Late Allocation and Early Release of Physical Registers

Teresa Monreal, Víctor Viñals, *Member*, *IEEE*, José González, *Member*, *IEEE Computer Society*, Antonio González, *Member*, *IEEE Computer Society*, and Mateo Valero, *Fellow*, *IEEE*

Abstract—The Register File is one of the critical components of current processors in terms of access time and power consumption. Among other things, the potential to exploit instruction-level parallelism is closely related to the size and number of ports of the register file. In conventional register renaming schemes, both register allocation and releasing are conservatively done, the former at the rename stage, before registers are loaded with values, and the latter at the commit stage of the instruction redefining the same register, once registers are not used anymore. In this paper, we introduce VP-LAER, a renaming scheme that allocates registers later and releases them earlier than conventional schemes. Specifically, physical registers are allocated at the end of the execution stage and released as soon as the processor realizes that there will be no further use of them. VP-LAER enhances register utilization, that is, the fraction of allocated registers having a value to be read in the future. Detailed cycle-level simulations show either a significant speedup for a given register file size or a reduction in the register file size for a given performance level, especially for floating-point codes, where the register file pressure is usually high.

Index Terms—Register renaming, out-of-order processors, register file optimization, physical register allocation and releasing, precise exceptions.

1 Introduction

Out-of-order superscalar processors exploit instruction-level parallelism by executing instructions as soon as their operands are ready, bypassing prior instructions in the sequential program order. Register renaming removes false register-dependences [18], [32] (output and anti-dependences) and creates a new register *version* for each register destination. Register versions are written only once and read as many times as needed to satisfy flow dependences. Among the versions of a given register, all but the committed one are speculative and may become useless if an exception or a branch misprediction occurs.

Most contemporary superscalar processors perform register renaming, but they differ in the architecture of the register versions storage and in the particular way versions are accessed and recovered after a branch misprediction or an exception occurs. Register versions can either be distributed among different data structures or centralized in a single file. There exist processors that keep noncommitted versions in the reorder buffer and copy these versions into the register file at commit [14]; other processors have a register file for noncommitted versions and another file for the committed ones [21]; and there are also some processors which have a single register file that merges together both committed and noncommitted versions [38]. In this work, we are interested in that last case, called *merged* register file in [32]. Details about the way we assume versions are accessed and recovered are shown in Section 2.

In dynamically scheduled processors, two key structures limit instruction overlapping: Issue Windows and Reorder Structures (Reorder or History buffers [33]). The Issue Window (IW) keeps track of data dependences and issues ready instructions to idle functional units, while the Reorder Structure retains the program order to enable recovery from exceptions and misspeculations. The instruction overlapping degree is highly dependent on the size of the IW and, thus, wide issue processors require large IW [36]. Besides, large IWs are needed to support high memory latencies. Unfortunately, the IW size is limited due to strict cycle time constraints [28]; however, some recent proposals are directed toward achieving the memory latency tolerance of large windows while maintaining the high clock rate of small windows [5], [8], [20]. Anyway, the support for many in-flight instructions (a large Reorder Structure) has negative implications for other critical parts of the microarchitecture such as the physical register file. Large register files are required to offer more registers, with more ports and a low latency [11], [28], [30]. For instance, Simultaneous Multithreaded processors need large register files to fulfill the above requirements in order to achieve their whole performance potential [35].

Many works point out that the register file access can impact the processor cycle time [2], [9], [11], [10], [39]. One solution proposed is to pipeline the register file access [16], [34]. The same techniques used for pipelining RAMs can be employed to pipeline the register file [27]. However, pipelining the register file is not trivial and significantly limits the processor performance. In particular, a multistage

- *T. Monreal and V. Viñals are with the Departamento de Informática e Ingeniería de Sistemas, Area de Arquitectura y Tecnología de Computadores, Centro Politécnico Superior, Universidad de Zaragoza, Edificio Ada Byron, Maria de Luna, 1, E-50018 Zaragoza, Spain. E-mail: {tmonreal, victor}@unizar.es.*
- *J. González is with Intel Barcelona Research Center, Intel Labs, Barcelona, Spain. E-mail: pepe.gonzalez@intel.com.*
- *A. González and M. Valero are with the Computer Architecture Department, Universitat Politècnica de Catalunya, c/ Jordi Girona 1-3, Mòdul D6, 08034 Barcelona, Spain. E-mail: {antonio, mateo}@ac.upc.es.*

Manuscript received 17 July 2003; revised 29 Jan. 2004; accepted 24 Mar. 2004.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0003-0403.

register file would increase the length of critical portions of the pipeline that are traversed by key microarchitectural loops. Loose loops, such as the branch resolution loop and the memory dependence loop, are shown to be critical in the issue-to-execute path and proposals such as the Distributed Register Algorithm (DRA) are targeted to remove the register file access out of that path [4].

On the other hand, the register file size (P registers) and the number of ports (T in total, read and write) determine silicon area, power consumption, and access time. For a range of interest of P and T (say $P < 160$, $T < 60$), the area and power consumption are proportional to $P \cdot T^2$ and the access time to $\sqrt{P} \cdot T$ [31].

Therefore, a more *direct* way to reduce register file delay is by reducing P, T, or both. To do this, common approaches trade off IPC for IPS (instructions per second). A first group of approaches address the file internal organization, basically without changing the interface with the functional units (T is unchanged), but reducing P in some way. Some examples are the Minimally-Ported Banked register file [2] or the two-level register hierarchy managed in an inclusive [9] or exclusive [2] way.

A second group of approaches suggest clustered microarchitectures, where the register file is sliced into banks, each bank feeding a functional unit cluster directly. Many of these solutions have been targeted to decentralize several critical structures and not only the register file. One example is the Dependence-Based architecture [28], where each bank is a complete copy of the register file ($= P, < T$), as in the Alpha 21264 two-cluster case [19]. Other examples are the Multicluster architecture [10] (each bank is assigned a subset of the ISA registers. $< P, < T$), schemes related to the distribution of physical registers [3], [7], and the Energy-Efficient Multicluster architecture [40] (each bank contains a subset of physical registers. $< P, < T$).

Both hierarchical register files and clustered microarchitecture ideas are collected in the DRA proposal [4]. DRA reduces the issue-to-execute latency by moving the time-consuming register file access out of that path. DRA uses clustered issuing logic and assigns instructions to a particular execution cluster at decode time. The monolithic register file is placed in the decode-to-issue path and a subset of all registers is distributed across small register caches which are placed within each execution cluster.

Finally, a third approach focuses on the mechanism that controls the allocation and release of physical registers, trying to reduce the average number of required registers. Current processors require many more physical registers than those strictly necessary to store the live register versions, that is, those versions that will be read in the future. This occurs because registers are allocated too early and released too late. Every instruction with a destination register allocates a physical register at the rename stage long before the result to be written into it is available. On the other hand, a physical register is sometimes released long after its last consumer-instruction commits, just when the first instruction writing the same logical register commits. In this paper, we concentrate on reducing the register waste due to both factors.

In order to delay allocation, we will use the virtual-physical register concept we proposed in previous works [12], [13], [24]. Virtual-physical registers (*VP*) allow the processor to delay the allocation of physical registers until the values they must store are available (at the end of the execution stage). In particular, we will use virtual-physical registers to perform *on-Demand allocation with Stealing from Younger* (*VP-dsy*), [24]. In short, this policy consists of an on-demand allocation of registers combined with a stealing mechanism that prevents older instructions from being delayed by younger ones.

On the other hand, to anticipate the release of registers we will adapt the *Last-Use*, *Next-Version* (*LU-NV*) concept we proposed in [25]. Under the *LU-NV* policy, a physical register is released as soon as it is known that no further use of it will be made. *LU-NV* release operates in processors supporting control speculation and precise exceptions, therefore, a register is not released if an exception handler or an alternative control-flow path could need it.

Our contribution in this work is the design and evaluation of VP-LAER, a renaming policy that simultaneously performs Late Allocation, based on *VP-dsy*, and Early Release, based on *LU-NV*. We will describe how these two components work in the VP-LAER extension, showing that VP-LAER is feasible and outperforms the best of the two previous schemes working alone.

This paper is structured as follows: Section 2 presents background on conventional renaming and the potential gains that a more aggressive policy could produce. Section 3 presents the rationale behind VP-LAER, while Section 4 details all the hardware resources needed and their collective operation. Section 5 presents the experimental framework and discusses the performance or savings of alternative design points. Finally, Section 6 outlines some related work and Section 7 summarizes the main conclusions of the paper.

2 Conventional Renaming Policies: Implementation and Efficiency

In this section, we introduce some terminology, review the conventional model for allocating and releasing registers, and, finally, give experimental evidence of its low efficiency.

Table 1 shows the number of physical registers (P) and ports (T) of four example processors with merged register files: MIPS R10/12K [38], [15], Alpha 21264 [19], and Intel P4 [16]. It also shows the size (N) and the name of the buffer containing the uncommitted instructions.

MIPS R10K supports up to $N = 32$ uncommitted instructions in a Reorder Structure called Active List. Since MIPS ISA has $L = 32$ logical integer registers and $P = 64$ physical registers, this processor never stalls due to a lack of physical registers. We call this kind of scenario a *loose* register file ($P \approx L + N$). By contrast, in MIPS R12K and Alpha 21264, a long enough instruction sequence without branches and stores can stall the renaming process. We call this alternative a *tight* register file ($P < L + N$). With a tight register file a sequence with less than N instructions writing $P - L$ registers consumes all physical registers, forcing the processor to stop filling the Reorder Structure. Intel P4 has large register files (128int + 128fp), but, because we do not have enough details about the mapping of ISA instructions into μops, we cannot conclude whether they are loose or tight.

Loose designs allow the whole Reorder Structure to be filled under any conditions, whereas tight designs may contribute to reducing processor cycle time if the register file is in the critical timing path. As we will see later, the

TABLE 1
Number of Registers and Ports of Merged Register Files in Some Out-of-Order Processors

	MIPS R10K	MIPS R12K	ALPHA 21264	INTEL P4
P = # of Phys. Reg. in **Integer** File	64		2x 80	128
T = # of Read and Write Ports	7R 3W		(4R 6W) $x2^a$	n.a.
P = # of Phys. Reg. in **FP** File	64		72	128
T = # of Read and Write Ports	5R 3W		6R 4W	n.a.
N = Reorder Structure Size	32	48	80	126 µops
Reorder Structure Name	*Active List*	*Active List*	*In-Flight Window*	*Reorder Buffer*

a.Replicated because a single file with 14 ports (8 rd + 6 wr) could not be realized without compromising cycle time.

performance impact of a given renaming scheme depends on whether it is applied to loose or tight register files.

As a baseline renaming, we assume a conventional mechanism similar to that of processors in Table 1. Fig. 1 shows the involved components: Map Table, Reorder Structure, Free List, and In-Order Map Table. The Map Table (MT) keeps the logical to physical mapping [18]. The Free List provides the destination physical register identifiers (pd). A Map Table copy is made at each branch prediction so that the processor state can be recovered on a mispredicted branch [17].

The Reorder Structure (ROS) keeps information about all uncommitted instructions in program order. We assume a FIFO behavior implemented with an SRAM and read/write pointers. Therefore, an ROS address can be used as a unique instruction identifier.

While instructions are renamed, three fields are written into the Reorder Structure bottom entry: $< \text{old\_pd}, \text{rd}, \text{pd} >$. The identifier of the physical register containing the *previous* version (old_pd) is read from MT. The logical and physical identifiers of the destination register (the *current* version) are rd and pd,[1] respectively.

As instructions commit, the pd and rd entries are used for updating the In-Order Map Table (IOMT), and the old_pd identifier is used to release the *previous* version physical register by adding it to the Free List. The IOMT keeps the logical to physical architectural mapping. When an exception has to be serviced, IOMT avoids the need to roll back the Reorder Structure to recover the architectural state [33]. In Intel P4, IOMT is called the Retirement Register Alias Table [16].

Physical registers can be Free or Allocated, and Allocated registers can be either *Empty*, *Ready*, or *Idle*, according to the usefulness of their content (Fig. 2a). A physical register is Empty from the moment it is allocated until it is actually written. A register is Idle from the commit of the instruction using that register for the last time until the commit of the instruction producing the next version. An Allocated register is Ready when it is neither Empty nor Idle.

Fig. 2b shows the execution of a sample program with instructions labeled *V*, *LU*, and *NV*, where the physical register p7 experiences all states. Instruction *V* (*version*) creates a new register version for its logical destination r2, which is renamed to physical register p7. Later on, instruction *LU* (*last-use*) reads r2 for the last time. Afterward, instruction *NV* (*next-version*) rewrites r2.

1. Each entry in the Reorder Structure stores the result identifier, as in an *indirect Reorder Buffer*. But, it also contains the previous-version identifier, as in an *indirect History Buffer* [33]. This is the reason why we have adopted Reorder Structure as a more general term.

Therefore, p7 is Empty from the rename of instruction *V* until its execution ends. Next, p7 shifts to the Ready state, from where it is available to be delivered to consumer instructions. Later on, p7 is Idle from the *LU* commit to the *NV* commit and, eventually, p7 will be released when instruction *NV* commits.

2.1 Efficiency of Conventional Renaming Mechanisms

We define IPC loss as the average number of noncommitted instructions per cycle due to lack of registers. Table 2 shows the IPC loss of an eight-way superscalar processor with tight register files of 64 registers ($L = 32$, $P = 64\text{int} + 64\text{fp}$, $N = 128$) in relation to a loose configuration ($P = 160\text{int} + 160\text{fp}$) for a subset of SPEC 95 programs. The last row shows the speedup of the loose configuration in relation to the tight one. We consider only integer registers for integer programs and FP registers for FP programs. The experimental framework is detailed in Section 5.

It can be observed that integer codes could reach a modest, though not negligible, speedup (1.08 on average—harmonic mean), whereas FP codes would benefit the most (1.50 on average—harmonic mean).

To get an insight into the renaming behavior, the average number of Allocated registers being either in the Empty, Ready, or Idle states is plotted in Fig. 3 along with the register file Utilization. Utilization is defined as the percentage of allocated registers having a value to be read in the future (Ready registers over Allocated registers). Results are again significantly different depending on the code type.

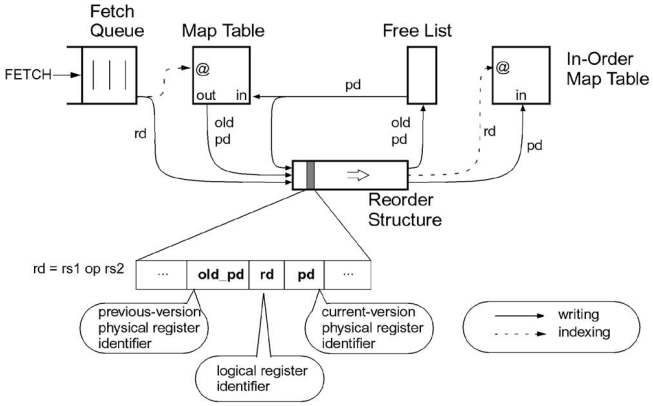

Fig. 1. Conventional renaming. Instructions enter the pipeline from the left, are placed in the Reorder Structure, and commit toward the right. Detail of an ROS entry showing the fields related to registers **old_pd**, **rd**, and **pd**.

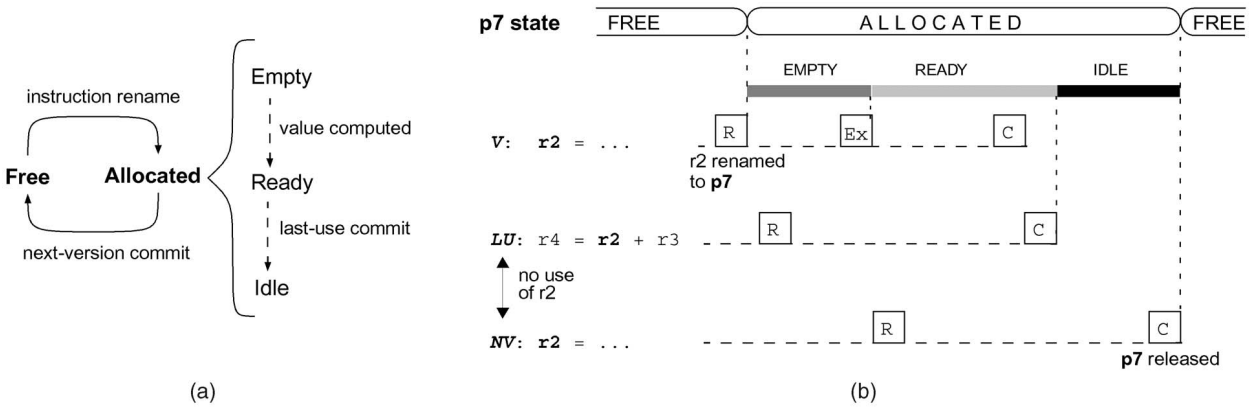

Fig. 2. (a) Breakdown of the ALLOCATED state and (b) example of state evolution of the physical register `p7` under a conventional rename policy. R = rename, Ex = execution, C = commit.

Integer codes have very low Utilization (39.3 percent on average) due to the heavy weight of Empty and Idle states (20.6 Idle and 11.9 Empty registers, on average). On the other hand, as expected from the modest attainable speedups just reported, integer codes are not strictly limited by lack of physical registers. Excluding `compress` and `li`, all applications have more than 11 free registers on average. Therefore, for this kind of programs improving renaming should mainly allow the integer register file to tighten without loosing IPC.

FP codes show higher Utilization (59.1 percent on average). The relative weight of the Idle state has decreased remarkably (12.2 Idle and 12.4 Empty registers, on average). But, in contrast to integer codes, FP codes put high pressure on the FP register file. `Mgrid`, `tomcatv`, and `swim` almost use all the available physical registers and the overall average number of free registers is as low as 3.8. Here, the main consequence of improving renaming should be an IPC increase.

Next, we introduce VP-LAER renaming, which removes the Empty state completely by delaying register allocation and reduces the Idle state stay by releasing registers earlier.

3 VP-LAER: Virtual-Physical Late Allocation with Early Release

In this section, we first describe the schemes used for allocation and release in VP-LAER separately and then the joint architecture is presented. Implementation details and examples are included in the next section.

3.1 Allocating Physical Registers in VP-LAER

The conventional renaming mechanism allocates physical registers in the rename stage. However, instructions do not really require physical registers until results are available. The reason for this early allocation is that physical registers are used both to store values and to keep track of dependences. However, only the latter objective requires bookkeeping at rename time, while the allocation of a storage location could be delayed if the processor used a separate artifact for tracking dependences. Such an artifact is what we call *virtual-physical* (VP) registers [12], [13], [24].

VP registers are tags and do not require any physical storage. In order not to stall the processor due to lack of VP registers, we need as many of them as the maximum number of in-flight instructions (number of ROS entries) plus the number of logical registers (int + fp). When an instruction is renamed, its logical destination register is mapped onto a VP register obtained from the list of free VP registers. Later on, when the instruction is in the last cycle of the execution stage, the VP register is mapped onto a physical register taken from the list of free physical registers. At the time the release is performed, both the VP and the physical registers return to their respective free lists. In our previous works, releasing operated in the conventional way, but VP-LAER uses the more aggressive approach outlined in the next section.

TABLE 2
IPC Loss and Speedup for an 8-Way Processor with 64int +64 fp Register File and a 128-Entry Reorder Structure

integer

	comp	gcc	go	li	perl
IPC loss	*0.48*	*0.12*	*0.14*	*0.22*	*0.16*
Speedup	*1.18*	*1.06*	*1.06*	*1.07*	*1.06*

floating point

	mgri	tomc	appl	swim	hydr
IPC loss	*1.23*	*0.68*	*1.02*	*1.20*	*0.78*
Speedup	*1.60*	*1.43*	*1.40*	*1.67*	*1.39*

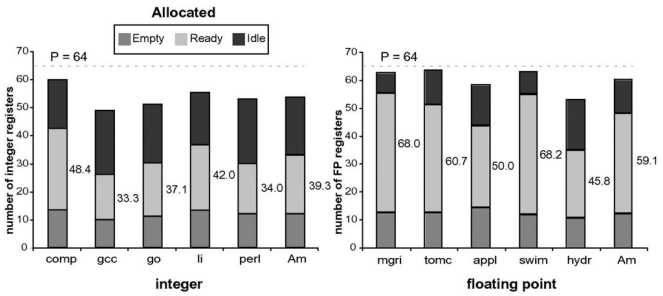

Fig. 3. Number of *Allocated* registers being either in the *Empty*, *Ready*, or *Idle* states for an 8-way processor with 64int + 64fp register file and a 128-entry Reorder Structure. The Register file Utilization appears close to the *Ready* bars.

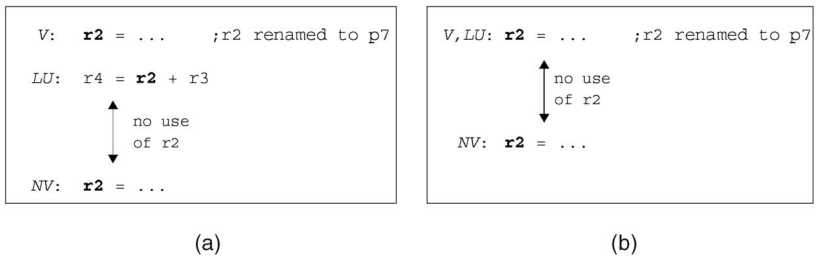

Fig. 4. Two examples of the early release potential under VP-LAER. In both codes, physical register p7 can be released when *LU* commits instead of waiting until *NV* commits.

When an instruction finishes execution and there are no free physical registers, the instruction has to wait to be reexecuted later on, with the expectation that some physical register will be released in the meantime. However, if this instruction was the oldest instruction in the processor, no instruction would commit and, thus, no physical register would be freed. To avoid this deadlock situation, a feasible allocation policy is to guarantee that a given number of the oldest instructions will obtain a register when they reach the write-back stage. We call this implementation parameter *Number of Reserved Registers*. In short, this allocation scheme assigns NRR registers to the oldest instructions with destination operand, whereas the rest of the physical registers are allocated on-demand, that is, they are assigned to the instructions that first reach the write-back stage. More details about the NRR scheme and its performance appear in [12], [13].

The NRR allocation scheme is not the only approach that may guarantee deadlock avoidance. Finding the optimal physical register allocation seems to be an unsolvable problem even with a perfect knowledge of future register references. However, we have found an implementable heuristic that outperforms the NRR approach and approximates such an optimal scheme [24]. This heuristic we have called *on-Demand with Stealing from Younger* (*dsy*) follows two register-handling rules:

- Registers should be allocated to those instructions that can use them earlier. In this way, the average number of unused registers is minimized.
- Given any two instructions fighting for the last free physical register, the execution of the younger instruction should be delayed, hoping that the winning, older instruction will release a physical register earlier (at commit time).

VP-LAER renaming uses this *VP-dsy* allocation scheme in which every writing instruction allocates a physical register in the last cycle of the execution stage if there are free registers. This meets the first criterion since registers will be allocated those instructions that finish execution first. If an instruction S reaches the last cycle of the execution stage and there is no free physical register, it is checked whether there is a younger instruction that has already allocated a register. If this is the case, it is better to stall the younger instruction, based on the second criterion. This can be achieved by stealing the register allocated to the younger instruction and giving it to S.

The stolen instruction must be reexecuted in the future, the reexecution factor being a significant overhead for the VP-dsy scheme. VP-LAER reduces this negative effect since the early release policy provides extra registers.

3.2 Releasing Registers in VP-LAER

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. In both codes of Fig. 4, this is true for register p7, which is a renamed version of r2. Fig. 4a shows the usual case where the last use of r2 is a read and Fig. 4b shows the infrequent case where the last use of r2 is a write and, so, the instruction labeled *V*, *LU* plays the dual role of creating a register version and using it for the last time.

Conventional renaming has a simple implementation supporting both control speculation and precise exception recovery (Section 2). However, the processor can run out of registers and stall in spite of having a number of Idle registers. By contrast, VP-LAER is able to release Idle registers earlier, enabling execution of instructions otherwise stalled for lack of registers. Under VP-LAER, in both codes of Fig. 4, physical register p7 could be released when the *LU* instruction commits.

The idea is to completely shift the release responsibility from the *NV* instruction to the *LU* instruction. To this end, each time an *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 corresponding *NV* instruction. We view this marking as follows: The *NV* instruction schedules an early release of a register for the commit of its *LU* instruction pair. It may happen that the *LU* instruction is already committed when renaming its *NV* pair, then no scheduling needs to be done and releasing can proceed immediately. This corresponds to the *LU-NV* early release policy we proposed in [25].

Under control speculation, a specific recovery hardware is used to undo the effects of a mispredicted path [17]. Likewise, if a speculative instruction *NV* has to be squashed, we need to undo any early releases it has scheduled. Recovery implementation in VP-LAER considers the following two cases:

- **Case 1.** Instruction *NV* is not speculative at rename. It may or may not be in the same basic block as instruction *LU* is. Any release scheduled by *NV* is considered as *safe*.
- **Case 2.** Instruction *NV* is speculative at rename. There are branches between *LU* and *NV* whose verification is pending. Therefore, any release scheduled by *NV* must be considered *speculative* and subject to squashing as long as instruction *NV* remains speculative.

All the concepts connected with release can be applied to both physical and VP registers. In fact, VP-LAER always

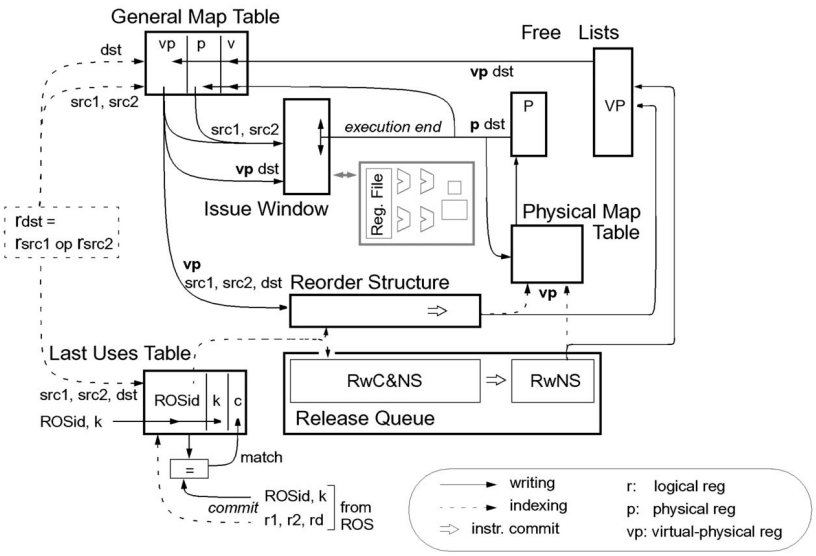

Fig. 5. VP-LAER renaming process.

releases a register pair: a VP register and the physical register which it is mapped to. Section 4.2 elaborates on the different situations in which registers are released in VP-LAER.

As regards precise exception recovery, the point in VP-LAER is that the value attached to a logical register r is only discarded if it is guaranteed that the first use of r is a write. With this, we can conclude that our mechanism does not strictly hold the usual condition of being precise:

> An interrupt is precise if the saved process state corresponds with a sequential model of program execution where one instruction completes before the next begins [33].

However, this definition of precise exceptions is sufficient but not necessary to guarantee a proper recovery. The optimization we propose is safe in that these discarded values are guaranteed not to be used by the program. As regards the use of discarded values inside an exception handler, two different situations can arise depending on the exception type. All exceptions not triggered by an anomalous execution (external interrupts, system calls, software breakpoints, undefined or illegal operation codes, etc.) find the correct architectural state because such exceptions are treated by the early release hardware as unconditional jumps. In contrast, exceptions raised by an execution fault (page fault, divide by zero, etc.) cannot use other than those referenced by the faulting instruction as parameter registers. Any extra parameter must be passed through memory (e.g., through the stack). But, it is no common practice at all to feed this kind of handlers with registers other than the instruction's own registers, whereby we consider that rewriting existing software is worth the effort (probably belonging to dynamic libraries) in view of the expected performance benefits. Similar optimizations in software were proposed elsewhere [22].

4 Hardware Resources and Examples

The global picture of the renaming process considered in VP-LAER is summarized in Fig. 5. The VP-LAER allocation part relies on a General Map Table which replaces the conventional Map Table and an Issue Window. The release part relies on a Last Uses Table, a Reorder Structure, and a Release Queue.

Two Free Lists and a Physical Map Table complement both parts. The P Free List is a pool of free physical registers, as in the conventional scheme, and the VP Free List is a pool of free virtual-physical registers. The **Physical Map Table** keeps the virtual-physical to physical mapping. When a writing instruction completes its execution, a new physical register is taken from the P Free List and the Physical Map Table is updated to reflect the new virtual-physical to physical mapping.

In VP-LAER, both renaming and releasing actions are always directed by virtual-physical registers. The Physical Map Table is the structure which stores, for each vp register, its latest physical mapping and it is mainly used either at branch misprediction in order to recover the proper physical map or at release time in order to feed the P Free List.

Next, we describe the components involved and their main interactions. Section 4.1 and Section 4.2 elaborate on how registers are allocated and released, also giving detailed examples.

General Map Table and *Last Uses Table*. The **General Map Table** (GMT) plays the role of the Map Table in conventional renaming. For each logical register, it keeps three fields:

- **vp:** current virtual-physical register identifier,
- **p:** current physical register identifier, if any,
- **v:** p field valid bit, that is, whether a physical register has already been allocated to this logical register.

The **Last Uses Table** (LUST) identifies the instruction using every logical register for the last time—the *LU* instruction. It has three fields per entry:

- **ROSid:** identifier of the *LU* instruction (its ROS address),
- **k:** kind of register use made by the *LU* instruction (src1, src2, dst),
- **c:** bit reporting whether instruction *LU* is still in the pipeline or already committed.

The renaming process in VP-LAER starts at the rename stage, where, for every instruction, two tasks proceed in parallel, namely, renaming of logical registers and scheduling of early releases. Let's consider the rename of a single instruction to make the explanation simpler.

In the first task, the General Map Table entries corresponding to the logical source registers are read in order to obtain the virtual-physical and also the physical fields once the physical allocation has already been performed (v set). Besides, a fresh virtual-physical identifier taken from the VP Free List is written into the vp field of the logical destination register and v is reset. Now, the renamed instruction can be dispatched to the Issue Window along with all the renames made (three virtual-physical registers and two, one, or zero physical registers). The instruction can also be inserted into the Reorder Structure, requiring only the three virtual-physical identifiers for release purposes. Instructions remain in both the Issue Window and the Reorder Structure until they are committed.

The second task identifies the instruction that used the logical destination register for the last time (the *LU* instruction) and schedules the early release of the involved register pair (just the pair mapped to the previous version of the destination register) for its commit. To this end, the ROS address of the *LU* instruction is obtained by reading the Last Uses Table entry corresponding to the destination register; the k field directs the scheduling towards the right *LU* instruction register. Using our terminology, this task is performed by an *NV* instruction. After this reading, the Last Uses Table is updated to reflect the current register uses. This is accomplished by spreading the ROS address of the instruction being renamed into the Last Uses Table entries pointed by all its logical register identifiers; k fields are also set with the proper values (src1,2, dst). Note that an *NV* instruction having coincident destination and source registers should self-schedule the release, skipping the Last Uses Table reading.

In order to support all the required accesses, the Last Uses Table needs a significant bandwidth; nevertheless, its small size allows an access time far below the smallest reasonable physical register file, leaving the Last Uses Table out of any critical processor path [25]. The same reasoning can be applied to the General Map Table.

As in conventional renaming, at each branch prediction, both the General Map Table and the Last Uses Table are checkpointed [17]. Therefore, the branch misprediction recovery mechanism can retrieve virtual-physical mappings and register uses properly. The physical mappings are restored from the Physical Map Table.

Issue Window. In order to support the allocation of registers in VP-LAER, an entry in the **Issue Window** (IW) has the following fields:

- **vp_src1,2; p_src1,2:** virtual-physical and physical identifiers of source operands,
- **rdy1,2:** ready bits of source operands. When an operand is ready, the corresponding physical field has a valid identifier,
- **vpd:** virtual-physical destination register identifier. It is used at execution time in order to index the Physical Map Table and to record the physical mapping,
- **pd:** physical destination register identifier. If the instruction is selected as a theft victim, this field supplies the physical identifier without using the Physical Map Table,[2]
- **exec:** execution flag. Bit indicating whether the instruction has completed its execution.

An instruction can be issued when both operands are ready. When an instruction is issued, it reads its register operands from the physical register file or from the network bypass using the p_src identifiers.

At the end of the execution, every instruction with a destination register allocates a fresh physical register for its virtual-physical destination (vpd). This physical register is written in the pd field in the Issue Window entry of the instruction and the exec bit is set. Then, the new mapping $< \mathrm{vpd}, \mathrm{pd} >$ is broadcast to the Issue Window to wake up all the instructions waiting for the vpd register. In all matching cases, the p_src field is updated with pd and the ready bit is set. This pair of registers is also broadcast to the General Map Table to update the p and v fields if some vp field matches vpd. With this, any new renamed instruction will find the physical mapping directly from the General Map Table. Finally, the new mapping is reflected in the Physical Map Table.

In VP-LAER, the allocated physical register pd is either taken from the pool of free physical registers, if any, or it is stolen from the youngest executed instruction in the Issue Window. In order to identify the instruction to be stolen, the Issue Window is searched for a younger entry with the exec flag set. If several of these entries are found, the youngest is chosen. The physical identifier is supplied from the pd field of the stolen instruction and both the General Map Table and Physical Map Table are updated to reflect that the mapping has changed. In order to reexecute the stolen instruction, the exec flag is reset in the Issue Window, enabling the issue logic to choose it again in the future. Besides, the pd field of the stolen instruction is broadcast to the Issue Window to turn off the ready bits of all matching entries. Notice that a steal operation can be done in a single cycle by tagging currently not ready operands with matching virtual-physical registers as ready and tagging currently ready operands with matching physical registers as not ready. This can be done by attaching comparators to the Issue Window busses to broadcast physical identifiers to source operands. Section 4.1 explains more details about reexecution.

Reorder Structure and *Release Queue*. Both structures are used to support register release in VP-LAER. At rename, every *NV* instruction schedules an early release by marking the ROS entry of its *LU* pair (see Fig. 6). The Reorder Structure stores the *safe* releases coming from nonspeculative *NV* instructions (Case 1 in Section 3.2), whereas the Release Queue stores *speculative* releases coming from speculative *NV* instructions (Case 2 in Section 3.2). Specifically, the left part of the Release Queue (RWC&NS) takes care of *LU* instructions still in the pipeline and the right part (RWNS) takes care of committed *LU* instructions.

Let's explain the **Reorder Structure** (ROS) role first. As regards releasing, each entry has the following fields:

- **r_1, r_2, r_d:** logical register identifiers (sources and destination),
- **vp_1, vp_2, vp_d:** virtual-physical register identifiers (sources and destination),
- **rel1, rel2, reld:** These are *safe* release schedulings. When set at commit time, they force releasing the corresponding vp_ register and its physical pair. The physical register is obtained by indexing the Physical Map Table with the vp_ identifier. We call *Release-when-Commit* (RWC) to all those bits

2. Note that pd field does not take part in the wake up logic and, thus, the entire pd array could be implemented as a separate structure in order not to increase cycle time.

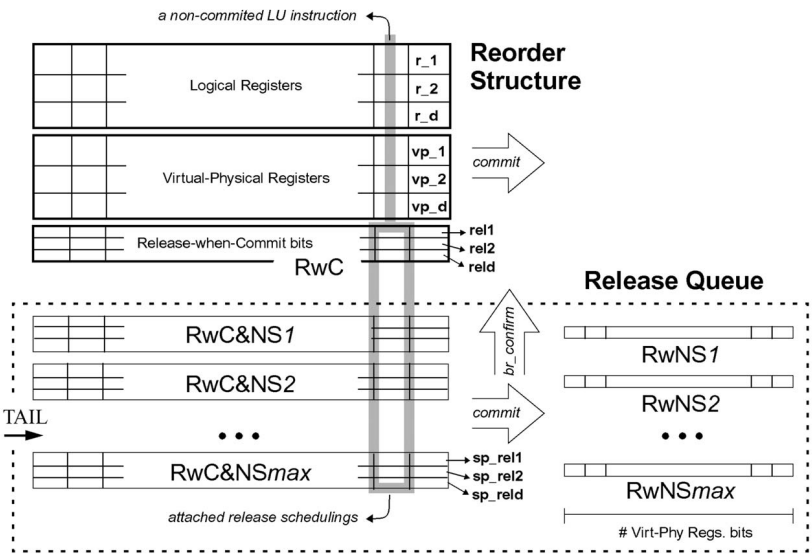

Fig. 6. Reorder Structure and Release Queue. Instructions enter from the left and commit toward the right. Branch confirmations shift up Release Queue levels.

belonging to all instructions entries in the Reorder Structure.

When renaming a nonspeculative *NV* instruction, if its *LU* pair is still in the pipeline, the suitable rel bit is set in the RWC entry occupied by the identified *LU* instruction. In this case, registers will be released when *LU* commits. On the other hand, if an *LU* instruction is already committed when renaming its nonspeculative *NV* pair, no scheduling is needed and registers can be released immediately.

As has been said, the **Release Queue** holds *speculative* releases that must be squashed if the responsible *NV* instructions become part of a wrong control path. As in a branch recovery mechanism, the Release Queue is used to checkpoint all the *speculative* register releases at every branch prediction. It has as many horizontal levels as branches pending verification the processor supports (from *1* to *max* in Fig. 6). These levels of the Release Queue are mainly managed in a FIFO way. Every time a branch is decoded, a new level is stacked at the bottom of the queue. Closer to the top are the levels related to older branches and we assume the pointer TAIL identifies the level allocated to the last branch decoded.

The left part of the Release Queue (RWC&NS) supports left-to-right shifting in order to track commit evolution just as the ROS does (*commit* arrow in Fig. 6). Therefore, when renaming a speculative *NV* instruction whose *LU* pair is still in the pipeline, the suitable sp_rel bit is set in the level pointed by TAIL, just at the entry occupied by the identified *LU* instruction (see the bottom part of the highlighted column in Fig. 6). At every branch confirmation, these *speculative* releases are shifted upward, becoming "less" *speculative* or even *safe* (arrow *br_confirm* in Fig. 6). But, it may occur that the *LU* instruction reaches the commit stage having *speculative* releases marked in the Release Queue (in RWC&NS). These are releases that are still subject to the confirmation of pending branches and cannot be lost. The same arises when, at rename of the speculative *NV* instruction, the *LU* pair is already committed. In both cases, releases have to be delayed until the confirmation of the oldest branch, the right part of the Release Queue (RWNS) keeping such releases.

Summarizing, every level n in the Release Queue is implemented by means of the following two structures:

- *Release-when-Commit & Non-Speculative* (**RWC&NS*n***). It keeps speculative releases scheduled by speculative *NV* instructions renamed past *n* branches pending verification. As all the *n* pending branches are being confirmed (before the *LU* instruction reaches its commit stage), these releases flow until reaching the RWC fields of the *LU* instruction by crossing the Release Queue border. With this, register releases will be effective at *LU* commit.
- *Release-when-Non-Speculative* bit array (**RWNS*n***). It has one bit per every virtual-physical register. It keeps *speculative* releases scheduled by speculative *NV* instructions renamed past *n* branches pending verification and whose *LU* instruction is already committed. A level receives *speculative* releases in two radically different cases. The first one arises when committing *LU* instructions with *speculative* releases attached to them; the second one when renaming an *NV* instruction whose *LU* pair is already committed. In the first case, the *LU* instruction with some sp_rel bits set in level *n* has to transfer the equivalent release information into the same level of RWNS.

  The second case arises when the commit bit is set in the Last Uses Table entry pointed by the logical destination register of the *NV* instruction being renamed. The scheduled target is the previous virtual-physical destination register of the *NV* instruction (the corresponding bit is set in the level TAIL of RWNS).

  At every branch confirmation, all levels from TAIL to top are also shifted upward. At the confirmation of the oldest branch, the top level transfers its set bits to the VP Free List and P Free List (through the Physical Map Table).

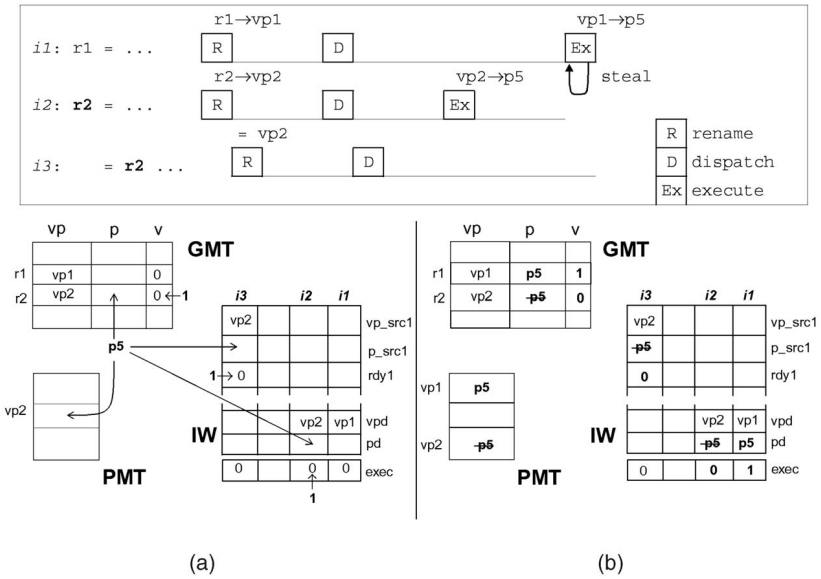

Fig. 7. Example of VP-dsy allocation in VP-LAER. (a) First, *i1*, *i2*, and *i3* are renamed and dispatched; second, arrows show changes after *i2* completes and allocates `p5`. (b) *i1* steals register `p5` from *i2* before *i3* issues.

4.1 Example of VP-dsy Allocation in VP-LAER Renaming

Fig. 7 shows a simple code and three samples of General Map Table, Issue Window, and Physical Map Table during its execution. Fig. 7a shows two snapshots, the first one after renaming instructions *i1*, *i2*, and *i3*, and the second one (with arrows) after instruction *i2* completes and allocates `p5`. Fig. 7b shows the situation after instruction *i1* executes and steals `p5`, the physical register allocated to instruction *i2*. All the time we are assuming that instruction *i3* cannot be issued.

- **Instructions *i1*, *i2*, and *i3* are renamed and dispatched,** Fig. 7a. For instructions *i1* and *i2*, the virtual-physical identifiers `vp1` and `vp2` are obtained from the VP Free List, are written into GMT, and their v bits reset. Besides, both instructions have been dispatched to IW, filling their vpd destination register fields with `vp1` and `vp2`.

  For instruction *i3*, its logical source register `r2` is renamed through the GMT. The resulting virtual-physical register—`vp2`—is written into the *i3* entry of IW while marking it as not ready. All three instructions have been marked as not executed in IW.
- **Instruction *i2* completes and allocates `p5`,** see arrows in Fig. 7a. Just before instruction *i2* ends execution, a physical register identifier is reclaimed for `vp2` from the P Free List. In the example, `p5` is supplied and the *i2* entry in IW is properly updated: `p5` is written into the pd field and the instruction is marked as executed.

  Next, in order to reflect the new map, PMT, IW, and GMT are updated with `p5`. So, PMT is indexed with `vp2` and written with `p5`. At the same time, the virtual-physical source fields in IW are associatively accessed to reflect that `vp2` is mapped to `p5` wherever `vp2` appears and to set the ready bits. To update GMT, `vp2` is associatively searched in all the vp fields, any matching entry is replaced with `p5`, and the v bit is set.
- **Instruction *i1* executes and steals `p5` from *i2*,** Fig. 7b. Let us assume that instruction *i1* finishes execution, that there are no free physical registers, and that instruction *i2* has been selected as the theft victim. For the stolen instruction, its vpd and pd fields are read from IW: $< \mathtt{vp2}, \mathtt{p5} >$ in the example. All data structures have to reflect that `vp1` is now mapped to `p5` and `vp2` is no longer associated to `p5`. In order to do that, the mappings in GMT and PMT are reversed, acting as described in the previous step.

  Because the stolen instruction *i2* must be reexecuted, the exec flag is reset in the IW, which enables the issue logic to select *i2* again in the future. Moreover, since *i2* has been executed in the past, the consumers of `vp2` (*i3* in the example) have this operand marked as ready. However, they are not ready anymore since the physical register has been stolen ($\mathrm{rdy1} = 0$ in the example). Note that a consumer instruction may be executing at the same time as `vp2` becomes detached from `p5`. Such an instruction has read a correct source operand and, thus, its result will be correct. At the time it finishes and allocates a register, it will be able to store their result and dependent instructions will be allowed to be issued.

4.2 Early Release Examples in VP-LAER Renaming

If there are no branches pending verification when renaming *NV* instructions, either the release scheduling is directly placed in the Reorder Structure (Section 4.2.1) or the release is immediately performed (Section 4.2.2). Examples where an *NV* instruction is speculative and its scheduling checkpointed on the left or right part of Release Queue appear in Section 4.2.3.

4.2.1 *Nonspeculative NV Instruction Finds Its LU Pair Not Yet Committed*

Before renaming instruction *LU* in the code of Fig. 8a, we assume the logical register `r2` is mapped to `vp2` and `p2` (see GMT contents). We will focus on the release of $< \mathtt{vp2}, \mathtt{p2} >$.

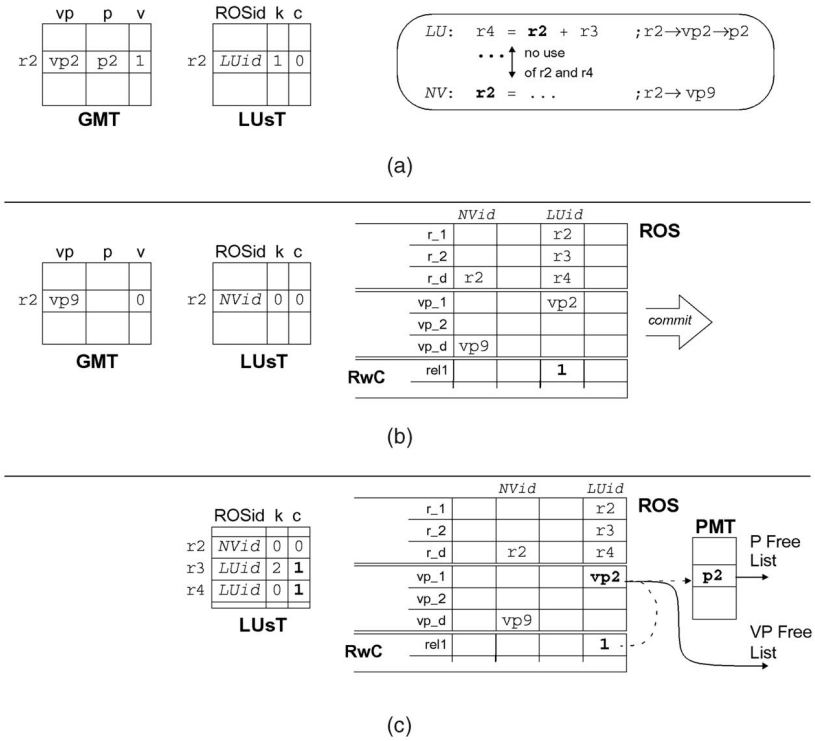

Fig. 8. (a) Example code and GMT/LUsT state after *LU* is renamed. (b) *NV* is renamed and schedules an early release of vp2 in the LUid entry. (c) $<\mathtt{vp2},\mathtt{p2}>$ release at *LU* commit and LUsT after *LU* commits.

1. **Rename of instruction LU.** This step writes the entries r2, r3, and r4 of the Last Uses Table to record register uses in program order. Fig. 8a shows the updated r2 entry in LUST. The ROSid field is set with the ROS address at which instruction *LU* is placed (*LUid*).
2. **Rename of instruction NV.** Just before renaming *NV*, the logical destination register r2 is used to index GMT and LUST and to get the fields p from GMT and $<\mathrm{ROSid},\mathrm{k},\mathrm{c}>$ from LUST. So, before renaming r2, we read: $<\mathtt{p2},\mathrm{LUid},1,0>$. Because the identified *LU* instruction is not committed (bit $\mathrm{c}=0$), the action here is to schedule a *safe* release of vp2. This is done by setting the k bit in the RWC array, just in the *LUid* entry (bit rel1 set in Fig. 8b). After that, we rename all *NV* registers and proceed as described in the Step 1. Fig. 8b shows the updated r2 entries in GMT and LUST.
3. **Commit of instruction LU.** Every logical register of instruction *LU* indexes the LUST to obtain ROSid fields, which are compared with *LUid*. Wherever a match is found, the bit c is set (see Fig. 8c). Notice that actions on bits c have to be extended to all LUST copies to achieve a proper branch misprediction recovery. Besides, as instruction *LU* commits the rel1 bit set in the RWC forces the release of vp2 and its attached register p2, which is obtained from PMT.

4.2.2 Nonspeculative NV Instruction Finds Its LU Pair Committed

Here, the action is twofold: On the one hand, we immediately release the physical register supplying it to the P Free List. On the other hand, the VP register of the previous version is directly granted to the incoming *NV* instruction (instead of supplying it to the VP Free List). Therefore the only required action in GMT is turning off the v bit. If this were the case in our previous example, the r2 entry in GMT would remain as $<\mathtt{vp2},-,0>$ and p2 would be added to the P Free List.

4.2.3 Speculative NV Instruction

All the bookkeeping done to update the Last Uses Table and identify *LU* instructions remains the same, but now the scheduled releases are *speculative* and have to be placed to the left (the *LU* instruction is still in the pipeline) or right (the *LU* instruction has already committed) parts of the Release Queue.

Fig. 9a shows an example of an *LU* instruction which has a release scheduling dependent upon one branch verification. If the instruction commits before that branch verification, the scheduling is moved from the commit front of RWC&NS*1* to RWNS*1*, waiting for the branch execution. This transfer requires decoding VP numbers to set RWNS bits (see *Mark* in Fig. 9a).

On the other hand, when the pending branch related to the level n is confirmed, the entire level n of the Release Queue (left and right parts) is *ored* with the previous $n-1$ level. At the same time, all younger conditional releases (from level TAIL to level $n+1$) *move* one level toward the ROS. The confirmation of the oldest branch ($n=1$ case) implies further action. Fig. 9b shows an example of the oldest branch confirmation with three pending branches. All register releases scheduled in RWC&NS*1* cross the Release Queue border and are *ored* with RWC. But, at the same time, the releases scheduled in RWNS*1* take effect (see the *Branch-Confirm Release* arrows in the figure). VP registers are sent (in a decodified way) to the VP Free List and PMT is

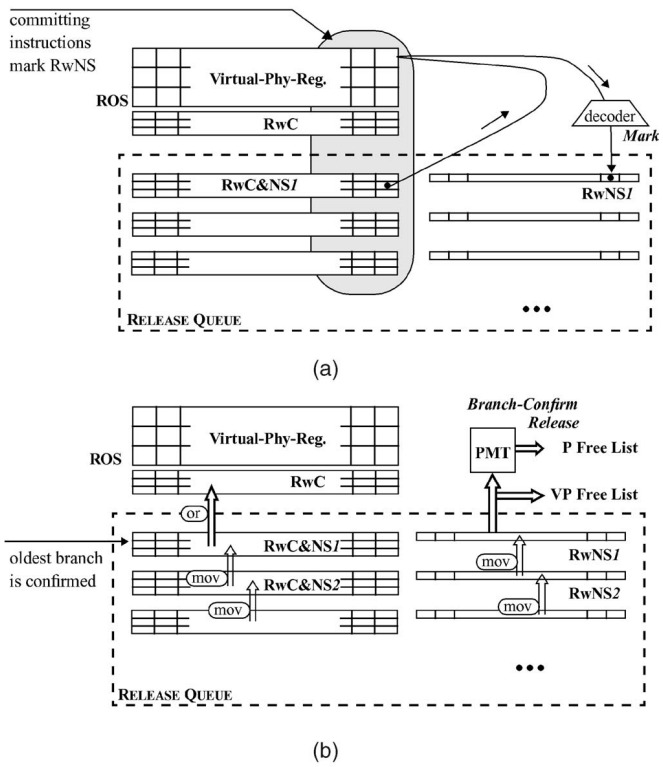

Fig. 9. Examples of the operations supported by the Release Queue. (a) Marking a bit in RWNS*1* to keep a *speculative* release coming from a committed *LU* instruction. (b) The oldest branch confirmation transforms *speculative* in *safe* releases and triggers a Branch-Confirm Release.

indexed in order to obtain the physical identifiers which are to be sent to the P Free List. We assume the interface between RWNS*1* and PMT has been designed to avoid the internal PMT address decoder.

Finally, if the prediction for the number n branch was wrong, all levels in the Release Queue from n to TAIL are cleared, then squashing all speculative releases scheduled from the mispredicted path. Therefore, TAIL is left pointing to the $n-1$ level. Notice that VP-LAER allows branches to be verified out of order.

To sum up, registers are released at three possible moments: the two we described in Section 4.2.2 (*immediate release*) and Section 4.2.1 (*commit of instruction LU*) and the oldest branch confirmation case presented in this section (*Branch-Confirm Release*). In all situations, the physical register release performed by VP-LAER takes place earlier than in the conventional scheme.

5 Experimental Results and Analysis

In this section, we first describe the experimental framework used to evaluate VP-LAER and the individual contribution of late allocation and early release. We then analyze the performance benefits of replacing conventional renaming with VP-LAER considering the two following scenarios. In Section 5.3, we assume that the register file is outside critical timing paths and, thus, processor cycle time and register file size are unrelated. On the other hand, Section 5.4 considers the opposite situation, where cycle time is closely related to the register file access time.

5.1 Experimental Framework

A cycle-based timing simulator derived from SimpleScalar v3.0 [6] has been set up. The out-of-order simulator has been modified to include physical register files (integer and FP) which are managed either with the conventional policy or with the proposed policies. Table 3 summarizes the parameters of the simulated microarchitecture. We will compare a baseline processor with conventional renaming to a similar processor enhanced with VP-LAER. The individual contributions of VP-dsy and early release will be also obtained. All reported speedups have conventional renaming as a baseline.

Ten benchmarks from the Spec95 suite are used: five integer (`compress`, `gcc`, `go`, `li`, and `perl`) and five FP programs (`mgrid`, `tomcatv`, `applu`, `swim`, and `hydro2d`). All programs were simulated to completion (by changing some inputs) excepting `tomcatv`, for which the initial part that reads a huge input file was skipped. Table 4 lists the inputs and the number of instructions executed.

5.2 Register File Utilization and IPC Analysis for a $64\text{int} + 64\text{fp}$ Register File

First, we review how Utilization improves when VP-LAER is applied to a $64\text{int} + 64\text{fp}$ register file, the design point formerly considered in Section 2.1. Fig. 10 shows the average number of Allocated registers that are in the Empty, Ready, or Idle states. As expected, there are no Empty registers and the number of Idle registers has decreased. Conversely, the average number of Ready and Free registers has increased. In particular, Ready registers increase by 19 percent and 32 percent and Idle registers decrease by 16 percent and 33 percent for integer and FP programs, respectively. Consequently, Utilization is largely improved and there are more Free registers. Utilization increases from 39 percent to 59 percent in integer codes and from 59 percent to 85 percent in FP codes, while the average number of Free registers more than doubles in both code types.

On the other hand, Fig. 11 shows the average number of instructions committed per cycle (IPC) for conventional renaming and VP-LAER. VP-dsy and early release acting alone are also plotted. A first observation is that FP programs benefit the most from any form of improved renaming because of their higher pressure on the FP register file.

Early release alone gives an average speedup of 2 percent and 5 percent, while VP-dsy alone achieves 5 percent and 24 percent for integer and FP, respectively. VP-LAER achieves 6 percent and 28 percent, which represents almost an additive behavior.

5.3 IPC versus Register File Size

If register file delay fits into the target cycle time, the performance trade offs can be established by just looking at IPC. In this situation, the processor designer can incorporate VP-LAER either as an IPC booster without changing the register file size or as a size tightener without losing performance.

VP-LAER for IPC boost, without changing the register file size. Fig. 12a illustrates how IPC varies according to the number of physical registers. We see how the IPC gap between conventional and VP-LAER renaming widens as the register file becomes tighter. In other words, VP-LAER makes the processor performance less dependent on

TABLE 3
Processor Parameters

Parameter	Value
Fetch width	8 instructions (up to 2 taken branches)
L1 I-cache	32 KB, 2-way set-assoc, 32 byte lines, 1 cycle hit time
Branch prediction	18-bit Gshare predictor with speculative updates, up to 20 outstanding branches
Window size	128 entries
Func. Units (*lat*)	8 simple int (*1*), 4 mult int (*7*), 4 ld/st, 6 simple FP (*4*), 4 multFP (*4*), 4 divFP (*16*)
Load/Store Queue	64 entries with store-load forwarding
Issue mechanism	out-of-order issue. Loads executed when all previously store addresses are known
Physical Registers	40-160 int / 40-160 FP (32 int / 32 FP logical)
L1 D-cache	32 KB, 2-way set-assoc, 64 byte lines, 1 cycle hit time
L2 Unified Cache	1 MB, 2-way set-assoc, 64 byte lines, 12 cycles hit time
Main Memory	unbounded size, 50 cycles access time
Commit width	8 instructions

register file size. In integer codes, register files having as few as 48, 56, or 64 registers become interesting because they reach an IPC close to that of a loose register file (15 percent, 9 percent, and 6 percent speedup, respectively). In floating-point codes, speedup is quite striking in the lower size range: from 70 percent to 25 percent for 40 to 72 registers. On the other hand, a 96 register file (12 percent speedup) performs as well as a loose register file.

VP-LAER for register file size reduction without losing performance. VP-LAER can be used to sustain a given IPC while reducing the number of registers, which is linearly related to the register file chip area [11]. Fig. 12b shows two sample configurations reaching 90 percent and 85 percent of the maximum average IPC, with and without VP-LAER. Area savings range between 25 percent and 30 percent, being slightly greater for FP codes.

Component analysis. As noted before, the IPC benefit obtained with VP-LAER is close to the sum of the benefit of its components acting separately. However, for very tight register files (roughly between 40 and 56 registers) and FP codes, VP-LAER is even higher than that sum. Overall, the weight of VP-dsy in IPC is dominant, whereas the early release contribution is modest. However, early release reduces the reexecution overhead significantly.

As Fig. 12c shows, early release helps reduce the number of reexecutions. For instance, on average, in a $64\text{int} + 64\text{fp}$ register file, the total number of reexecutions decreases by 20 percent and 14 percent for integer and FP codes, respectively. In `li` and `tomcatv`, this reduction reaches 34 percent and 25 percent, respectively.

TABLE 4
Benchmarks, Compaq/Alpha Fortran, and C Compilers (Maximum Optimization Level: -O5 for Fortran and -O4 Migrate for C)

Application	Inputs	Executed instr. (M)
compress	40000 e 2231	170
gcc	genrecog.i	145
go	9 9	146
li	7 queens	243
perl	scrabbl.in	47
mgrid	test (replacing the two first lines to 5 and 18)	169
tomcatv	test	191
applu	train (changing dt=1.5e-03 and nx=ny=nz=13)	398
swim	train	431
hydro2d	test (replacing ISTEP=1)	472

In order to support reexecution, we assume an Issue Window as big as the ROS and able to keep instructions until they commit. This fact also eases selective recovery for other kinds of data speculation (e.g., cache-hit or load-store independence speculation), but big Issue Windows can impact cycle time. In contrast, processors with smaller and faster Issue Windows may work at higher frequencies, but at the expense of not taking advantage of the VP-LAER's full potential unless efficient recovery mechanisms are devised. And, although this study is out of the scope of this paper, a possible solution could be to merge VP-LAER with some recent proposals of big Issue Windows which are able to operate at small cycle times [5], [8], [20].

5.4 IPS versus Register File Size

Heavily pipelined processors may have to choose between a multiple-cycle register file, possibly limiting IPC, or a single-cycle register file, possibly limiting the processor cycle time [4], [11], [34], [35]. Fig. 13 shows performance measured in BIPS assuming that processor cycle time equals the access time of the slowest register file, which is the FP register file here due to its higher number of ports. Results have been computed according to the model of Rixner et al. for a 0.18μ technology [31].

With the register file placed in a critical timing path, the designer's job is to find the tighter register file giving the higher performance. VP-LAER sets a design point which simultaneously requires fewer registers and attains more BIPS. In integer codes, the preferred size reduces from 72 to 48 registers, whereas BIPS increase by 9 percent. In FP codes, the best design shifts from 120 registers to two attractive design points, namely, 96 and 80 registers, which increase BIPS by 7.2 percent and 6 percent, respectively.

If we want to optimize the design for a processor executing both integer and FP applications, we can consider the overall workload BIPS average. In this case, the best design point shifts from 120 to 72 registers with a BIPS increase of 9.1 percent.

6 Related Work

Another approach to delaying the allocation of physical registers was proposed by Wallace and Bagherzadeh to

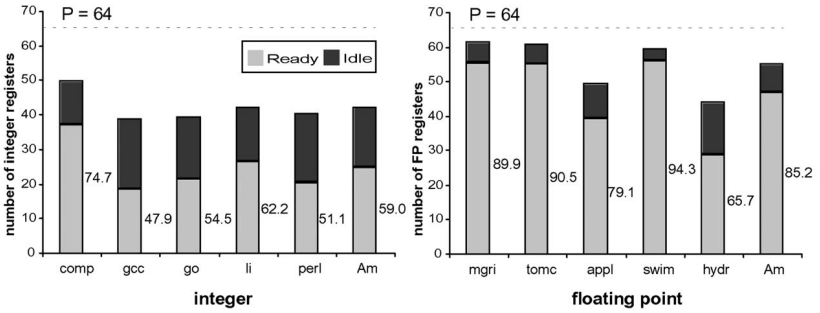

Fig. 10. VP-LAER renaming. The number of Allocated registers that are in the Ready or Idle states. Register file Utilization is listed next to the Ready bars.

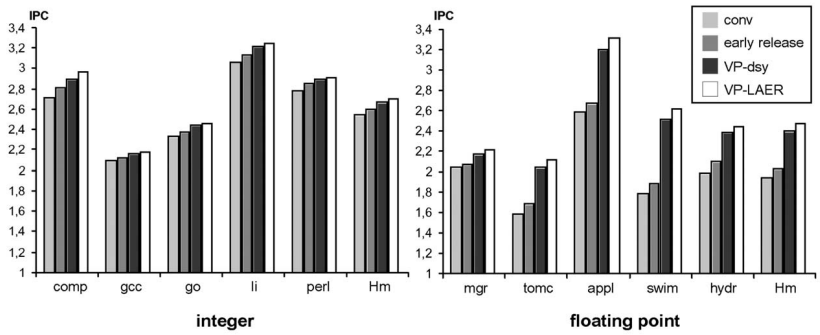

Fig. 11. IPC for a 64int + 64fp register file. The bar group to the right of each plot is the harmonic mean.

manage a merged register file organized in a scalable way. Scalability relies on splitting the register file into multiple banks with only a single write port per bank [37]. As in VP-LAER, each register-writing instruction is renamed at decode using a unique identifier (called *itag*). Later on, the functional units arbitrate for a physical place to store the result, which implies the simultaneous allocation of a bank port and a free register of that bank. This delayed allocation is called *dynamic result renaming (DRR)* and it allows dealing with the multiple banks without conflicts. Nevertheless, this scheme has the same type of deadlock hazard as virtual-physical registers have. Their proposed solution relies on shifting the processor to a scalar mode when the oldest instruction is unable to complete because all physical registers have been allocated. In scalar mode, only the oldest instruction is allowed to execute and its result will be stored in the register that this instruction will release at commit (the old_physical register recorded in the Reorder Structure). Compared with VP-dsy, i.e., the Late Allocation scheme used in VP-LAER, DRR provides a less aggressive solution to deadlock avoidance.

In [29], Park et al. also use banking to reduce register file write port requirements and, thus, energy. While the use of multiple banks reduces energy by maintaining fewer ports per bank, this port reduction results in more bank conflicts. To resolve these conflicts, the authors use the concept of virtual-physical tags (assigned at the rename stage) and delay physical register and bank allocation until writeback. Their technique is called *decoupled rename* and, by avoiding write bank conflicts, it succeeds in mitigating the performance degradation caused by write banking. To avoid the deadlock problem, they use the same number of virtual-physical and physical registers.

In the context of earlier physical register release, Farkas et al. compare an imprecise-exception early release model to the conventional one [11]. They propose releasing registers when all the instructions between two redefinitions have completed their execution instead of waiting for the last-use commit as is done in VP-LAER. Because they are mostly interested in performance bounds, no implementation was proposed for this model.

Another approach intended to release registers earlier was suggested by Moudgill et al. in [26]. In this work, they suggest releasing physical registers eagerly, as soon as the last-use instruction completes out of order. 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. Later in the same paper, they present a simplified approach that supports precise exceptions by delaying the release and associating it to the commit of the instruction computing the next version. This is the baseline in our experiments using conventional release (conv and VP-dsy).

A similar mechanism is the base approach used by Balasubramonian et al. in [2] and in [1]. In [2], a two-level register file organization is proposed to reduce the register file size requirements. The first-level (L1) register file contains only those values that are active providers for the functional units and the second-level (L2) contains those values that are waiting for precise conditions to be released. Registers are allocated from L1 at dispatch time and moved toward L2 using the Moudgill et al. early-release mechanism. Values are retained within L2 to ensure program correctness because they might be needed in the event of a branch misprediction or an excepting instruction. On the other hand, in [1], all available registers are dynamically allocated between two threads: the primary and the future thread. Once again, using the imprecise early-release mechanism, the future thread is able to make forward progress by examining a very large instruction window and by jumping far ahead to execute ready instructions. The

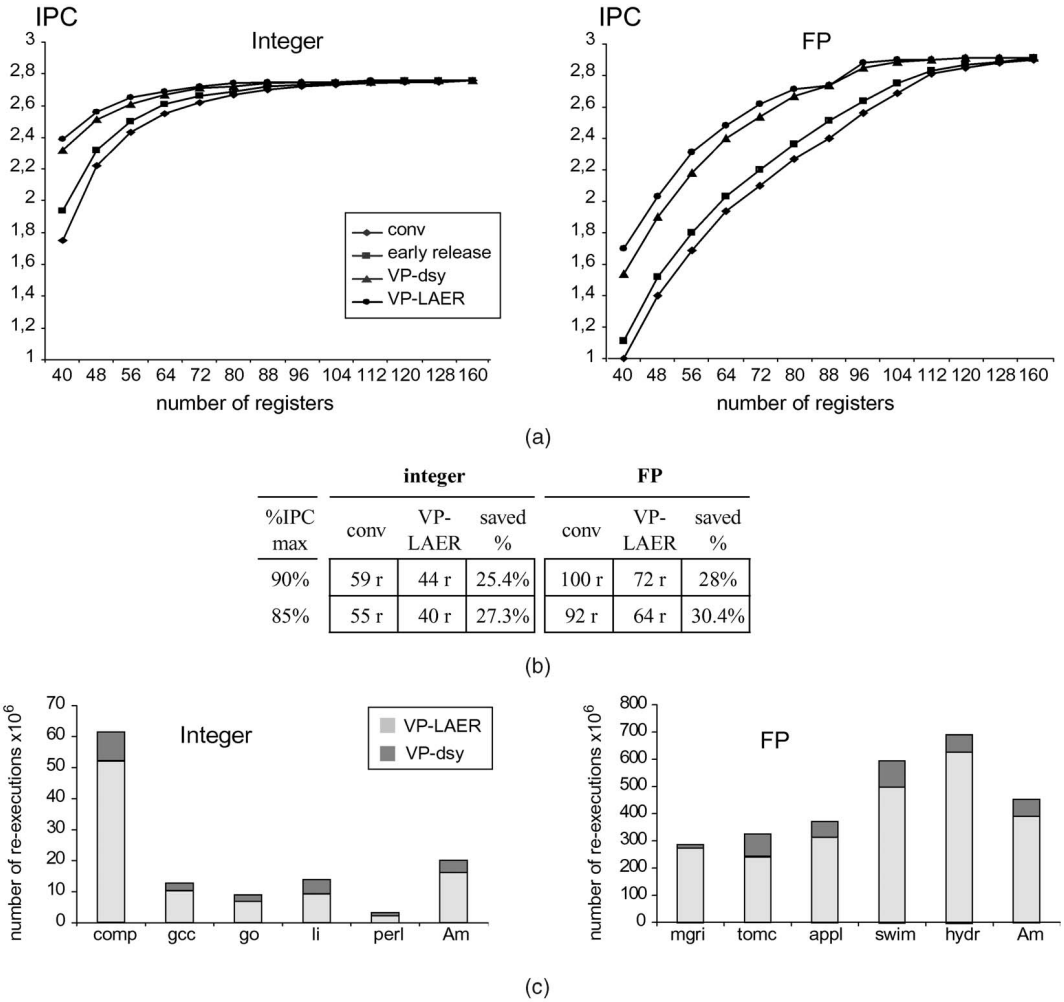

Fig. 12. (a) IPC harmonic mean versus number of physical registers for *conventional*, *early release*, *VP-dsy*, and *VP-LAER*. (b) Register file sizes affording some slight IPC degradations with and without *VP-LAER*. (c) Total number of reexecutions in *VP-dsy* and *VP-LAER* with 64int + 64fp registers.

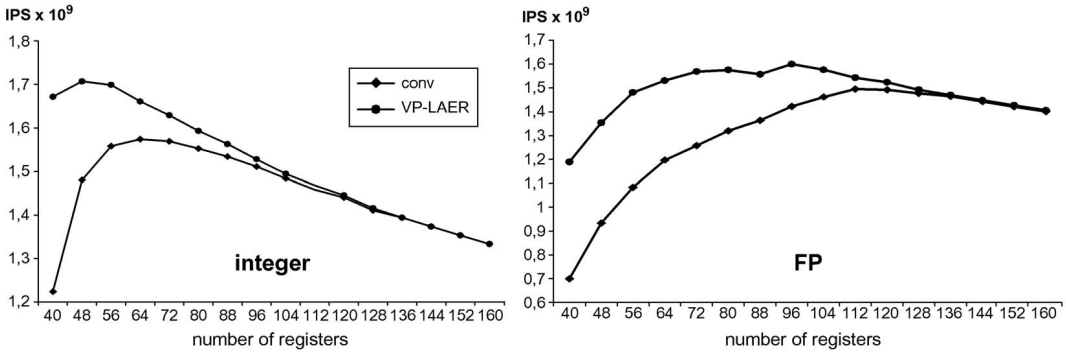

Fig. 13. BIPS harmonic mean versus number of physical registers for *conventional* and *VP-LAER* renaming.

future thread is dynamically spawned by the hardware when the primary thread runs out of physical registers and stalls. This thread consists of only a program counter and a register state and it serves the purpose of warming up the register file, data, and instruction caches, and of resolving mispredicted branches early. Later on, when the primary thread is not stalled, it will reexecute these instructions in order to ensure in-order commit and program correctness.

Other researchers also use the compiler to detect last-use instructions in order to release physical registers early [22], [23]. The compiler can identify registers containing dead values and inform the hardware. To do this, either extra instruction bits or new instructions are added to the ISA so that the compiler can schedule software releases. On the other hand, the compiler has a limited knowledge of the dynamic control flow and the release scheduling must be conservative. By contrast, hardware solutions like the Early Release approach of VP-LAER have the potential to dynamically identify last-use instructions, releasing more registers early.

7 Conclusions

In this paper, we have introduced VP-LAER, an alternative renaming mechanism for out-of-order processors. VP-LAER increases the register file utilization so that the average number of allocated physical registers having pending reads is increased. Increasing utilization will boost performance on applications with a relatively high ILP which put significant pressure on the register file. This situation typically arises in FP applications (over a wide range of register file sizes) and in integer applications with tight register file configurations.

VP-LAER optimizes register file utilization by using two complementary approaches. On the one hand, physical registers are allocated as late as possible, just at the end of instruction execution. Virtual-physical registers enable the late allocation by decoupling dependency tracking from value storage management. On the other hand, registers are released as soon as the processor can guarantee no further use, which usually occurs earlier than with the conventional approach, but the capability of a precise recovery from exceptions is still maintained.

VP-LAER is a more complex alternative than conventional renaming, but that complexity is located out of critical timing paths. Besides, the storage cost devoted to VP-LAER is quite reasonable in the context of a high-performance microprocessor.

Designs having a moderate clock rate where the register file delay fits well into the processor cycle time can either use VP-LAER to increase IPC while maintaining the size of the register file or reduce its size while maintaining IPC. For instance, a tight 64fp register file configuration managed with VP-LAER gives 28 percent more IPC. Alternatively, the number of registers can be reduced from a loose 55int to a 40int register file without any IPC loss if conventional renaming is replaced with VP-LAER.

On the other hand, when the processor cycle time is set by the register file delay, the designer goal is to find the smallest size giving the highest IPS rate. In our combined workload (integer + FP), in comparison with conventional renaming, VP-LAER increases IPS by 9.1 percent while reducing the register file size by 40 percent for a 0.18μ technology.

Acknowledgments

This work was supported by the Ministry of Education and Science of Spain (CICYT TIC2001-0995-C02-02), by the Diputación General of Aragón (Grupo Emergente de Investigación) and by the computing resources of CEPBA. The authors would like to thank the Associate Editor for his useful suggestions and the referees for their insightful comments. They would also like to thank Elena Castrillo for her contributions in editing this paper.

References

[1] R. Balasubramonian, S. Dwarkadas, and D.H. Albonesi, "Dynamically Allocating Processor Resources between Nearby and Distant ILP," *Proc. 28th Ann. Int'l Symp. Computer Architecture (ISCA '01)*, pp. 26-37, June 2001.

[2] R. Balasubramonian, S. Dwarkadas, and D.H. Albonesi, "Reducing the Complexity of the Register File in Dynamic Superscalar Processors," *Proc. 34th Ann. ACM/IEEE Int'l Symp. Microarchitecture (MICRO '01)*, pp. 237-249, Dec. 2001.

[3] A. Baniasadi and A. Moshovos, "Instruction Distribution Heuristics for Quad-Cluster, Dynamically-Scheduled, Superscalar Processors," *Proc. 33rd Ann. ACM/IEEE Int'l Symp. Microarchitecture (MICRO '00)*, pp. 337-347, Dec. 2000.

[4] E. Borch, E. Tune, S. Manne, and J. Emer, "Loose Loops Sink Chips," *Proc. Eighth Int'l Symp. High-Performance Computer Architecture (HPCA '02)*, pp. 299-310, Feb. 2002.

[5] E. Brekelbaum, J. Rupley, C. Wilkerson, and B. Black, "Hierarchical Scheduling Windows," *Proc. 35th Ann. ACM/IEEE Int'l Symp. Microarchitecture (MICRO '02)*, pp. 27-36, Dec. 2002.

[6] D. Burger and T. Austin, "The Simplescalar Tool Set v2.0," Technical Report TR-1342, Computer Science Dept., Univ. of Wisconsin-Madison, June 1997.

[7] R. Canal, J. Parcerisa, and A. González, "Dynamic Cluster Assignment Mechanisms," *Proc. Sixth Int'l Symp. High-Performance Computer Architecture (HPCA '00)*, pp. 133-144, Jan. 2000.

[8] R. Canal and A. González, "Reducing the Complexity of the Issue Logic," *Proc. 15th Int'l Conf. Supercomputing (ICS '01)*, pp. 312-319, June 2001.

[9] J.L. Cruz, A. González, M. Valero, and N.P. Topham, "Multiple-Banked Register File Architectures," *Proc. 27th Ann. Int'l Symp. Computer Architecture (ISCA '00)*, pp. 316-325, June 2000.

[10] K.I. Farkas, P. Chow, N.P. Jouppi, and Z. Vranesic, "The Multicluster Architecture: Reducing Cycle Time through Partitioning," *Proc. 30th Ann. ACM/IEEE Int'l Symp. Microarchitecture (MICRO '97)*, pp. 149-159, Dec. 1997.

[11] K. Farkas, N. Jouppi, and P. Chow, "Register File Considerations in Dynamically Scheduled Processors," *Proc. Second Int'l Symp. High-Performance Computer Architecture (HPCA '96)*, pp. 40-51, Feb. 1996.

[12] A. González, J. González, and M. Valero, "Virtual-Physical Registers," *Proc. Fourth Int'l Symp. High-Performance Computer Architecture (HPCA '98)*, pp. 175-184, Jan.-Feb. 1998.

[13] A. González, M. Valero, J. González, and T. Monreal, "Virtual Registers," *Proc. Third Int'l Conf. High Performance Computing (HiPC '97)*, pp. 364-369, Dec. 1997.

[14] L. Gwennap, "Intel's P6 Uses Decoupled Superscalar Design," *Microprocessor Report*, vol. 9, no. 4, pp. 9-15, Feb. 1995.

[15] L. Gwennap, "Mips r12000 to Hit 300 Mhz," *Microprocessor Report, Micro Design Resources*, vol. 11, no. 13, pp. 1-4, Oct. 1997.

[16] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The Microarchitecture of the Pentium 4 Processor," *Intel Technology J. Q1*, Feb. 2001.

[17] W.W. Hwu and Y.N. Patt, "Checkpoint Repair for Out-of-Order Execution Machines," *Proc. 14th Ann. Int'l Symp. Computer Architecture (ISCA '87)*, pp. 18-26, June 1987.

[18] R.M. Keller, "Look-Ahead Processors," *ACM Computing Surveys*, vol. 7, no. 4, pp. 177-195, Dec. 1975.

[19] R.E. Kessler, "The Alpha 21264 Microprocessor," *IEEE Micro*, vol. 19, no. 2, pp. 24-36, Mar./Apr. 1999.

[20] A.R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg, "A Large, Fast Instruction Window for Tolerating Cache Misses," *Proc. 29th Ann. Int'l Symp. Computer Architecture (ISCA '02)*, pp. 59-70, May 2002.

[21] D. Levitan, T. Thomas, and P. Tu, "The PowerPC 620 Microprocessor: A High-Performance Superscalar Risc Microprocessor," *Proc. 40th IEEE CS Int'l Conf., (COMPCON '95)*, pp. 285-291, Mar. 1995.

[22] J.L. Lo, S.S. Parekh, S.J. Eggers, H.M. Levy, and D.M. Tullsen, "Software-Directed Register Deallocation for Simultaneous Multithreaded Processors," *IEEE Trans. Parallel and Distributed Systems*, vol. 10, no. 9, pp. 922-933, Sept. 1999.

[23] M.M. Martin, A. Roth, and C.N. Fischer, "Exploiting Dead Value Information," *Proc. 30th Ann. ACM/IEEE Int'l Symp. Microarchitecture (MICRO '97)*, pp. 125-135, Dec. 1997.

[24] T. Monreal, A. González, M. Valero, J. González, and V. Viñals, "Delaying Physical Register Allocation through Virtual-Physical Registers," *Proc. 32nd Ann. ACM/IEEE Int'l Symp. Microarchitecture (MICRO '99)*, pp. 186-192, Nov. 1999.

[25] T. Monreal, V. Viñals, A. González, and M. Valero, "Hardware Schemes for Early Register Release," *Proc. Int'l Conf. Parallel Processing (ICPP '02)*, pp. 5-13, Aug. 2002.

[26] M. Moudgill, K. Pingali, and S. Vassiliadis, "Register Renaming and Dynamic Speculation: An Alternative Approach," *Proc. 26th Ann. Int'l Symp. Microarchitecture (MICRO '93)*, pp. 202-213, Nov. 1993.

[27] K. Nowka and M. Flynn, "Wave Pipelining of High Performance CMOS Static Ram," Technical Report TR-94/615, Computer Systems Laboratory, Jan. 1994.
[28] S. Palacharla, N.P. Jouppi, and J.E. Smith, "Complexity-Effective Superscalar Processors," *Proc. 24th Ann. Int'l Symp. Computer Architecture (ISCA '97)*, pp. 206-218, June 1997.
[29] I. Park, M.D. Powell, and T.N. Vijaykumar, "Reducing Register Ports for Higher Speed and Lower Energy," *Proc. 35th Ann. ACM/IEEE Int'l Symp. Microarchitecture (MICRO '02)*, pp. 171-182, Dec. 2002.
[30] Y.N. Patt, S.J. Patel, M. Evers, D.H. Friendly, and J. Stark, "One Billion Transistors, One Uniprocessor, One Chip," *Computer*, vol. 30, no. 9, pp. 51-57, Sept. 1997.
[31] S. Rixner, W. Dally, B. Khailany, P. Mattson, U. Kapasi, and J. Owens, "Register Organization for Media Processing," *Proc. Sixth Int'l Symp. High-Performance Computer Architecture (HPCA '00)*, pp. 375-386, Jan. 2000.
[32] D. Sima, "The Design Space of Register Renaming Techniques," *IEEE Micro*, vol. 20, no. 5, pp. 70-83, Sept./Oct. 2000.
[33] J.E. Smith and A.R. Pleszkun, "Implementation of Precise Interrupts in Pipelined Processors," *Proc. 12th Ann. Int'l Symp. Computer Architecture (ISCA '85)*, pp. 36-44, June 1985.
[34] D.M. Tullsen, S.J. Eggers, J.S. Emer, H.M. Levy, J.L. Lo, and R.L. Stamm, "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor," *Proc. 23rd Ann. Int'l Symp. Computer Architecture (ISCA '96)*, pp. 191-202, May 1996.
[35] D.M. Tullsen, S.J. Eggers, and H.M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," *Proc. 22nd Ann. Int'l Symp. Computer Architecture (ISCA '95)*, pp. 392-403, June 1995.
[36] D.W. Wall, "Limits of Instruction-Level Parallelism," *Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '91)*, pp. 176-188, Apr. 1991.
[37] S. Wallace and N. Bagherzadeh, "A Scalable Register File Architecture for Dynamically Scheduled Processors," *Proc. 1996 Conf. Parallel Architectures and Compilation Techniques (PACT '96)*, pp. 179-184, Oct. 1996.
[38] K.C. Yeager, "The Mips R10000 Superscalar Microprocessor," *IEEE Micro*, vol. 16, no. 2, pp. 28-40, Apr. 1996.
[39] J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero, "Two-Level Hierarchical Register File Organization for Vliw Processors," *Proc. 33rd Ann. ACM/IEEE Int'l Symp. Microarchitecture (MICRO '00)*, pp. 137-146, Dec. 2000.
[40] V.V. Zyuban and P.M. Kogge, "Inherently Lower-Power High-Performance Superscalar Architectures," *IEEE Trans. Computers*, vol. 50, no. 3, pp. 268-285, Mar. 2001.

Teresa Monreal received the MS degree in mathematics and the PhD degree in computer science from the Universidad de Zaragoza in 1991 and 2003, respectively. She is an assistant professor in the Informática e Ingeniería de Sistemas Department at the University of Zaragoza, Zaragoza, Spain. Her research interests are register management, superescalar processors, and register file optimization. She is also a member of the Grupo de Arquitectura de Computadores de la Universidad de Zaragoza **(gaZ)**.

Víctor Viñals received the MS degree in telecommunication and the PhD degree in computer science from the Universitat Politècnica de Catalunya (UPC) in 1982 and 1987, respectively. Currently, he is an associate professor in the Informática e Ingeniería de Sistemas Department at the University of Zaragoza, Zaragoza, Spain. His research interests include processor microarchitecture, memory hierarchy, and parallel computer architecture. He is member of the ACM, the IEEE, and the IEEE Computer Society. He is also founding member of the Juslibol Midday Runners.

José González received the MS and PhD degrees from the Universitat Politècnica de Catalunya (UPC). In January 2000, he joined the Computer Engineering Department of the University of Murcia, Spain, and became an associate professor in June 2001. In March 2002, he joined the Intel Barcelona Research Center, where he is a senior researcher. Currently, he is working on new paradigms for the IA-32 family, in particular, Thermal and Power-Aware clustered microarchitectures. He is member of the IEEE Computer Society.

Antonio González received the MS and PhD degrees from the Universitat Politècnica de Catalunya (UPC), Barcelona, Spain. He has been a faculty member of the Computer Architecture Department of UPC since 1986 and he is currently a professor in this department. He leads the Intel-UPC Barcelona Research Center, whose research focuses on new microarchitecture paradigms and code generation techniques for future microprocessors. His research has focused on computer architecture, compilers, and parallel processing, with a special emphasis on processor microarchitecture and code generation. He has published more than 150 papers in the areas power-aware microarchitectures, clustered microarchitectures, speculative multithreaded processors, data value and data dependence speculation and reuse, cache architectures, register file architecture, modulo scheduling, code analysis and optimization, mapping parallel algorithms to multicomputers, prolog-oriented architectures, instruction fetching mechanisms, and digital image processing. He is an associate editor of the *IEEE Transactions on Parallel and Distributed Systems*, *ACM Transactions on Architecture and Code Optimization*, and *Journal of Embedded Computing*. He has served on more than 50 program committees for international symposia in the field of computer architecture, including ISCA, MICRO, HPCA, PACT, ICS, ICCD, ISPASS, CASES, and IPDPS. He has been program cochair for ICS 2003, ISPASS 2003, and MICRO 2004. He is a member of the IEEE Computer Society.

Mateo Valero received the MS degree from the Technical University of Catalonia (UPC) in 1974 and the PhD degree UPC in 1980. He has been teaching at UPC since 1974 and, since 1983, he has been a full professor in the Computer Architecture Department. He has served as the Computer Architecture Department chair (1983-1984, 1986-1987, 1989-1990, and 2001-current) and the dean of the Computer Science School (1984-1985). His research topics are centered in the area of computer architecture, with a special emphasis on high-performance computers. He has coauthored more than 200 publications. He has served on organization committees for more than 180 international conferences. He has been an associate editor for several journals, such as the *IEEE Transactions on Parallel and Distributed Systems* and *Parallel Programming Languages*, and as a guest editor of special issue for *IEEE Transactions on Computers* and *Computer*. His research has been recognized with several awards, including the King Jaime 1 by the Geralitat Valenciana presented by the Queen of Spain, the Spanish national award "Julio Ray Pastor" to recognize research on IT technologies by the Spanish Ministry of Science and Technology, presented by the King of Spain, and the "Distinction to recognize and promote research at the university" presented by the Generalitat, Government of Catalonia. In December 1994, he became a founding member of the Royal Spanish Academy of Engineering. In 1998, he was appointed "Favorite Son" of his bith town, Alfamén (Zaragoza). In 2001, he was made a fellow of the IEEE and, in 2002, he was named an Intel Distinguished Research Fellow and a fellow of the ACM.