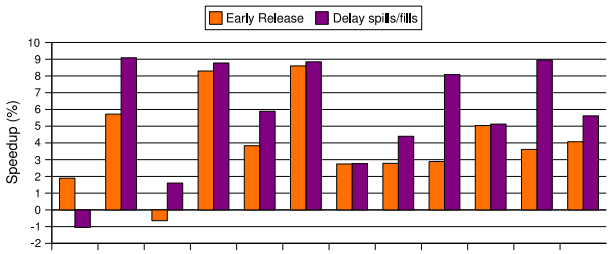


**Figure 7. Performance evaluation of Delayed Spill/Fill technique. Speedups normalized to the baseline register window scheme with 160 physical registers.**

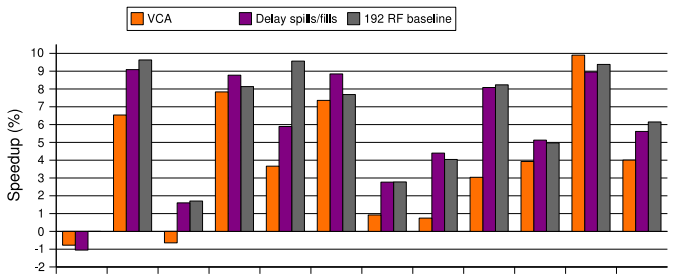


**Figure 8. Performance evaluation of VCA scheme, our Delayed Spill/Fill technique and the baseline register window scheme with 192 physical registers. Speedups normalized to the baseline register window scheme with 160 physical registers.**

believe that it is a more reasonable design point for such an out-of-order processor with 128 logical registers. We will show that, although our scheme has much less registers, not only it outperforms the 160-registers baseline, but it still performs within a 1% of a 192-registers baseline. At the end of the section we provide a more complete performance comparison of both schemes, for a wide range of 128 to 256 physical registers.

Figure 7 compares the performance of a configuration that uses only the Context Release and Register Release Spill techniques (labelled Early Release in the graph), and another configuration that also applies the Delayed Spill/Fill technique (labelled Delayed Spill/Fill in the graph). With only one exception (*gzip*), the Delayed Spill/Fill configuration outperforms the basic Early Release configuration, with average speedups over the baseline of 6% and 4% respectively. Adding the Delayed Spill/Fill technique to the Early Release configuration has an special performance impact on *vortex* and *twolf*, where the number of fill operations drastically drop from 7% to 1% of the total numer of committed instructions. Such a reduction produces notable performance improvements because it not only reduces memory traffic but also the amount of physical registers allocated by fill operations. On average, by using the Delayed Spill/fill tehcnique fill operations drop from 4% to 1% in comparison to the Early Release configuration.

The graph also shows a slight performance loss (1%) for *gzip* when applying the Delayed Spill/Fill technique. Although initiating spills and fills on demand avoids unnecessary operations, with a positive effect on register presure, it also reduces the distance in time between fill operations and their dependent instructions. Therefore, it reduces the tolerance of these instructions to memory latency, which may degrade performance if cache misses are frequent. This is the case with *gzip*, which has the highest cache miss rate (14%) for fill operations, in contrast with the average 2.5% of all benchmarks. For *gzip*, the average waiting time spent by instructions that depend on fill operations increases fom 15 cycles to 30 cycles when applying the Delayed Spill/Fill technique.

Figure 8 compares the performance of our proposal, with Early Release and Delayed Spill/Fill techniques, to the VCA scheme. Our scheme slightly outperforms the VCA on all benchmarks, except in *twolf* (where it loses by 1%). Notice that the VCA experiences a similar slowdown for *gzip*, because it also generates fills on demand, as discussed above. On average, our scheme achieves a performance advantage of 2% over VCA. The graph also shows the performance of a scheme configured as the baseline but giving it the advantage of a large 192 physical register file and assuming no increase on its access latency. Except for one benchmark (*crafty*) our 128-registers scheme achieves almost the same performance as the optimistic 192-registers scheme.

Figure 9 compares the performance of our proposal, with Early Release and Delayed Spill/Fill techniques, to the baseline, when the number of physical registers varies between 128 and 256. As expected (similar results have been published elsewhere), the baseline improves performance by increasing the number of registers, up to a saturation point around 192, beyond which it only gets marginal additional improvements. As shown in the graph, our scheme consistently outperforms the baseline. However, the most remarkable result is that the baseline curve drastically degrades as the number of registers decreases (up to a 37% IPC), while our proposal suffers just a very small performance loss (less than 4% IPC). On average, our scheme

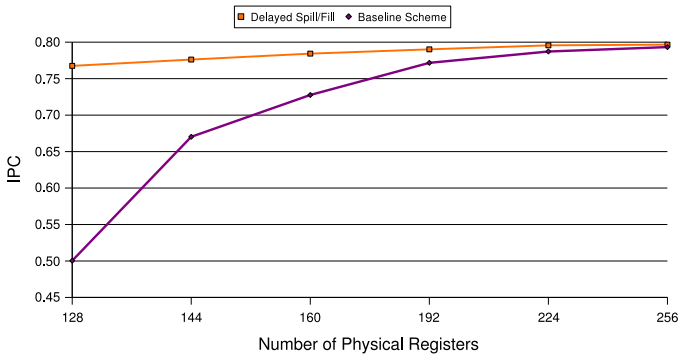


**Figure 9. Performance evaluation of our Delayed Spill/fill technique and the baseline register window scheme in terms of IPC, with different number of physical registers.**

is capable to achieve the performance of the 192-registers baseline with only the minimum number of physical registers (i.e. the number of architected registers plus one).

## 5 Cost and Complexity Issues

The previous section evaluates our proposal only in terms of performance. However, it is also interesting to analyze it in terms of cost and hardware complexity.

We have proposed a low cost implementation of the rename logic to support register windows on out-of-order processors. Compared to an in-order processor with register windows, our scheme only adds the ACDT and three map table pointers. We have experimentally found that a simple 8-entry ACDT table achieves near-optimal performance.

Compared to the VCA approach, our scheme is substantially less complex, since our proposal uses a conventional, direct-mapped table whereas the VCA requires a larger set-associative map table to hold memory address tags. Fitting the rename delay into the processor cycle time is typically a challenging problem because of its inherent complexity. Hence, adding complexity to the rename stage increases its latency, which may have implications on cycle time and power.

Moreover, the effectiveness of the register windows technique depends on the size of the architectural register file [18] so future implementations may take advantage of increasing the map table size, which emphasizes the importance of a simple scheme for scalability.

Finally, our proposal produces a lower number of spill operations than the VCA, which not only affects to performance, but also reduces the power requirements. In comparison to the baseline, which generates a 0,5% of spill operations from the total number of commited instructions, the VCA generated up to 3% from the total number of commited instructions, whereas our scheme generates only up to 1% from the total number of commited instructions.

## 6 Summary and Conclusions

In this paper we proposed two early register release techniques for out-of-order processors that, by using the information provided by register windows, achieve a drastic reduction in the size of the physical register file. These techniques, called *Context Release* and *Register Release Spill*, are based on the observation that if none of the instructions of a procedure are currently in-flight, not all mappings of the procedure context need to stay in the map table. These mappings, called *not-active*, can be released, as well as their associated physical registers.

The *Context Release* technique releases mappings defined by a procedure whose closing return instruction has committed. The values of these mappings are *dead values* that will never be used again, so their associated physical registers can be safely released.

The *Register Release Spill* technique is automatically triggered when the rename stage runs out of physical registers and it releases not-active mappings that belong to caller procedures. However, unlike the previous technique, the values contained in these mappings are *live values* that may be used in the future, after returning from the procedure, so they must be spilled before releasing them.

We introduce the *Active Context Descriptor Table* (ACDT), that tracks all uncommitted context states to accomplish two main purposes: identify the active mappings at any point in time, which is required by our early register techniques; and implement precise state recovery of context descriptor information in case of branch mispredictions and other exceptions.

Moreover, in order to avoid unnecessary rename stalls caused by the spills and fills generated by *alloc* and *br.ret* instructions respectively, we propose to defer spills and fills operations until the corresponding registers are used. Hence, a fill is generated and its corresponding physical register is allocated when the mapping is actually used by a subsequent instruction. In the same way, a spill is generated when the mapping is actually reassigned by a subsequent instruction.

Applying these techniques, a processor fitted with only the minimum required number of physical registers, i.e. the number of architected registers plus one (which is 128 in our experiments with IPF binaries), achieves almost the same performance as a baseline scheme with an unbounded register file. Moreover, in comparison to previous techniques, our proposal has much lower hardware complexity and requires minimal changes to a conventional register

window scheme.

## 7 Acknowledgements

This work is supported by the Spanish Ministry of Education and Science and FEDER funds of the EU under contracts TIN 2004-03072, and TIN 2004-07739-C02-01, and Intel Corporation.

## References

[1] Standard performance evaluation corporation. spec. *Newsletter, Fairfax, VA*, September 2000.

[2] H. Akkary, R. Rajwar, and S. t. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *MICRO 36: Proceedings of the 36th annual international symposium on Microarchitecture*, 2003.

[3] J.-L. Cruz, A. Gonzalez, M. Valero, and N. P. Topham. Multiple-banked register file architectures. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 316–325, 2000.

[4] D.L.Weaver and T.Germond. *SPARC Architecture Manual (version 9)*.

[5] O. Ergin, D. Balkan, D. Ponomarev, and K. Ghose. Increasing processor performance through early register release. In *ICCD: Proceedings of International Conference on Computer Design*, 2004.

[6] K. Farkas, N. Jouppi, and P. Chow. Register file considerations in dynamically scheduled processors. In *HPCA '96: Proceedings of the 2th International Symposium on High-Performance Computer Architecture*, pages 40–51, 1996.

[7] A. Gonzalez, J. Gonzalez, and M. Valero. Virtual-physical registers. In *HPCA '98: Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, 1998.

[8] L. Gwennap. Intel, hp make epic disclosure. *Microprocessor report*, 11:1 – 9, October 2001.

[9] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the pentium 4 processor. *Intel Technology Journal Q1*, February 2001.

[10] Intel Corporation. *Intel Itanium Architecture Software Developer's Manual. Volume 2: System Architechiture*, 2002.

[11] T. M. Jones, M. F. P. O'Boyle, J. Abella, A. Gonzalez, and O. Ergin. Compiler directed early register release. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 110–122, Washington, DC, USA, 2005. IEEE Computer Society.

[12] A. KleinOsowski and D. J. Lilja. Minnespec: A new spec benchmark workload for simulation-based computer architecture research. 2002.

[13] T. Monreal, V. Viñals, A. Gonzalez, and M. Valero. Hardware schemes for early register release. In *ICPP-02: Proceedings of International Conference on Parallel Processing*.

[14] M. Moudgill, K. Pingali, and S. Vassiliadis. Register renaming and dynamic speculation: An alternative approach. In *MICRO 26: Proceedings of the 26th annual international symposium on Microarchitecture*, pages 202–213, Nov. 1993.

[15] D. W. Oehmke, N. L. Binkert, T. Mudge, and S. K. Reinhardt. How to fake 1000 registers. In *MICRO 38: Proceedings of the 38th annual International Symposium on Microarchitecture*, pages 7 – 18, 2005.

[16] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large intel itanium programs with dynamic instrumentation. In *MICRO 37: Proceedings of the 37th annual International Symposium on Microarchitecture*, pages 81–92, 2004.

[17] E. Quiñones, J.-M. Parcerisa, and A. Gonzalez. Improving branch prediction and predicate execution in out-of-order processors. In *HPCA '07: Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, Feb. 2007.

[18] R. Rakvic, E. Grochowski, B. Black, M. Annavaram, T. Diep, and J. P. Shen. Performance advantage of the register stack in intel itanium processors. In *Workshop on Explicit Parallel Instruction Computing (EPIC) Architectures and Compiler Techniques*, 2002.

[19] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August. Microarchitectural exploration with liberty. In *MICRO 35: Proceedings of the 35th annual international symposium on Microarchitecture*, pages 271 – 282, 2002.

[20] R. Yung and N. Wilhelm. Caching processor general design. In *ICCD: Proceedings of International Conference on Computer Design*, pages 307–312, Oct. 1995.