Speculative Early Register Release

Jesús Alastruey, Teresa Monreal, Víctor Viñals
gaZ-I3A-HiPEAC, Universidad de Zaragoza
Spain
{jalastru, tmonreal, victor}@unizar.es

Mateo Valero
DAC-BSC-HiPEAC, UPC
Spain
mateo@ac.upc.edu

Abstract

The late release policy of conventional renaming keeps many registers in the register file assigned in spite of containing values that will never be read in the future. In this work, we study the potential of a novel scheme that speculatively releases a physical register as soon as it has been read by a predicted last instruction that references its value. An auxiliary register file placed outside the critical paths of the processor pipeline holds the early released values just in case they are unexpectedly referenced by some instruction. In addition to demonstrate the feasibility of a last-use predictor, this paper also analyzes the auxiliary register file (latency and size) required to support a speculative early release mechanism that uses a perfect predictor. The obtained results set the performance bound that any real speculative early release implementation is able to reach. We show that in a processor with a 64int+64fp register file, a perfect early release supported by an unbounded auxiliary register file has the potential of speeding up computations up to 23% and 47% for SPECint2000 and SPECfp2000 benchmarks, respectively. Speculative early release can also be used to reduce register file size without losing performance. For instance, a processor with a conventionally managed 96int+96fp register file could be replaced for equal IPC with a 64int+64fp register file managed with perfect early register release and backed with a 64int+64fp auxiliary register file, this representing a 12% IPS (Instructions Per Second) increase if the processor frequency were constrained by the register file access time.

Categories and Subject Descriptors

C.1.1 [Computer Systems Organization]: Processor Architectures — Single Data Stream Architectures.

General Terms

Design, Performance, Measurement.

Keywords

Register renaming, physical register release, register file optimization.

CF'06, May 3–5, 2006, Ischia, Italy.

1. INTRODUCTION

Current designs of monolithic register files in dynamically scheduled superscalar processors implement a register management scheme where the release of a physical register is delayed until the commit stage of its redefining instruction [8][10][17]. While this is the simplest way for recovering a value contained in the register after a branch misprediction or an exception, it is also the most conservative one in terms of register reuse and pipeline stalls. Such problems are supposed to become more critical with the increase in instruction window size and in issue width expected in next generation processors. New proposals are needed in order to overcome such a conservative management in an efficient and complexity-aware form.

Several works have proposed some form of early register release to speedup computations, save register file area or reduce energy consumption [1][5][11][13]. But no one has tried to abstract from any concrete implementation and show the potentials of early register release when it is pushed to its limits: reuse any physical register as soon as the last consumer has read it, no matter the next register redefinition has entered the pipeline or not.

This work analyzes the performance potential of a novel and more aggressive policy of speculative early release of physical registers. By applying this policy, a physical register is released by means of predicting its last consumer in program order. As a result, it is reduced the average time a physical register remains allocated, so performance could be either increased or maintained with a smaller register file. In order to ensure correctness in presence of mispredictions, the value read for the predicted last consumer is saved in an auxiliary store placed outside of critical paths. Unexpected uses of prematurely discarded values (code of exception handlers, branch mispredictions) can be solved by recovering the value either from the register file, without paying any penalty, or from the auxiliary store, depending on the required physical register has been overwritten or not.

By delaying the release of a physical register until the commit stage of a redefining instruction, the conventional release mechanism makes sure that the value stored in that register will not be read anymore. However, there are other times, prior to the commit stage, where it may be possible to discard the value that a register stores. In this context, several register management techniques have been proposed to advance register release [11][13]. In [11] a register is released when all consumers have read the value and the redefining instruction becomes non-speculative (it can not be squashed because of a branch misprediction). In [13] the release responsibility is transferred to the *last-use instruction* that consumes the value, but once again, register release is only done after the redefining instruction becomes non-speculative.

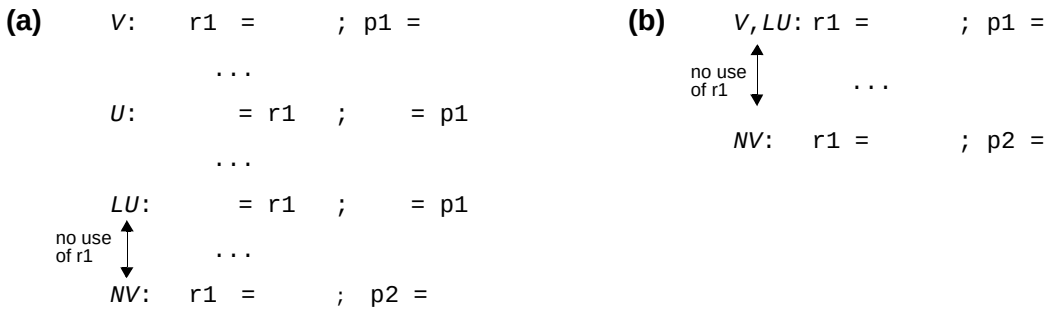

Figure 1. Instructions with last-use operands, either a source operand (a) or a destination operand (b).

In the same context of early register release, other designs try to be more aggressive by relaxing the non-speculative condition of the redefining instruction to allow a register release even though the redefining instruction is speculative [1][5]. The condition these *Speculative Early Release* (*SER*, for short) schemes still demand is the renaming of the redefinition. In contrast, our SER proposal is the first one that does not impose delaying the register release until the redefining instruction is renamed.

In order to be precise, an SER proposal has to address two issues. On the one hand, the conditions under which a physical register can be released earlier -because its value is expected not to be used anymore-. On the other hand, how to recover the value if the early release becomes premature. The referenced work addresses the first issue by tracking continuously the selected conditions, but the hardware needed for this kind of polling is complex [1][5]. Concerning this, our proposal predicts an instruction that reads a value for the last time, the *predicted last-use instruction*, and associates to that instruction the release of the corresponding physical register.

To solve the second problem, previous work saves an early released value to a register placed in a second level register file [1] or in a *shadow* register file (*checkpointed register file* in [5]). In both cases, when recovering from a branch misprediction, all values released (prematurely) due to redefinitions arising from a mispredicted code path have to return to their previous physical registers within the main register file. Because those reinserted values may not be read anymore, register pressure may remain high. Instead, we propose a simple auxiliary register file where released values are saved, and where consumer instructions appearing after an early release, *unexpected uses*, are redirected to read theirs operands. The use of virtual tags associated to physical registers can solve this redirection in a simple way [12]. Virtual tags will address instructions to locate their operand values either in the register file -if the value still remains in the released register- or in the auxiliary register file when the physical register has been overwritten.

We present two main contributions in this work. First, the concept of *patterns of use* is defined and used to characterize both last-use patterns and last-use instructions. This characterization demonstrates the feasibility of a last-use predictor to support an SER policy. Second, the potential of this novel policy is evaluated for a superscalar processor, and exploited by either boosting performance of reducing the size of the physical register file. The promising results obtained encourage future research on cost-effective implementations.

The remainder of this paper is organized as follows: Section 2 discusses conventional register release. Section 3 shows the rationale behind our SER policy, introduces the concept of *patterns of use*, analyzes the behaviour of last-use instructions in SPEC2000 benchmarks, and suggests some basic elements to exploit SER based on last-use prediction. Section 4 presents the experimental methodology and the potential gains that a perfect speculative early release policy achieves. Section 5 reviews the related work. Section 6 summarizes the conclusions of the paper.

2. CONVENTIONAL RELEASE. MOTIVATION

In this section we review the conventional model for renaming registers and give experimental evidence of the low efficiency in its releasing part.

The conventional release of physical registers is inefficient in a sense that a register is not released until the commit stage of the next redefinition of the corresponding logical register. This model is used in many current processors such as Alpha 21264, MIPS R10/12K and Intel P4 [7][8][10][17]. This releasing mechanism has a simple implementation supporting both control speculation and precise exception recovery. However, it releases registers late, causing unnecessary stalls if the processor runs out of physical registers while having registers holding values that will not be read anymore.

Instructions named *V* and *NV* in Figure 1 create two consecutive versions for the same logical destination register `r1`. Free physical registers `p1` and `p2` are mapped to `r1` in the rename stage of *V* and *NV*, respectively. *U* and *LU* are intermediate instructions having `r1` as operand. We say an instruction is the *last use* of any of its registers (*LU* for `p1` in Figure 1), when it is the last instruction in program order that references the operand value, either to consume it (Figure 1a) or to produce it (Figure 1b). A conventional register release will not return `p1` to the free list until *NV* commits.

Figure 2a shows the states of physical register `p1` under the conventional mechanism. `p1` changes from Free to Allocated when instruction *V* is renamed, and returns to the Free state when *NV* is committed. During the cycles `p1` remains Allocated, it passes through three more states. Until *V* executes, `p1` does not contain any value and we say it is in the Empty state. After the writeback of instruction *V*, `p1` contains the value to be read by its consumer instructions. `p1` is now in the Used state. The Non-Used state begins with the read of the last-use instruction (*LU*) and ends when *NV* commits.

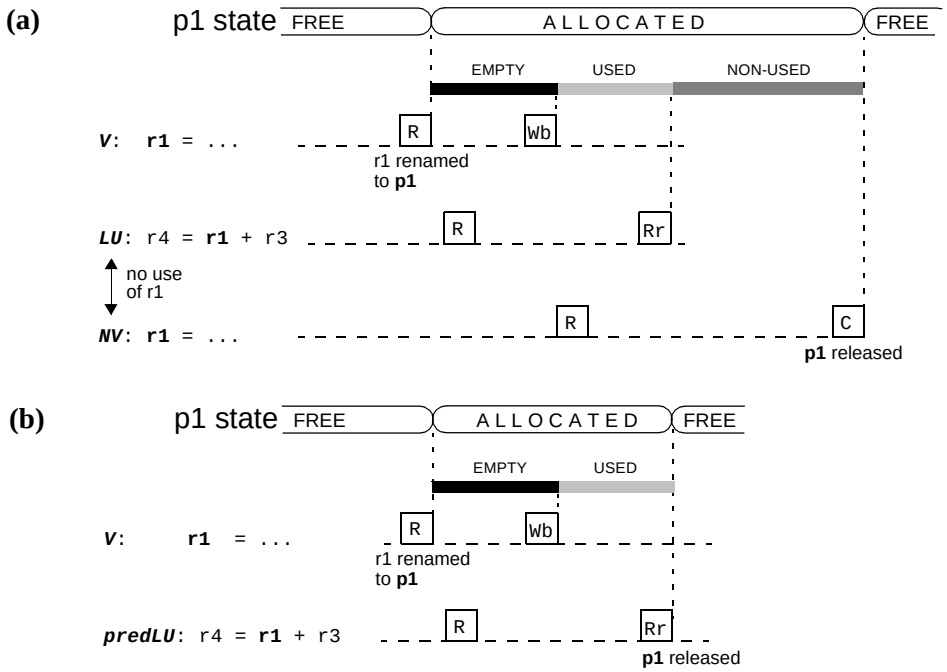

Figure 2. Example of state evolution of physical register `p1` under a conventional rename policy (a) and under an SER policy based on last-use prediction (b). R = Rename, Rr = Register read, Wb = Writeback, C = Commit.

The state evolution of physical register `p1` under our SER scheme can be observed in Figure 2b, where *predLU* instruction has been signalled by the last-use predictor. Releasing `p1` after its predicted last read completely eliminates the Non-Used state, significantly reducing register pressure. According our simulations of SPEC2000, it is possible to advance the releasing an average of 41 and 64 cycles for integer and floating point programs, respectively. It is also interesting to note how many destination registers hold a value that will never be read (9% for integer programs). When the predictor labels a destination register as last-use, our policy does not allocate any physical register. This action will further reduce the register file pressure.

To get an insight into the inefficiency of conventional releasing and the potential of the perfect SER scheme presented in Section 4, we have measured the average number of Allocated registers being either Empty, Used or Non-Used in an 8-way superscalar processor detailed in Section 4. Figure 3 shows conventional releasing (left bars) and perfect SER releasing (right bars). It can be observed the high pressure that a conventional release imposes in both integer and floating point codes. In both kind of codes, the average number of Free registers across applications is very low (9% and 11% for int and fp, respectively), this number being near zero in almost half of fp programs. And due to the late release of physical registers, a large number of registers is maintained in the Non-Used state: on average, nearly half of the Allocated registers are Non-Used. Notice that *eon, gcc* and *swim* have more than 60% of all the Allocated registers in this state.

Figure 3 also shows the potential effect of the perfect SER scheme to the register states. The most important effect would be the register pressure reduction due to the elimination of the Non-Used state. On average, under perfect SER, up to 32% and 27% of the integer and floating point registers becomes Free. In addition, the elimination of the Non-Used state also implies an appreciable growth in Empty registers and an increase in Used registers.

3. SER-LUP, SPECULATIVE EARLY RELEASE BASED ON LAST-USE PREDICTION

SER-LUP releases a physical register based on the prediction that a given instruction is using it for the last time before being redefined. SER-LUP, in contrast with previous proposals, can release a physical register even when the register redefinition is yet far from being fetched, achieving good resource usage when the Reorder Buffer becomes full and several register redefinitions are about to be renamed.

In the frequent case of an instruction having a last-use prediction on a source register, SER-LUP labels the instruction so that the value read from the register file is sent to a separate store and the physical register identifier is immediately released. We call this separate store *auxiliary register file* (XRF). Instructions with a predicted last-use destination register write their results directly to XRF, and the renaming mechanism does not allocate any physical register. This avoids useless writes to the register file as well as reduces the register pressure, specially for integer codes as can be seen below.

Due to several reasons we detail below, an instruction may require a value saved in XRF. In that case, we say the instruction performs an *unexpected use* of a (perhaps prematurely) released register. In order to manage the location of register operands, either present in the register file or in XRF, we use virtual tags to represent register dependences as suggested in [12].

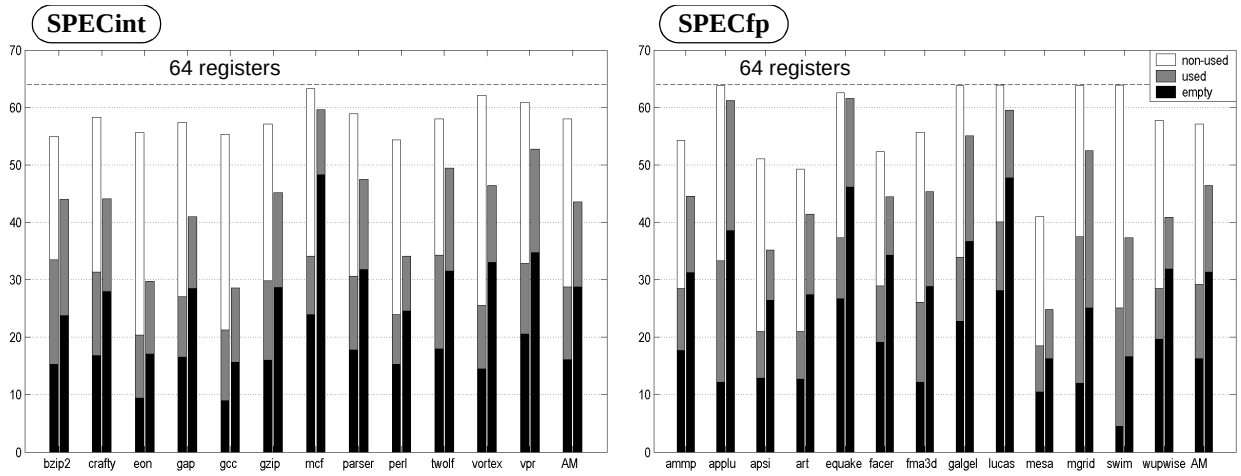

Figure 3. Average number of Allocated registers being either in Empty, Used, or Non-Used states for a processor with a 64int+64fp register file. For each benchmark, the two bars represent figures for conventional (left) and perfect SER policy (right).

```
                                        last-use
                                        s1 s2 d (y=yes, n=no)
        addq      r1, r2 -> r3    ;     n  n  n
        stl       r1 -> 0(r3)     ;     n  n  -
        ldl       0(r3)-> r3      ;     y  -  y
        bne       r2, A           ;     n  -  -    taken branch
         ...
A:      ldl       8(r1) -> r3     ;     y  -  n
        addq      r3, r2 -> r1
```

Figure 4. Execution of a code fragment showing several patterns of use. Note that if this code executes again by following the not-taken path, the pattern of use of `ldl 0(r3)-> r3` may change.

The following subsections analyze the feasibility of a last-use predictor and outline the auxiliary store (XRF) required to exploit SER-LUP. It is not the goal of this paper to search for a specific implementation, so we do not detail the remaining structures needed.

3.1 Patterns of register use

During program execution, instructions can be classified according to the future use of the registers they reference. So, any register whose next reference is a write can be labelled as a last-use. Otherwise, a register reference having one or more reads before being written is labelled as a non last-use. We define the *pattern of use* of an instruction as the set of labels of all its registers. Different instances of the same static instruction can have different patterns of use, as illustrated by Figure 4.

Next, the dynamic frequency and predictability of the patterns of use is analyzed in order to assess the viability of a last-use predictor.

3.1.1 Dynamic frequency of patterns of use

Figure 5 shows the distribution of patterns of use found in SPEC2000. According to the number of last-use source registers (2, 1 or 0) and the number of last-use destination registers (1 or 0), we can distinguish among six patterns of use, namely: 2s+1d, 2s+0d, 1s+1d, 1s+0d, 0s+1d, 0s+0d. There is no distinction between last use of source 1 and source 2.

Our experiments show that on average, almost 60% of instructions have a pattern with some last-use register. The most common pattern is “last-use of one source register” (1s+1d and 1s+0d), showing a 42% and a 40% on average for integer and floating point, respectively. “Last-use of both source registers” (2s+1d and 2s+0d) is observed in 10% and 22% of the executed instructions (int and fp, respectively). Finally, notice that for the integer benchmarks, 9% of destination registers hold a value that will never be read and under SER-LUP will not need a physical register allocation.

These figures give an insight into the continuous renewal of the values stored in the physical registers. Also, it can be stated that the release of at least one physical register could be advanced for more than half of the instructions (around 60%).

3.1.2 Predictability of patterns of use

Different instances of the same static instruction can change the pattern of use depending on the execution path. Figure 6 shows the number of different patterns of use experienced by every static instruction. Only committed instances have been taken into account for each static instruction. For both SPECint2000 and SPECfp2000, nearly 100% of all static instructions present at most two different patterns, and almost 90% and 95% of static instructions present always the same pattern of use for integer and floating point benchmarks, respectively.

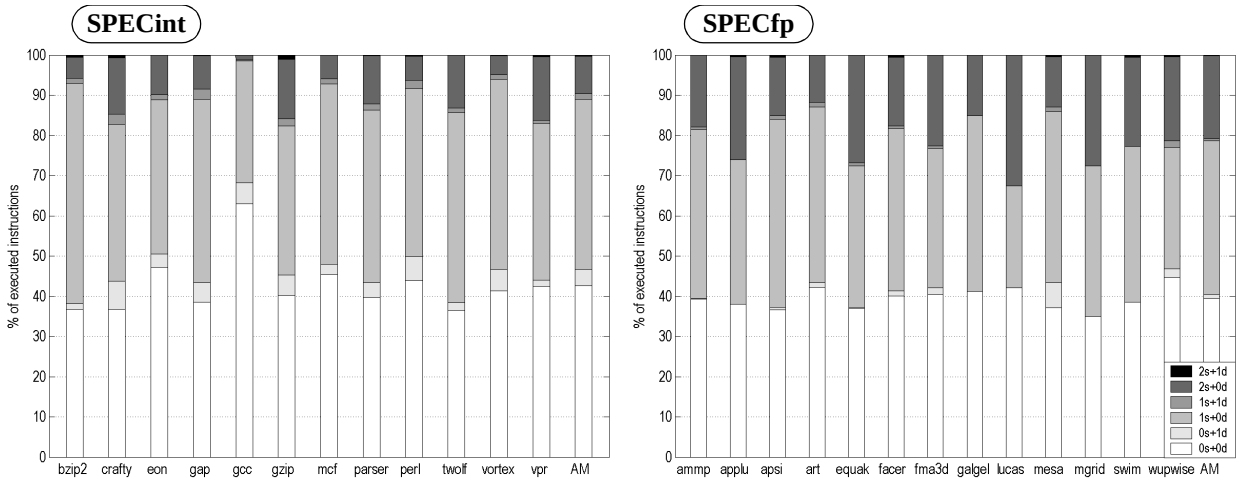

Figure 5. Distribution of patterns of use. AM is the arithmetic mean.

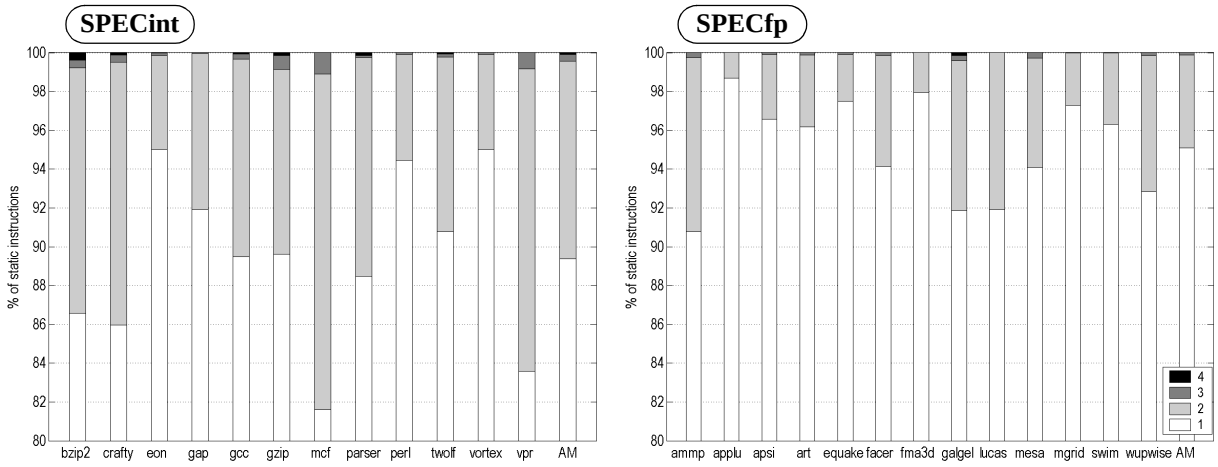

Figure 6. Predictability of patterns of use: distribution of static instructions according to the number of different patterns of use observed in their committed instances. Please, note the offset of the y-axis.

Therefore, we can conclude that conditional branches do not affect too much the pattern of use shown by static instructions. This fact encourages the design of low-complexity last-use predictors, which can rely on associating pattern of use to static instructions, instead of to dynamic instances. Next Section presents and evaluates a simple last-use predictor able to capture the register behaviour with a limited memory budget.

3.2 Last-Use predictor

A last-use predictor (LUpred) could be organized as an associative cache indexed by program counter where each entry holds a pattern of use for a single instruction having at least one last-use in a past execution. Figure 7 shows a two-way set associative LUpred with two required fields per entry: address tag and 3-bit vector representing pattern of use. A table located in the commit stage tracks last-use instructions and trains LUpred by adding to the current pattern the observed last-use register. This simple update policy "sticks" last-use predictions to an instruction even though its subsequent executions do not exhibit the same pattern. Untraining is only possible by means of entry replacement.

Next we analyze how misprediction rates vary with predictor size. Figure 8 shows the percentage of mispredictions for all register references. The five bars of each group, from left to right, represent a 4-way LUpred with 32, 128, 512, 2k and 8k entries. Mispredictions are split into two classes:

- misses when predicting "last-use register" (solid bar): a register is labelled as last-use but it is really not. For instance, predicting a pattern of use `<n,y,n>` in the first instruction in Figure 4 has a misprediction of this kind, which leads to the prematurely release of a physical register. So, it may impact performance because of the more costly access to XRF.
- misses when predicting "not a last-use register" (hollow bar): they occur when a true last-use register is not identified. For instance, predicting a pattern of use `<y,-,n>` in the first `ldl` in Figure 4 shows this kind of misprediction, which implies a lost opportunity of releasing a register with a dead value.

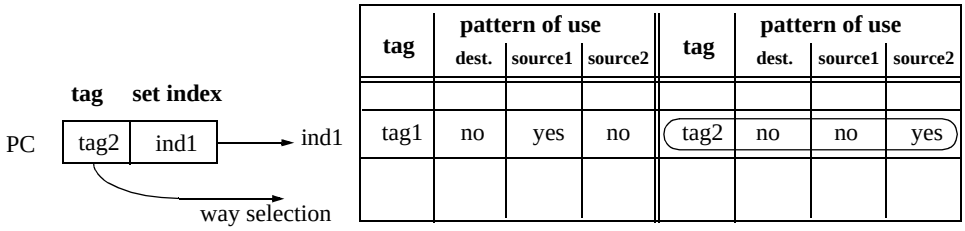

Figure 7. Two-way set associative last-use predictor.

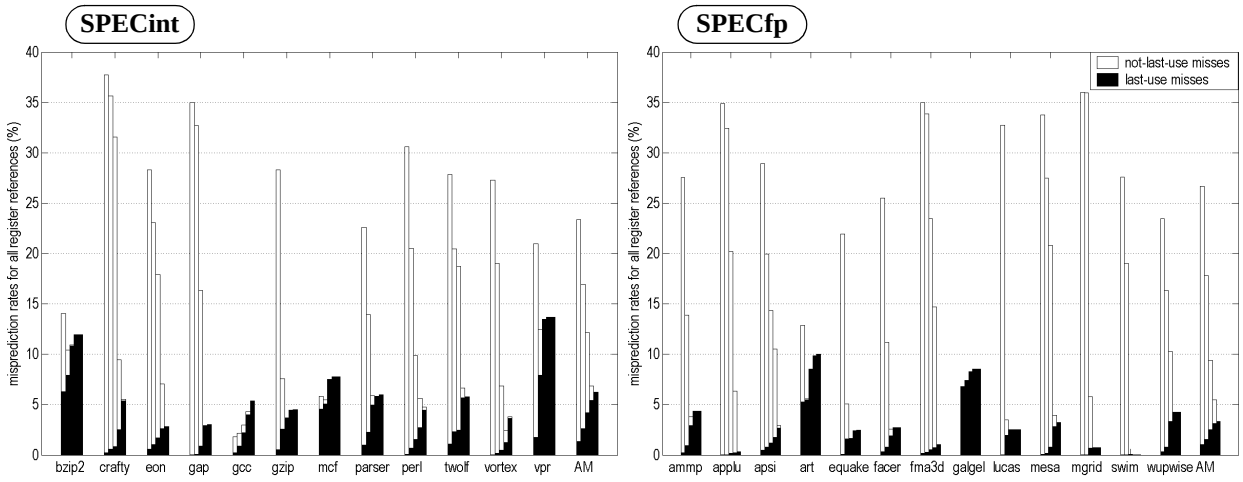

Figure 8. Misprediction rates for all register references. For each benchmark, the five bars represent the misprediction rates of 32, 128, 512, 2k and 8k-entry predictors. The solid (hollow) bars represent a register reference wrongly labelled as a "last-use register" ("not a last-use register").

Concerning "last-use register" mispredictions, they increase as the number of entries vary from 32 to 8k: from 1.4% to 6.2% in integer and from 1.1% to 3.3% in floating point, on average. This is because a large predictor tends to keep infrequent last-use patterns that become (correctly) replaced in small predictors. On the other hand, "not a last-use register" mispredictions are more frequent in small-sized predictors (for a 32-entry predictor, 22% and 26% for integer and floating point, respectively) and decrease noticeably with larger predictors, being negligible with the largest ones (for a 8k-entry predictor, 0.05% and 0.02% for integer and floating point, respectively).

Although the overall misprediction ratio is lower for the largest predictor, notice that the obtained performance may not be the best. This is because the gain obtained by correctly identifying more last-use registers must make up for the penalty of prematurely releasing more registers.

A suitable LUpred configuration depends on the recovery policy supporting SER-LUP and the corresponding misspeculation penalty. Such evaluation requires a detailed SER-LUP implementation beyond the scope of the paper.

3.3 Auxiliary Register File (XRF)

As stated in Section 3.2, a "last-use register" misprediction can cause a premature movement to XRF of a value required later on for some instruction. But this unexpected use (UU, from now on) is not the only one that can appear in the presence of control speculation and out-of-order execution. UUs may still appear even with a correct last-use prediction. Figure 9 shows examples of the two additional situations in which UU instructions can appear. In Figure 9a, a correctly predicted last-use instruction (*predLU*) issues out-of-order before a previous instruction (*U*) reads the register `r1`. The *U* instruction, when issued, finds released the value of `r1`, becoming an *out-of-order unexpected use* (oooUU). Figure 9b shows the case in which a branch misprediction directs the program execution towards a path using the released value. We call *mispredicted-path unexpected use* (mpUU) to this case. A final case of unexpected uses can be found inside exception handlers, but their impact on performance should be negligible.

The XRF handles all UUs by supplying the prematurely released values at a higher latency than the register file. Eventually, a value is released from XRF when the right redefinition commits.

In order to locate a value stored in XRF neither logical nor physical identifiers can be used. Figure 10 shows an example illustrating this fact, where it can be seen that the instruction *UU* has lost the link to `value1`, which has been moved to XRF. To solve this problem, we can allocate an unique virtual tag to each redefining instruction, as suggested by Monreal et al. in [12]. Virtual tags support instruction wakeup in the Issue Queue while physical register identifiers are still used to access the register file. By adding some additional mapping

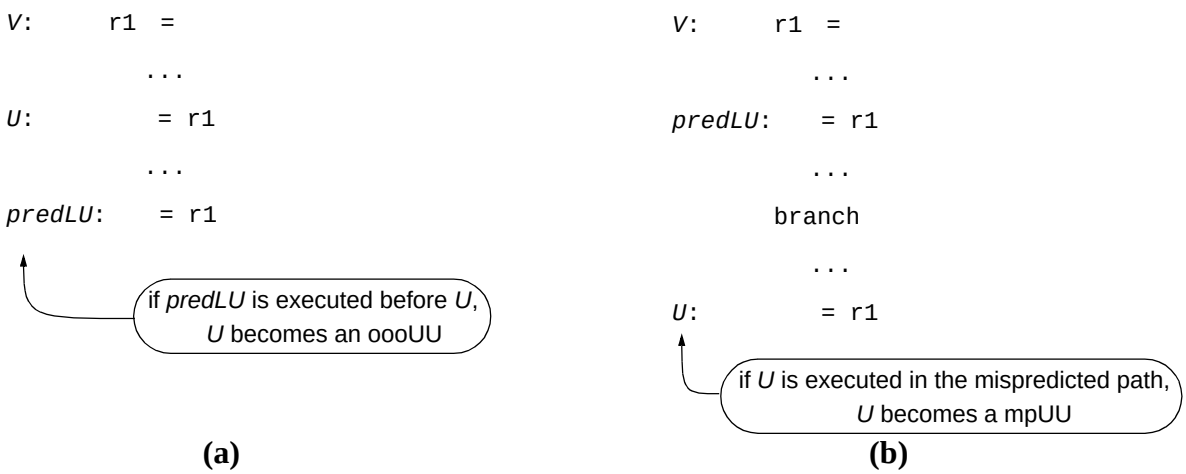

Figure 9. Examples of an out-of-order unexpected use (oooUU) (a) and a mispredicted path unexpected use (mpUU) (b).

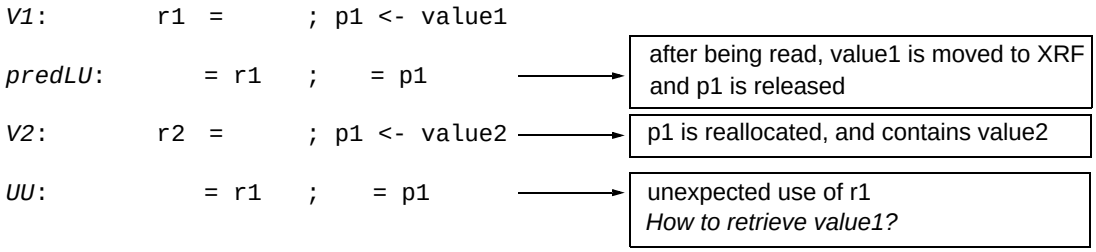

Figure 10. Instruction *UU* wants to read the logical register `r1`, but its current mapping (`p1`) does not contain the right value, which was saved in XRF.

tables it is possible finding the right XRF register identifier and using it to satisfy any unexpected use. But notice that from the moment a physical register experiences an early release until the moment it is overwritten, several cycles can pass. Therefore, it is an implementation option whether to take profit or not from this situation. Section 4.2 will elaborate more on this topic, showing the performance margin separating both options.

4. RESULTS

First, we describe the simulation environment and the used workload. Next we analyze the performance potentials of the SER-LUP policy based on a perfect predictor and an unbounded XRF with all the necessary number of entries and ports. Finally, a limited-size XRF is taken into account.

4.1 Simulation Environment and Workload

In all experiments, we have used a detailed cycle-based simulator derived from SimpleScalar v3.0 [2]. We have modified the SimpleScalar RUU-based management of resources by modelling an explicit use of separate register files (RFint, RFfp, XRFint and XRFfp), a Reorder Buffer and two Instruction Issue Queues (int+mem, fp). The main parameters of the simulated microarchitecture are given in Table 1.

All the benchmarks of SPEC2000 have been used as workload except for *sixtrack*, which could not be executed within our framework. We use the Alpha binaries compiled by C. Weaver (www.simplescalar.com) and simulate a contiguous run of 100M instructions from the points suggested by Sherwood et al. in [16].

In the following analysis, a processor with conventional renaming is compared to a processor enhanced with our SER-LUP proposal. We use a 64int+64fp register file, the design point considered in Section 2. In order to set performance bounds, we consider first an unbounded XRF along with and ideal last-use predictor. The chosen predictor is an oracle having perfect knowledge of the last instruction that references a register in program order. We also consider that XRF is the only provider of unexpected uses, the conservative option mentioned above. From now on, we will refer this configuration as SER-OB (SER-Oracle Based).

4.2 Sensitivity of performance to XRF latency

Figure 11 shows the average number of instructions committed per cycle (IPC) for conventional (conv) and SER-OB schemes working on a 64int+64fp register file. The leftmost bar in each group represents conventional renaming, and the following bars represent SER-OB with an unbounded XRF with unexpected use penalty of 0, 2 or 4 cycles.

It can be observed that, for both integer and floating point codes, the SER-OB mechanism significantly outperforms the conventional one. SER-OB backed by the fastest XRF gives an average speedup of 23% and 47% for integer and floating point, respectively. In particular, the improvements obtained in

Table 1. Processor parameters.

Parameter	Value	Parameter	Value
Fetch, decode and rename width	8 instructions, up to 2 taken branches fetched	L1 I-cache	64 KB, 4-way set-associative 32-byte line size, 1-cycle hit time
Branch prediction	hybrid predictor: bimodal + gshare 16-bits with speculative update	L1 D-cache	32 KB, 2-way set-associative 64-byte line size, 2-cycle hit time
Functional Units (latency)	8 simple int (*1*), 2 mult int (*7*), 4 load/store, 4 simple FP (*4*), 4 FP mult/div (*4/16*)	L2 U-Cache	256 KB, 8-way set-associative 128-byte line size, 10-cycle hit time
Reorder Buffer Issue window size	256 entries int+mem: 64 entries; fp: 32 entries	L3 U-Cache	4 MB, 4-way set-associative 128-byte line size, 16-cycle hit time
Load/Store Queue	128 entries with store-load forwarding	Main Memory	unbounded size, 200-cycle access time
Issue width	8int + 4fp	TLB	256 entries, 4-way set-associative
Issue mechanism	Out of order. Loads are executed when all previous store addresses are known	Physical registers	36-288 int / 36-288 fp (32 int / 32 fp logical)
Commit width	8 instructions	XRF	32-287 int /32-287 fp 0-2-4 cycles read penalty

swim and *applu* are impressive (211% and 85%, respectively). These were the floating point benchmarks with the fewest number of Free registers and the highest number of Allocated registers in the Non-Used state (see Figure 3).

Figure 11 also shows that for almost two thirds of the programs, IPC is nearly insensitive to the the XRF latency. Since a mpUU slows down the mispredicted path execution, it does not affect performance, so it is expectable that benchmarks affected by the increase in the XRF latency are mostly the ones that suffer from a large number of oooUUs. Whenever these unexpected uses are on any critical path, their negative effect on performance will be higher.

As stated previously, a given implementation of SER-OB can take profit of early released physical registers until the moment in which they are overwritten, avoiding some fraction of XRF accesses. Next we consider this option, calling it SER-OB+[1].

Figure 12 compares the percentage of XRF reads (number of values read from XRF divided by total number of values read) of SER-OB and SER-OB+ assuming a 4-cycle, unbounded XRF. From left to right, the two bars in each group represent SER-OB (all unexpected uses supplied by XRF), and SER-OB+. All reads directed to XRF correspond to unexpected uses, and each bar is split according to the contribution of mispredicted-path instructions (mpUU) and out-of-order execution (oooUU). In SER-OB, just as stated above, it can be verified that the benchmarks with higher number of oooUUs also suffer the largest performance reduction when increasing the XRF latency (see *gcc*, *parser*, *twolf*, *ammp* and *mesa* in Figure 11).

As Figure 12 shows, taking advantage of the unexpectedly referenced values still being in the register file removes most of oooUUs (from almost 2% to around 0.1%, for both integer and floating point benchmarks). For integer benchmarks, more than half of mpUUs are also suppressed with SER-OB+. This results in an average IPC improvement of 4.6% and 1.3% for integer and floating point programs, respectively. This enhancement is supposed to be more effective with a real last-use predictor because it will possibly hide last-use mispredictions.

Unless noted otherwise, from now on all metrics are computed with SER-OB+ working with a 4-cycle XRF.

4.3 Sensitivity of performance to XRF size

The upper limit on the number of entries needed for each XRF (int and fp) is determined by both the ROB size and the number of logical registers: ROBsize+NLR-1 (287 entries in our configuration). Figure 13 shows the average and maximum number of allocated entries for the unbounded XRF for SER-OB+. The average number of entries allocated in XRF is 73 for integer benchmarks, and 95 for floating point codes, far lower than the upper limit. These figures suggest that XRFs of reasonable sizes may sustain good performance levels, since few opportunities of early register release would be lost due to lack of XRF entries.

In order to analyze the performance sensitivity to the XRF size, Figure 14 shows IPC for conventional and the SER-OB+ scheme. The leftmost bar in each group represents conventional renaming and the following bars represent SER-OB+ scheme with XRFs of 32, 64, 128 and 287 entries. All these figures apply to a 64int+64fp register file.

Once again, it can be observed that any SER-OB+ configuration significantly outperforms the conventional one for both integer and floating point codes. Comparing with Figure 11 we notice that SER-OB working with the biggest, zero-cycle XRF performs the same than SER-OB+ working with the biggest 4-cycle XRF. Figure 14 also shows that for most integer benchmarks IPC does not increase noticeably with the XRF size (*crafty*, *vortex*, *twolf* and *vpr* are exceptions). Nevertheless, the opposite occurs for floating point benchmarks where all but two programs (*equake* and *lucas*) take advantage of larger XRFs. These results suggest a design with XRFs of different size for the integer and floating point data paths.

[1] This is also the perfect SER scheme used in Figure 3.

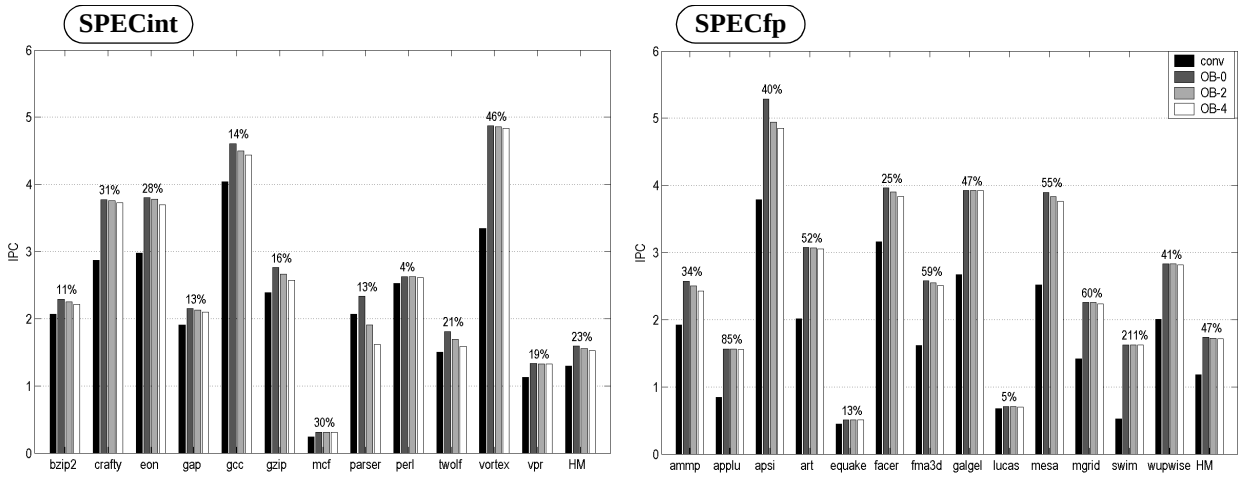

Figure 11. IPC for a 64int+64fp register file. The rightmost bar group of each plot is the harmonic mean (HM). Percentage figures report speedups between conventional and SER-OB releasing with the fastest XRF.

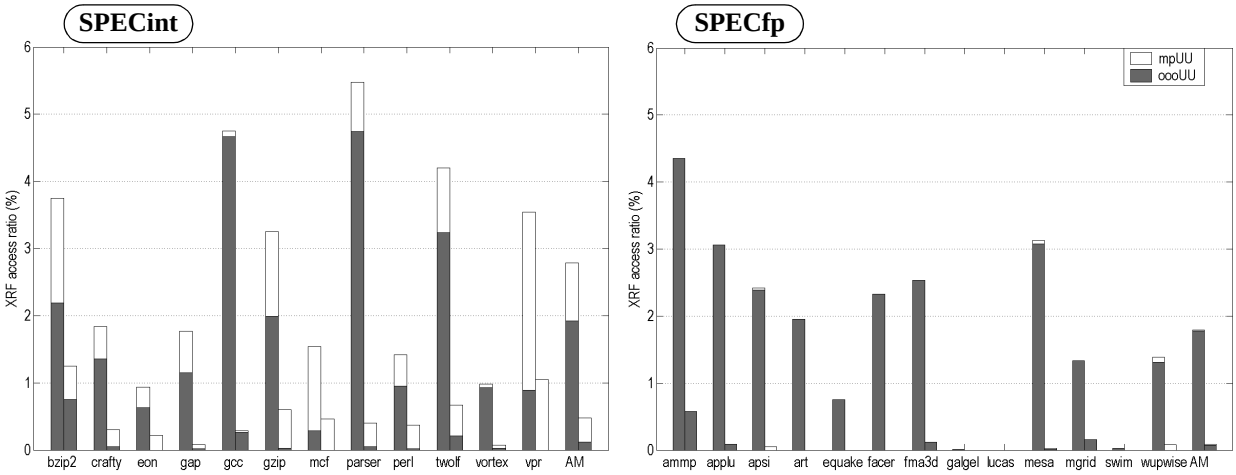

Figure 12. Percentage of XRF reads for SER-OB releasing and a 64int+64fp register file. For each benchmark, the two bars represent the access ratio for SER-OB and SER-OB+. The rightmost bar group is the arithmetic mean (AM).

4.4 Reducing the register file size

It is also interesting to analyze the effect of varying the register file size. Figure 15 illustrates how IPC varies according to the number of physical registers for five sample configurations (conventional, SER-OB+ with XRF of 32, 64, 128 and 287 entries). We can see how SER-OB+ makes the processor performance less dependent on the register file size. For integer codes and XRF of 287 entries, IPC is nearly insensitive to the register file size. The smaller simulated size of 36 physical registers, reaches nearly 90% of the IPC obtained with the largest register file. For floating point codes, the maximum IPC is reached from 96-entry register files when XRFs of 128 or more entries are used. Combining the biggest XRF with register files of 48 and 64 entries gives 90% and 97% IPC, respectively.

If the processor cycle time is constrained by the register file access time, a reduction in the number of registers may lead to an increase of the clock frequency, and an added performance improvement. Figure 16 shows billions of instructions per second (BIPS[2]) executed by processors with conventional renaming and SER-OB+ with XRFs of different sizes. For integer benchmarks, the best performance for conventional renaming is obtained with 96 registers. But when SER-OB+ is applied with a XRF of 32 entries, this number is reduced to 64. With XRFs of 64, 128 and 287 entries, the best figures correspond to a register file of only 40 entries. For floating point programs, the best performance for conventional renaming is obtained with 128 registers. When adding SER-OB+ with a XRF of 32, this number is reduced to 96. With a XRF of 64, best performance is obtained with 80; and with XRFs of 128 and 287, the best figures correspond to a register file with 64 entries.

From these figures, we can conclude that SER-OB+ achieves both a great reduction in the register file size and a performance increase. For instance, considering both integer and floating point code, a processor with a conventionally

[2] Cycle times have been obtained according to the Rixner et al. model [15].

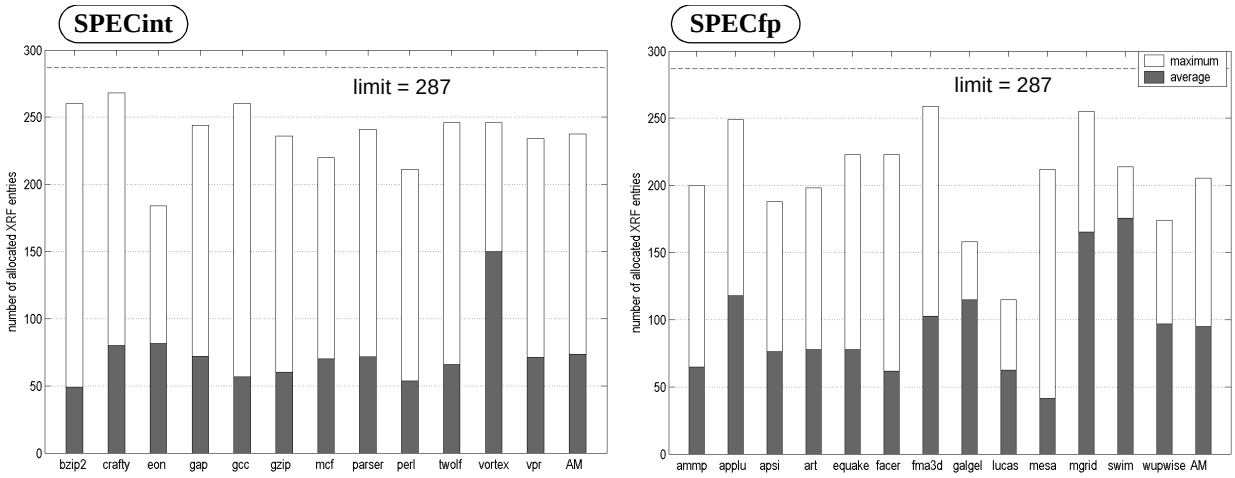

Figure 13. Average and maximum number of allocated XRF entries (64int+64fp register file). The rightmost bar is the arithmetic mean (AM).

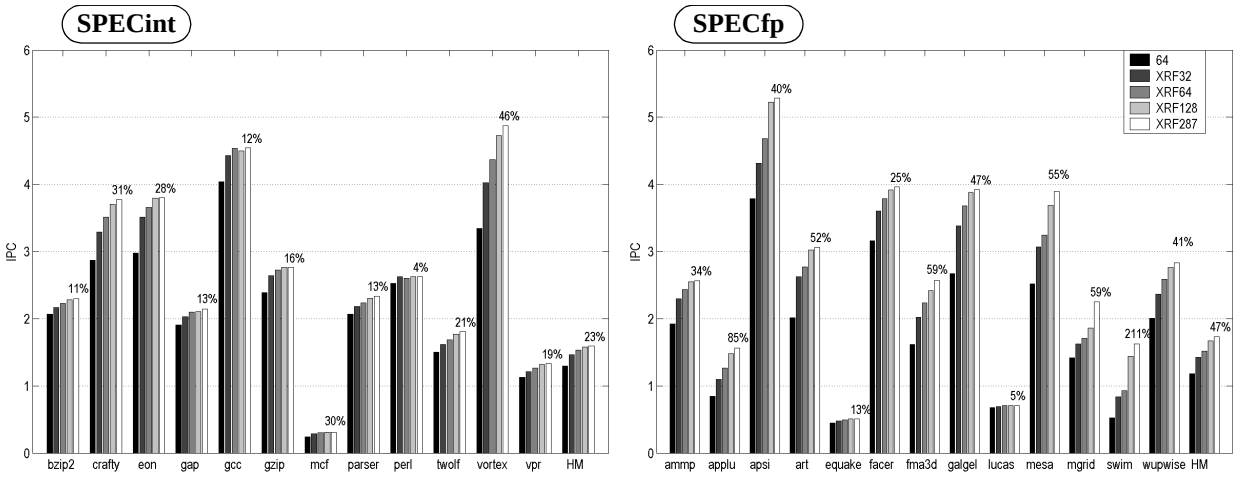

Figure 14. IPC for a 64int+64fp register file. The rightmost bar group is the harmonic mean (HM). Percentage figures report speedups between conventional and SER-OB+ with the biggest XRF.

managed 96int+96fp register file could be replaced for equal IPC with a 64int+64fp register file managed with SER-OB+ and backed with a 64int+64fp XRF, this achieving a 12% IPS increase.

5. RELATED WORK

Farkas et al. were also interested in setting performance bounds on the mechanism that controls the release of physical registers [6]. Nevertheless, they focused on an imprecise exception model where early released values are not saved in any auxiliary store. The release policy they test can not act until the redefinition is executed, delaying register release when the redefinition is far from the last-use. Moreover, to be able to recover from branch mispredictions a register can not be released while its redefining instruction is speculative. These conservative conditions delay register release and keeps their performance bound far from the SER-OB+ limit.

The *use-based* register caching proposed by Butts and Sohi pursues a goal similar to ours' [4]. They suggest a small register cache holding the physical registers with more predicted pending references. Insertion and replacement is directed by a *degree of use* predictor which is very accurate for low degree of uses [3]. In contrast to SER-LUP they need to swap contents and mappings between the cache and the main register file in the event of a cache miss. Moreover, the main register file is managed in a conventional way, without any early release policy.

In the context of the register management several mechanisms have been proposed to release physical registers early and eagerly [1][5][9][11][14]. Jones et al., proposed a technique based on a compiler analysis to release a physical register as soon as it has been read by its only consumer [9]. To do that, several logical registers are reserved to hold the values that the compiler has been able to identify as degree of use one. The drawback of this technique is the limited knowledge of the run-time behaviour that prevents the identification of all values having only one consumer instruction. Moudgill et al., proposed to release registers as soon as last-use instructions complete out of order [14]. Their last use tracking is based on counters which record the number of pending reads for every physical register. This proposal does not support precise

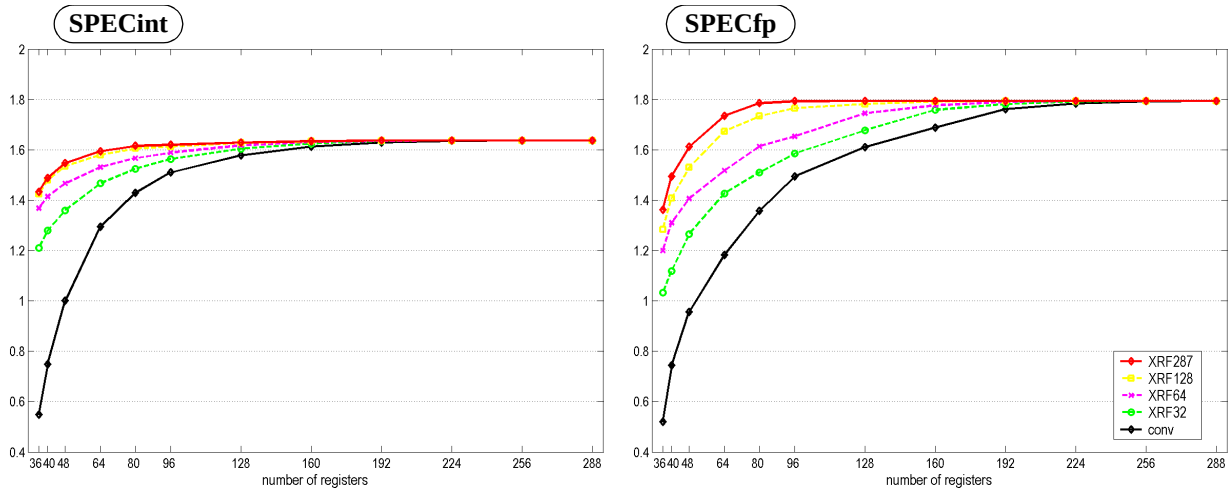

Figure 15. IPC harmonic mean *vs.* number of physical registers for conventional and SER-OB+ working with XRFs of 32, 64, 128 and 287 entries.

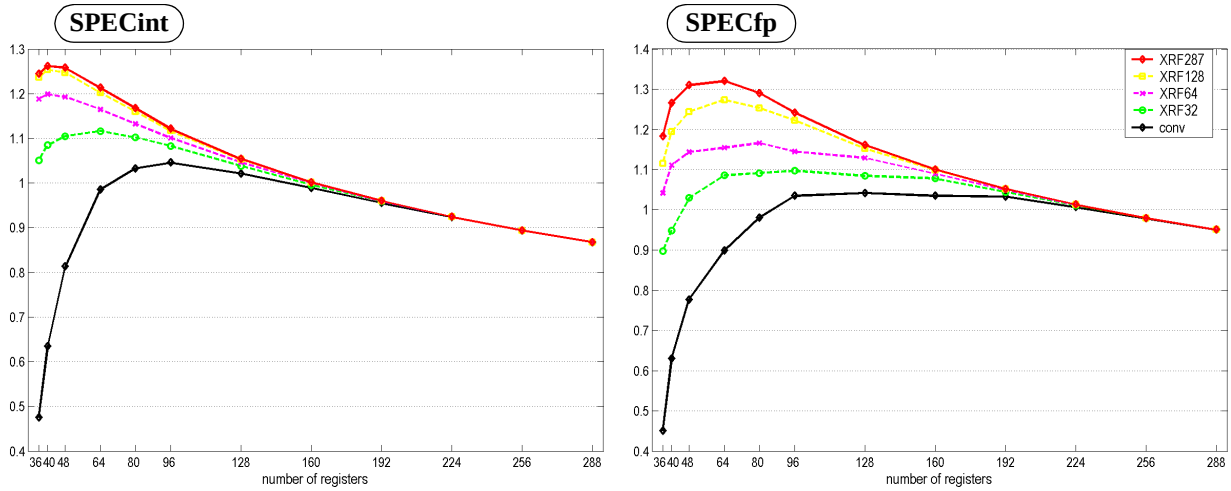

Figure 16. Billions of instructions per second executed (BIPS harmonic mean) *vs.* number of physical registers for conventional and SER OB+ working with XRF of 32, 64, 128 and 287 entries.

exceptions since counters are not correctly recovered when instructions are squashed. In Cherry, the release of any physical register also requires the instruction that redefines its logical register becoming non-speculative [11]. The support of precise exception recovery in this mechanism is achieved by a periodical checkpoint of all the values held in the register file.

Balasubramonian et al. proposed a precise exception model for releasing registers based on the movement of inactive registers to the second level (L2) of a two-level register file [1]. A value is reinstalled back to the level of active registers (the first level, L1) when redefinitions arising from a mispredicted path are squashed. Ergin et al. implements a mechanism similar to the proposed by Moudgill et al. where a precise exception support is added by means of the management of a *shadow* register file where released values are kept [5]. Early release of short-lived values is performed in the commit stage of their generating instructions, whenever all consumers have executed and the corresponding redefining instructions have been renamed. A complementary technique is responsible for the release of physical registers whose redefining instructions are decoded after the commit of their value producers. The number of early released values kept in the shadow file is limited by the physical register file size, so small register files lose a lot of opportunities of recycling their entries.

All previous hardware schemes are not able to release a physical register until the next redefinition of the logical register it is mapped to is renamed [1][5]. Some of them require in addition that the redefinition becomes non-speculative [11][14]. This hard condition delays the early release of a register in case its redefining instruction appears many cycles later (according to our experiments, an average of 12 cycles -64int+64fp register file-). On the other hand, the array of counters used in all these works to track the pending consumers of a physical register must be restored after every mispredicted branch, this involving the need of a complex hardware. Besides, when a redefining instruction is squashed because of a mispredicted branch, the incorrectly superseded register revives, so in [1] and [5] it is proposed the return of the speculatively released values to their previous physical registers, even if they will not be referenced again. Acting in this form, the high pressure in the register file is still maintained.

Opposite to this kind of solutions, our policy does not wait until the rename of a redefining instruction to release the physical register allocated to the previous version. A last-use predictor replaces the complex hardware that other proposals need to verify the release conditions. Moreover, the values released by mispredicted path instructions do not return to the physical register file. Instead of this, these values continue feeding unexpected uses during the cycles they reside in the physical register file before being overwritten, this avoiding most XRF reads.

As this paper presents an idealized scheme of SER-LUP, we believe it would be unfair to compare it with the real implementations referenced in [1] and [5]. A quantitative comparison should include not only performance figures, but also complexity, cost and cycle time estimates. Our future work on real SER-LUP implementations will address such detailed comparisons.

6. CONCLUSIONS

In this paper we have analyzed the performance potentials of a novel and more aggressive *speculative early release* policy based on last-use prediction (SER-LUP). Under this policy the usefulness of a register value is determined by means of the prediction of its last consumer, and early released values are recovered either from the register file without paying any penalty, or from a separate *auxiliary register file* (XRF) if the released physical register has been overwritten. As a result, this new policy will permit optimized designs for the register file reaching at the same time a performance increase and a size reduction.

Following the same idea presented in [13], SER-LUP shifts the release responsibility from redefining instructions to predicted last-use instructions. But, in contrast with previous proposals, it can release a physical register even when the register redefinition has not been fetched, achieving good register recycling when several register redefinitions are stalled prior to be renamed. The idea of using a last-use predictor also replaces the hard checking of conditions proposed by other authors. Besides, the introduction of virtual tags to keep track of instruction dependences also enables the location in XRF of the released values.

Apart from showing the SER-LUP potentials, in this paper we have also analyzed the viability of a real *pattern of use* predictor by demonstrating that last-use instructions can be accurately predicted. All of these results encourage the study of a real design where last-use registers are predicted by a real predictor, and the released values are backed up with a low-ported XRF.

7. ACKNOWLEDGMENTS

This work was supported in part by Diputación General de Aragón grant "Grupo Consolidado de Investigación" (BOA 20/04/2005), Spanish Ministry of Education and Science grant TIN2004-07739-C02-01/02, and European Union Network of Excellence HiPEAC (High-Performance Embedded Architectures and Compilers, FP6-IST-004408).

8. REFERENCES

[1] R. Balasubramonian, S. Dwarkadas and D.H. Albonesi, “Reducing the Complexity of the Register File in Dynamic Superscalar Processors”. Proc. 34th Int’l Symp. Microarchitecture (MICRO 01), Dec. 2001, pp. 237-249.

[2] D. Burger, and T.M. Austin, The Simplescalar Tool Set v2.0, Technical Report 1342, Computer Science Dept., University of Wisconsin-Madison, June 1997.

[3] J.A. Butts and G. Sohi, “Characterizing and Predicting Value Degree of Use”, Proc. 35th Int’l Symp. Microarchitecture (MICRO 02), Nov. 2002, pp. 15-26.

[4] J.A. Butts and G. Sohi, “Use-Based Register Caching with Decoupled Indexing”, Proc. 31st Int’l Symp. Computer Architecture (ISCA 04), June 2004.

[5] O. Ergin, D. Balkan, D. Ponomarev and K. Ghose, “Increasing Processor Performance Through Early Register Release”. Proc. 22nd Int’l Conf. on Computer Design (ICCD 04), Oct. 2004, pp. 480-487.

[6] K.I. Farkas, N.P. Jouppi and P. Chow, “Register File Considerations in Dynamically Scheduled Processors”, Proc. 2nd Int’l Symp. High-Performance Computer Architecture (HPCA 96), Feb. 1996, pp. 40-51.

[7] L. Gwennap, “MIPS R12000 to Hit 300 MHz,” Microprocessor Report, Micro Design Resources, vol. 11, no. 13, Oct. 1997, pp. 1-4.

[8] 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, Feb. 2001.

[9] T.M. Jones, M.F.P. O’Boyle, J. Abella, A. González, and O. Ergin, “Compiler Directed Early Register Release”, Proc. 14th Int’l Conf. on Parallel Architectures and Compilation Techniques (PACT 05), Sept. 2005, pp. 110-122.

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

[11] J. Martinez, J. Renau, M. Huang, M. Prvulovich, J. Torrellas, "Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors", Proc. 35nd Int’l Symp. Microarchitecture (MICRO 02), Nov. 2002, pp. 3-14.

[12] T. Monreal, A. González, M. Valero, J. González and V. Viñals, "Delaying Physical Register Allocation Through Virtual-Physical Registers", Proc. 32nd Int’l Symp. Microarchitecture (MICRO 99), Nov. 1999, pp.186-192.

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

[14] M. Moudgill, K. Pingali and S. Vassiliadis, “Register Renaming and Dynamic Speculation: an Alternative Approach”, Proc. 26th Int’l Symp. Microarchitecture (MICRO 93), Nov. 1993, pp. 202-213.

[15] S. Rixner, W. J. Dally, B. Khailani, P. Mattson, U. J. Kapasi and J.D. Owens, “Register Organization for Media Processing”, in Proceedings of the 6th Int’l Symposium on High-Performance Computer Architecture (HPCA 00), January 2000, pp. 375-386.

[16] T. Sherwood, E. Perelman, G. Hamerly and B. Calder, “Automatically Characterizing Large Scale Program Behavior,” Proc. 10th Int’l Conf. Architectural Support for Programing Languages and Operating Systems (ASPLOS 02), Oct. 2002, pp. 45-57.

[17] K.C. Yeager, “The MIPS R10000 Superscalar Microprocessor”, IEEE Micro, vol. 16, no. 2, Apr. 1996, pp. 28-40.