A Novel Register Renaming Technique for Out-of-Order Processors

Hamid Tabani, Jose-Maria Arnau, Jordi Tubella, Antonio González
Department of Computer Architecture, Universitat Politècnica de Catalunya
Barcelona, Spain
$\{htabani, jarnau, jordit, antonio\}$@ac.upc.edu

***Abstract*—Modern superscalar processors support a large number of in-flight instructions, which requires sizeable register files. Conventional register renaming techniques allocate a new storage location, i.e. physical register, for every instruction whose destination is a logical register in order to remove false dependences. Physical registers are released in a conservative manner when the same logical register is redefined. For this reason, many cycles may happen between the last read and the release of a physical register, leading to suboptimal utilization of the register file.**

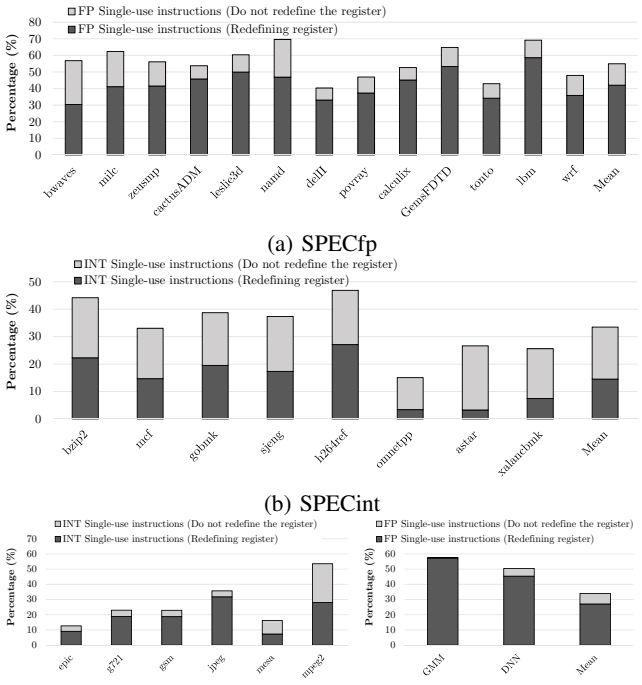
We have observed that for more than 50% of the instructions in SPECfp and more than 30% of the instructions in SPECint that have a destination register, the produced value has only a single consumer. In this case, the RAW dependence guarantees that the producer-consumer instructions pair will be executed in program order and, hence, the same physical register can be used to store the value produced by both instructions.

In this paper, we propose a renaming technique that exploits this property to reduce the pressure on the register file. Our technique leverages physical register sharing by introducing minor changes in the register map table and the issue queue. We also describe how our renaming scheme supports precise exceptions. We evaluated our renaming technique on top of a modern out-of-order processor. Our experimental results show that it provides 6% speedup on average for the SPEC2006 benchmarks. Alternatively, our renaming scheme achieves the same performance while reducing the number of physical registers by 10.5%.

***Keywords*-Register Renaming; Register File; Precise Exceptions;**

I. Introduction

Dynamically-scheduled superscalar processors exploit instruction-level parallelism (ILP) by reordering and overlapping the execution of instructions in an instruction window. The number of instructions that can be executed in parallel is highly dependent on the instruction window size and, thus, wide issue processors require a large instruction window [1]. However, a large instruction window has some implications in other critical parts of the microarchitecture, such as the size of the physical register file [2]. In this work, we are concerned with this issue. On the other hand, in spite of being able to execute instructions out-of-order, the amount of ILP that current superscalar processors can exploit is significantly restricted by data dependences. Due to the limited number of architectural registers, at some point compilers start reusing them, which may cause name dependences (write-after-read and write-after-write dependences). Dynamic renaming schemes eliminate these name dependences by assigning a new storage location to the destination register of each instruction. This increases the amount of independent instructions that can be executed in parallel, which results in an increase in the ILP.



(a) SPECfp

(b) SPECint

(c) Mediabench and Cognitive applications

Figure 1: Percentage of instructions with a destination register that are the only consumers of the value of a register. We distinguish between instructions that redefine the single-use register and instructions that redefine a different logical register.

Larger instruction windows require a higher number of physical registers. However, increasing the size of the register file is challenging and has important implications in terms of energy consumption, access time and area [2].

This work is motivated by the observation that, in a

2378-203X/18/$31.00 ©2018 IEEE
DOI 10.1109/HPCA.2018.00031

significant percentage of instructions with a destination register, this register has a single consumer. Figure 1 shows that more than 50% of the instructions in SPECfp and more than 30% in SPECint exhibit this property. In this case, the RAW dependence between producer and consumer will force sequential execution of the two instructions. In addition, since there is only a single consumer then no other instruction will require the value produced by the first instruction. Therefore, producer and consumer can safely use the same physical register as their destination.

In this paper, we propose a register renaming scheme that implements this reuse of registers in dynamically scheduled processors that implement precise exceptions. We show that identifying single-use registers can be accomplished with simple hardware. To this end, the register map table is extended with one bit per entry indicating whether this register has seen at least one consumer. An instruction that finds a source operand with this bit clear is the first consumer of the value and, hence, there are no older consumers. To detect the absence of younger consumers, two possible cases arise. The simplest case happens when the consumer is also the redefining instruction of the single-use register, in which case it is guaranteed that there are no younger consumers. Note that this is the case for a significant percentage of the instructions, as shown in Figure 1. In case the consumer is not the redefining instruction, a single-use predictor is employed to speculatively reuse the physical register. We show in Section VI that a simple predictor is able to achieve very high accuracy for SPEC benchmarks.

To be able to recover the state of the processor in a branch misprediction, interrupt or exception, we propose to use a multi-bank register file with check-pointed register banks using shadow cells [3]. Check-pointed registers are allocated whenever it is predicted that a register might be reused. Hence, the former value of a register can be recovered in an event of branch misprediction, interrupt or exception. The proposed register renaming scheme does not require any ISA changes nor compiler support.

In short, the main contributions of this paper are the following:

- We present an analysis of benchmarks that shows a high opportunity to reuse physical registers, by exploiting the large percentage of single-use values.
- We propose a novel register renaming technique that reduces the pressure on the register file by enabling physical register sharing.
- We use a cost-effective register file design with check-pointed and conventional registers to recover the state of the processor in event of branch mispredictions, interrupts or exceptions.
- We show that for SPEC benchmarks, the proposed register renaming scheme results in 6% speedup for a given register file size, or 10.5% reduction in register file size for the same performance.

The remainder of the paper is organized as follows. Section II reviews the traditional register renaming techniques. Section III presents the motivation for this work. Section IV describes our register renaming scheme. Section V presents our evaluation methodology, and the experimental results are provided in Section VI. Section VII reviews some related works. Finally, Section VIII summarizes the main conclusions of this work.

II. Register Renaming

Register renaming is key for the performance of out-of-order processors. Instructions, after being decoded, are kept in the reorder buffer until they commit. The size of the reorder buffer determines the maximum number of in-flight instructions. These instructions are usually called instruction window.

The goal of renaming is to remove register name dependences, write-after-read and write-after-write dependences for the instructions in the instruction window. This is achieved by allocating a free storage location for the destination register of every new decoded instruction. The most common solution to provide the storage locations is a merged register file [4]. In this case, there is a physical register file that contains more registers than those defined in the ISA, which are referred to as logical registers. A register map table is used to manage the translations from logical to physical identifiers. When an instruction commits, the physical register allocated by the previous instruction with the same logical destination register is freed. This has become the adopted approach of practically all current microprocessors, due to its energy efficiency, and is the baseline technique assumed in this work. Other schemes such as renaming through the reorder buffer [5] are much less commonly used nowadays, since they tend to be less energy efficient.

In the merged register file organization, a number of physical registers close to the number of logical registers plus the window size is required since the majority of the instructions have a destination register. A number of physical registers equal to the number of logical registers is needed to keep the committed state of the processor. In addition, for every instruction whose destination operand is a register, an additional register is allocated when it enters the window at rename stage and a physical register is released when it leaves the window at commit stage.

Renaming schemes are conservative to guarantee correct execution. In conventional schemes, a physical register is released when the instructions that redefines the corresponding logical register commits. In this manner, it is guaranteed that there is no other potential consumer of this value stored in the physical register being released, since the redefining instruction is no longer speculative. Note that many cycles may happen between the last read of the register and its release, which leads to underutilization of the register file.

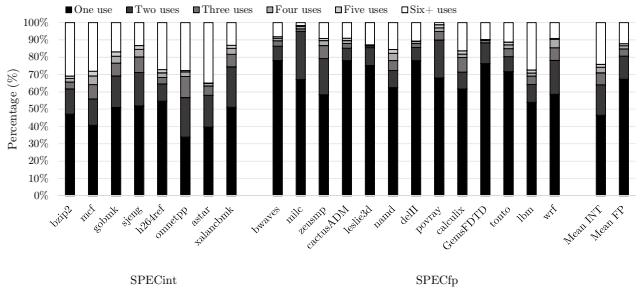


Figure 2: Percentage of registers consumed one, two, three, four, five and six or more times. Most of the values are consumed just once in SPEC.

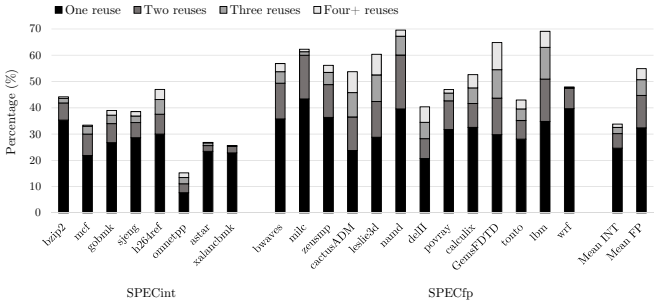


Figure 3: Percentage of instructions that can reuse a physical register, if a register can be reused up to one, two, three or an unlimited number of times. Note that we only consider instructions with a destination register.

This results in an unnecessary increase of the register file pressure.

III. Motivation

This work is primarily motivated by the observation that, in a significant percentage of instructions, the value stored in a register has only a single consumer (see Figure 2). In conventional register renaming schemes, the single-consumer instruction allocates a new physical register for the destination register. However, in this case, once the value of the source register is read by the consuming instruction, the same physical location can be used to write the result since no more consumers need the previous content of the register. In other words, the source register can be reused for the destination instead of allocating a new one. As Figure 1 shows, for SPEC2006 benchmarks, this happens for more than 50% and 30% of SPECfp and SPECint instructions respectively. Note that, on average, more than 85% of the instructions require a physical register as a destination. Furthermore, we often find chains of instructions where a given logical register is both the destination operand and a single consumer operand of it. This is the case of instructions *I1, I4, I5* and *I6* in Figure 4. In this case, all the instructions in the chain can share the same physical register as destination, further reducing the pressure on the register file.

Following this idea of physical register sharing for the instructions in a chain, Figure 3 shows the percentage of instructions that can avoid allocating a new physical register if each register can be reused up to one, two, three or an unlimited number of times. Note that Figure 3 only considers instructions with a destination register, i.e. instructions that require an allocation of a physical register at renaming. For this reason, the percentage of *One Reuse* in Figure 3 is not equal to the category *One use* in Figure 2, as it does not include single-uses performed by stores, comparisons and other instructions without a destination register. However, since the majority of instructions include a destination register, still more than 50% of the instructions in SPECfp and more than 30% of the instructions in SPECint can reuse a physical register multiple times. More specifically, 32.3%, 12.3% and 5.9% of the instructions in SPECfp can reuse a physical register up to one, two and three times respectively. Two reuses means a chain of three instructions whose only consumer is the next instruction in the chain. Similarly, three reuses means a chain of four instructions sharing the same physical register as destination. On the other hand, only 4.1% of the instructions can reuse a physical register more than three times, i.e. chains of more than four instructions are unusual. Regarding SPECint benchmarks, 22%, 5.2% and 2.3% of the instructions can reuse a physical register up to one, two and three times respectively, while only 1.2% of the instructions can reuse a physical register more than three times.

IV. Renaming with Physical Register Reuse

In this section, we present a novel register renaming scheme for out-of-order processors that exploits physical register sharing to reduce the pressure on the register file. First, we illustrate the technique through an example. Second, we provide the implementation details of the technique, describing the changes to the different hardware structures of the processor, such as the register map table or the issue queue. Finally, we extend our renaming scheme to support precise exceptions, describing the required support in the register file.

A. Proposed Register Renaming Technique

For the sake of clarity, we first explain the proposed register renaming scheme through an example. Figure 4 presents the assembly code for several instructions of an application. To execute the eight instructions in this example, conventional renaming schemes allocate eight different physical registers, one per instruction, as illustrated in Figure 4(a). The outcome of the register renaming and the step-by-step updates to the register map table are also shown in Figure 4(a). As we can see in this example, four different physical registers are employed for the same logical register, *r1*.

In this example, instructions *I1, I4, I5* and *I6* form a chain of read-after-write dependences that guarantees that they will be executed in program order and, therefore,

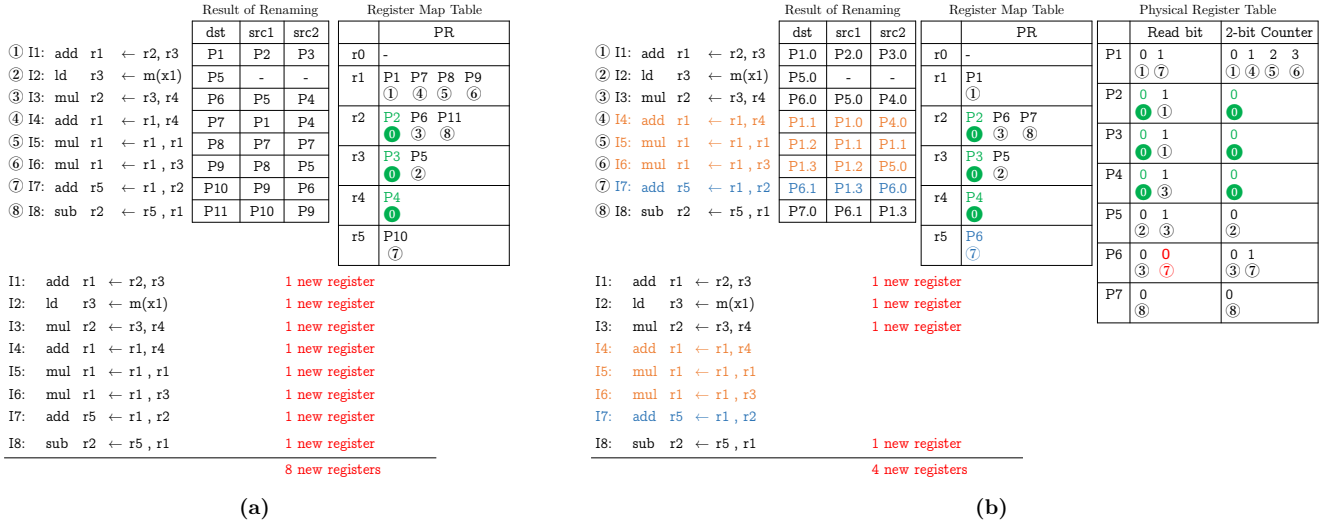


Figure 4: Step-by-step renaming of several instructions, including the changes in the Register Map Table and the Physical Register Table. In this example, we assume that physical registers *P2, P3* and *P4* have been previously assigned to *r2, r3* and *r4* respectively. (a) conventional renaming scheme, (b) the proposed renaming scheme.

they will write their results in order in the register file. In addition, each instruction is the only consumer of the previous one. In this case, the same physical register can be used as the destination for these four instructions, since the RAW dependence guarantees that an instruction produces its value before the next instruction in the chain is issued and, moreover, the single-use condition guarantees that the value is not used by any other instruction. In other words, there is no other instruction reading the value between the producer and the consumer in this chain of instructions that may introduce a WAR hazard. Hence, there is no requirement of keeping the four values alive in different physical registers for correct program execution.

In this paper, we propose to reuse the same physical register in the aforementioned condition, i.e. when two instructions exhibit a RAW dependence and the second instruction is the only consumer of the value. To leverage physical register sharing, we introduce a new hardware structure: the Physical Register Table (PRT). The PRT contains one entry per physical register, as shown in Figure 4(b). Each PRT entry includes one *Read bit* and a *2-bit Counter*. The *Read bit* is used to identify the first consumer of a register. If the *Read bit* is set it indicates that the physical register is the source operand for an in-flight or a committed instruction in the pipeline. On the contrary, if the *Read bit* is clear, it indicates that no consumer of the value has been found (i.e. fetched and renamed) yet.

When the first consumer of a register is being renamed, in order to reuse the source register for the destination register, we also have to verify that there will be no future consumers. In case the consumer is also redefining the first-use register, it is guaranteed that there will be no younger consumer of the value. For example, instruction *I5* in Figure 4 satisfies this property, as it is the only consumer of *r1* and it also redefines *r1*. In case the instruction being renamed is not the redefining instruction (see instruction *I8* in Figure 4), a simple single-use predictor is employed to decide whether the same register is reused or a new physical register is allocated. Section IV-D provides more details about the single-use predictor and the actions taken in case of misprediction.

On the other hand, the *2-bit Counter* keeps track of the number of instructions sharing the same physical register, and it is used to maintain the true dependences. Due to the sharing of registers, the same name, i.e. the same physical register ID, is used to identify different values produced by different instructions. Therefore, it is not possible to correctly identify which instructions have to be woken up in the issue stage when a value is produced using only this ID. To avoid this ambiguity, we append the *2-bit Counter* to the register ID, so the source or destination of an instruction is specified as the N-bits of the physical register ID plus the two bits of the counter. The *2-bit Counter* is increased each time the same physical register is reused and it identifies the different versions of this register. In this manner, up to four instructions can share the same physical register but yet RAW dependences can be identified, as different instructions produce or wait for different versions of the register. As shown in Figure 4(b), both instructions *I5* and *I6* take P1 as source operand, but they wait for version one *(P1.1)* and version two *(P1.2)* respectively. When instruction *I4* produces *P1.1* the issue logic only wakes up *I5*, that is the instruction waiting for version one.

This scheme can be generalized to employ an *N-bit counter* to allow up to 2^N instructions to share the same physical register as destination. Note that when the counter is saturated we cannot longer reuse the register, as it would not be possible to differentiate its multiple versions to keep track of the RAW dependences. We found that in SPEC

benchmarks it is uncommon to have more than three reuses (see Figure 3). Furthermore, additional bits represent larger overheads in the PRT and the issue queue. We found that a 2-bit counter provides a good trade-off between the degree of physical register sharing and cost.

Figure 4(b) shows the proposed renaming technique step-by-step. In this case, instructions *I1, I4, I5* and *I6* share the same physical register *P1*. Our renaming scheme only requires four physical registers, instead of the eight employed by the conventional approach. Next sections provide further details on the renaming technique, describing how the processor operates and the changes required to the different hardware structures.

1) Renaming Source registers: Every time a source operand of an instruction is renamed, the Register Map Table is accessed as in the conventional approach to get the physical register ID that is assigned to the logical register. Next, the physical register ID is used to index the PRT. The *Read bit* of the corresponding entry is set to indicate that an in-flight instruction will read the value stored in the register. In addition, the *2-bit counter*, that indicates the most recent version of the register, is read from the PRT, so the renaming logic provides the physical register ID plus the *2-bit counter*.

2) Renaming Destination Registers: Our renaming scheme tries to reuse some of the source registers as the destination for the instruction being renamed, in order to avoid an allocation of a new physical register. For this purpose, the renaming logic checks first the *Read bit* of the source registers in the PRT. If this bit is zero for some of the sources, it means the instruction being renamed is the first consumer of the value stored in that register. To identify single-use condition, the renaming logic also checks whether the instruction is the last consumer of the value. To this end, the source register ID is compared with the destination register ID of the instruction. If the source register matches the destination, it is guaranteed that the instruction is the last consumer of the value. On the contrary, the single-use predictor, described in Section IV-D, is triggered to decide whether the instruction is the only consumer of the value.

On the other hand, the *2-bit counter* of the source register is also accessed to verify that it is not saturated, i.e. that there are versions of the register available and, hence, the processor will be able to maintain the RAW dependences for another reuse. If the instruction is identified as the single consumer of the source register and the *2-bit counter* is not saturated, the source physical register is reused as the destination of the current instruction being renamed, and no allocation of a physical register is performed. The corresponding entry in the Register Map Table is updated to map the logical register to the physical register being reused. In addition, the *Read bit* is set to zero and the *2-bit counter* is increased in the corresponding PRT entry.

In case the instruction cannot be identified as the single consumer of a source register or the *2-bit counter* is saturated, a new physical register is allocated. The Register Map Table entry is updated with the ID of the allocated register, whereas the *Read bit* and the *2-bit counter* are set to zero in the PRT.

3) Releasing a Physical Register: When a physical register is being reused, no register allocation is required for the new instruction. This technique is equivalent to a release-on-rename scheme. Although no modification is done to the list of free registers, the result of the technique is identical to releasing the physical register and immediately allocating it to the new instruction.

In case the physical register cannot be reused, a new one is allocated. The old register is released when the redefining instruction commits. Therefore, if a physical register can be reused the technique mimics the behavior of a release-on-rename scheme, otherwise it works as the conventional release-on-commit approach.

4) Lack of Physical Registers: Conventional renaming schemes stall when the list of free registers is empty. In our approach, the renaming will be blocked only when there is no available physical register and there is no possibility of reusing a register as described in Section IV-A2. Our experimental results presented in Section VI show that our scheme is very effective avoiding stalls in the renaming stage due to lack of registers.

B. Mispredictions, Interrupts and Exceptions

Modern out-of-order processors use dynamic speculation to issue an instruction before it is known whether or not the instruction should be executed. In our renaming scheme, multiple instructions may reuse the same physical register, and each instruction in a chain of reuses overwrites the value produced by the previous instruction. Since values in a shared physical register are speculatively overwritten, it is necessary to recover the previous value of a register in case of a branch misprediction or an exception between two instructions in the chain of reuses.

In the example of Figure 4, assume that instruction *I2* causes a TLB miss or raises a page fault exception and, when the exception is triggered, instruction *I4* has already written its result in P1. In this case, the previous value of P1, i.e. the value produced by instruction *I1*, must be recovered before invoking the exception handler to maintain precise exceptions.

In order to deal with branch mispredictions and support precise exceptions, the different versions of a shared register must be kept. However, this requires extra storage and increases the pressure on the register file, which is exactly what our renaming technique is trying to avoid by using physical register sharing. Shadow bit cells are a cost-effective solution to keep previous values of a register in a check-pointed register file [6]. Shadow copies of a register introduce a small overhead since they are independent of the number of ports, i.e. they are not directly accessible.

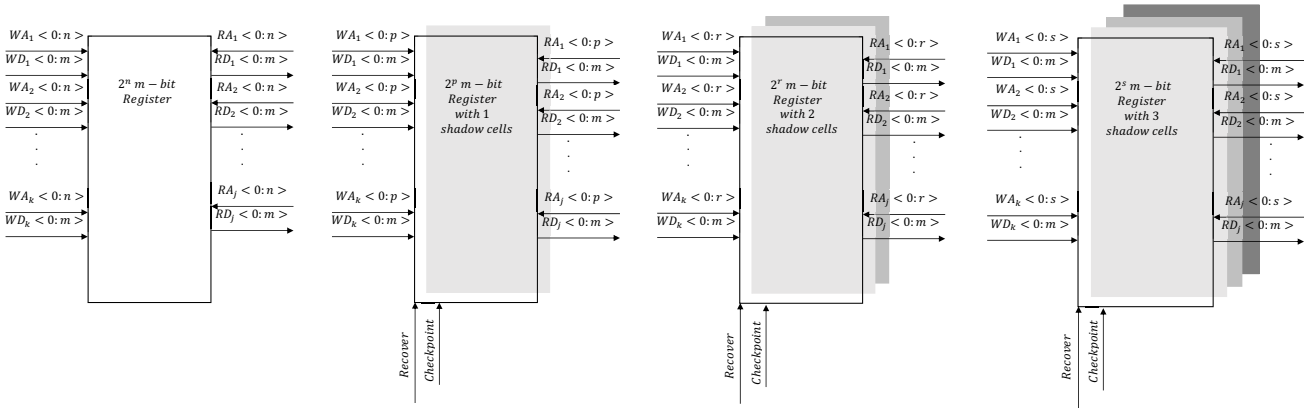


Figure 5: The structure of a four-bank *m-bit* register file with *j* read ports and *k* write ports. (In order from left to right) A bank with single-bit-cell, one, two and three shadow cells.

Previous values of a shared register are only required in the infrequent case of a branch misprediction or exception, whereas only the last version is required for normal execution. Therefore, the most recent version is stored in the normal bits, that are directly accessible, whereas older versions of a shared register are stored in the shadow bits with a small area overhead, and recovered when necessary.

In event of an interrupt or an exception the entire pipeline is flushed. Before the interrupt or the exception handler can be invoked, all logical registers must reflect their state before the interrupt or the exception. To this end, the processor consults the rename and retirement map tables and any entry that differs indicates a logical register whose correct state needs to be recovered from the shadow cells. Although this recovery process may take a few cycles more than in the baseline, the infrequent nature of interrupts and exceptions makes this cost negligible.

C. The Register File

As described in Section IV-B, our renaming scheme employs a check-pointed register file with shadow bit cells to store the different versions of a shared physical register. With a *2-bit counter* in the PRT, our scheme allows up to three reuses of the same physical register, which means that up to three shadow copies must be kept in the register file. Hence, a straightforward implementation would include three shadow cells for each physical register.

Although each shadow copy represents a minor overhead [6], including three shadow copies for each physical register is not cost-effective, since all the copies are not required most of the time. As shown in Figure 3, most of the registers require zero or just one shadow copy, whereas chains of two or three reuses are much less common. Therefore, we propose to split the register file in four banks, where each bank includes registers with zero, one, two or three shadow copies respectively. By using this organization, our scheme is able to cover most of the cases while it avoids the extra cost of including three shadow copies in all the registers. In the following sections we provide further details on the implementation of the register file.

1) The Register File Design And Its Mechanism: To reduce the latency and increase the efficiency in area, register files are implemented in multiple banks [7]. In this work, we propose to have a multi-bank register file in which some of these banks have registers with one, two or three shadow cells embedded.

In such registers, each traditional register bit-cell is backed-up by pairs of cross-coupled inverters which are connected to the main bit-cell using a pass transistor. Shadow cells are accessed only through the main SRAM-cell of the register. Therefore, they do not require any additional read or write ports. A single-bit cell with *n* embedded shadow cells can hold up to *n+1* different contents. The processor can simply manage these contents. At the *write* stage, the value of a register is stored in a shadow cell and a *recover* command copies back the content of a particular shadow cell to the main storage of the register. A physical register can be reused only if it has free shadow cells to store the previous content of the register.

Figure 5 shows the structure of four different banks of a register file with *j* read ports and *k* write ports. As it can be seen, the first bank in the left has single-bit-cell registers while the other banks have register cells with one, two and three embedded shadow cells. The design of a register bit cell with one shadow cell is shown in Figure 6. As this figure shows, the shadow cell is only accessible through the main cell.

The additional area required by the shadow cells is independent of the number of register ports. Therefore, the area overhead of the shadow bits becomes relatively smaller as the number of ports increases. Furthermore, we have observed that most of the registers do not need to have shadow cells. For this reason, the majority of the registers do not have shadow cells which helps to reduce the overhead significantly. We employ a bank with conventional single-bit-cell registers, which includes the majority of the registers,

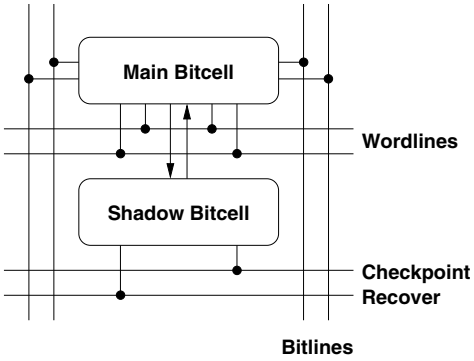


Figure 6: The design of a register bit cell with one shadow cell [6].

and three banks with registers that have one, two and three embedded shadow cells respectively.

When a new physical register is allocated, the register type predictor (described in section IV-D) predicts the expected use of it (single-use/multiple use; number of reuses), and allocates it in the corresponding bank according to this prediction. When a single-use register is written, the previous content of the register is stored in the shadow cell according to the *2-bit* counter of the register ID. In event of branch mispredictions, exceptions or interrupts, a recover command is issued for each register that needs its previous value to place them back to the main storage of the registers. In order to recover the value from the registers with more than one shadow cells, the added 2-bit to the register ID determines the correct shadow cell to recover the previous content of the register. If the physical register does not have enough shadow cells to store the previous content of the register, the register cannot be reused and a new register is allocated at the renaming stage.

2) Impact on the Performance: According to our analysis, there is a very small increase, less than 1%, in the access time of the register file with the shadow cells due to longer word select and bit lines, as reported eslewhere (for instance, see [3]). The reason for the slight delay is that in the design of the registers with shadow cells, no gate capacitance is added.

Whenever a register is to be written in the *write* stage, according to its register ID the value of the register is stored in parallel to the appropriate shadow cell, so no extra latency is added to the write operation. In event of branch mispredictions, exceptions and interrupts there may be some registers whose previous values need to be recovered from the shadow cells. Therefore, recovering the state of the processor in such events may take a few cycles more with respect to the baseline. In our experiments, we have taken this into account.

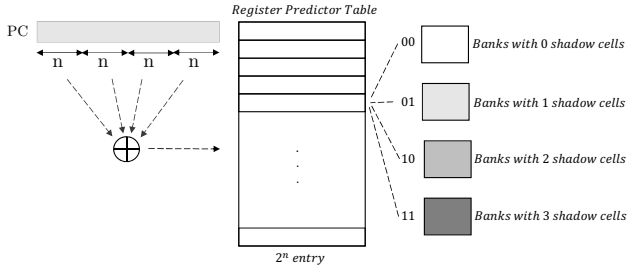


Figure 7: The design of the proposed register type predictor.

D. Register Type Predictor

When an instruction is being renamed, if it needs a new physical register (i.e., the source registers cannot be reused because they do not have shadow cells available or they are not the first use), a hardware predictor determines the type of the register which should be assigned to it. We design a simple 2-bit entry predictor which predicts the type of the register that should be allocated. Using the PC of the instruction, a simple hashing function determines an entry in the register predictor table, as Figure 7 shows. Then, according to the value of the entry, if there is a free register of that type, a new physical register is allocated. In the register predictor table, *00* indicates a normal register (i.e., it implicitly predicts that the register is not single-use) whereas *01, 10* and *11* indicate registers with 1, 2 and 3 shadow cells respectively (i.e., register is predicted to be reused).

If there are no free registers of the predicted type, a register with the closest number of shadow cells will be allocated. In case there is no free register of any type, renaming will stall as in the conventional scheme.

Whenever a physical register is released, the entry in the register predictor table that has been used to allocate this register is updated to reflect the actual number of reuses. If not all the allocated shadow copies have been used, the value of the corresponding entry in the register predictor table is decremented.

On the other hand, if a register that is predicted to be single-use (i.e. has been allocated in a bank with one, two or three shadow copies) is detected to be used more than once, the corresponding entry in the predictor is reset to zero.

Finally, if a register predicted to be single-use is tried to be reused (i.e. it is the source operand of an instruction and it is the first use) but there are no shadow cells available, the register is not reused and the corresponding entry in the predictor is increased, so that next time it allocates a register with a greater number of shadow copies.

1) Handling Single-Use Mispredictions: A register predicted to be single-use may be reused and later encounter that the single-use prediction was wrong because there is an additional use. This is illustrated in Figure 8. Register r_1 is predicted to be a single-use register. Therefore, its physical register (P_1) is assigned to register r_x. Later, while renaming instruction *3*, it is discovered that r_1 is mispredicted to

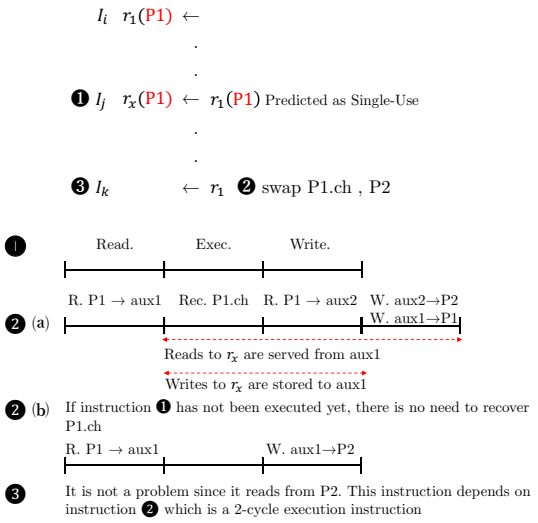


Figure 8: Timing for the proposed micro-operations to move the value of the check-pointed register.

be a single-use register. Now, since the physical register P_1 has been assigned to another register, the register r_1 in instruction *3* cannot be renamed to P_1 as it holds the value of r_x. Since register r_x is renamed to P_1 and there might be instructions before instruction *3* that use r_x, it is more cost effective to rename further consumers of r_1 to a new physical register. Therefore, the value of the r_1 which was in physical register P_1 needs to be moved to this new physical register.

For this purpose, we propose to use two different micro-operations, depending on whether instruction *1* has been executed when instruction *3* is being renamed or not, which move the value of r_1 to a new physical register (instructions *2(a)* and *2(b)* in Figure 8). If instruction *1* has been executed before the instruction *3* is being renamed, the value of r_1 is check-pointed in a shadow cell of P_1 ($P_{1.ch}$). Hence, as instruction *2(a)* in Figure 8 shows, first, the current value of P_1 is stored in an auxiliary register. Then, the check-pointed value in P_1 ($P_{1.ch}$) is being recovered and moved to another auxiliary register. Finally, last step is to move back the correct value of P_1 and move the recovered value of r_1 to the new physical location. If instruction *1* has not been executed before the instruction *3* is being renamed, the value of r_1 is not check-pointed and it only needs to be moved to the new physical register (see the timing for instructions *2(b)* in Figure 8).

V. Methodology

A. Simulation Environment

In this work, we model an out-of-order ARM processor using the GEM5 [8] cycle-accurate simulator with the parameters presented in Table I. This processor uses a merged register file and releases a physical register when the redefining instruction commits. The simulator models in detail both the baseline and our proposed technique. We use CACTI 6.5 [9] to estimate area of the register files, PRT, issue queue and the predictor. CACTI 6.5 supports different models for SRAMs, DRAMs and register files. We specify different parameters including the number of read/write ports, number of banks, technology, bus width, etc. to estimate the area, power and latency of different configurations.

	Configuration
Core	ARMv8 ISA, 2.0 GHz, 128-entry ROB 40-entry Issue Queue, Fully Out-of-Order 3-Width Decoder, 3-Width Instruction Dispatch 48 KB, 3-Way TLB 48-Enrty Fully-Associative L1 TLB
Caches	32 KB L1-D Cache, 2-Way, 1 Cycle 48 KB L1-I Cache, 3-Way, 1 Cycle 1 MB L2 Cache, 16-Way, 12 Cycles 64 Bytes Cache Line Size
Prefetcher	Stride Prefetcher (Degree 1) 2K Branch Target Buffer (BTB) 32-Instruction Fetch Queue 15 Cycles Misprediction Penalty
DRAM	DDR3 1600 MHz, 2 Ranks/Channel 8 Banks/Rank, 8 KB Row Size. t_{CAS} = t_{RCD} = t_{RP} = CL = 13.75ns t_{REFI} = 7.8 us

Table I: System Configuration.

B. Benchmarks

We use SPECfp and SPECint CPU2006 [10] benchmarks for our experiments. We run the benchmarks using the *ref* inputs provided in the SPEC software package. All the SPEC benchmarks have been compiled using GCC version 4.8.4 with *-O3 -mtune=armv8* optimization flags. For each benchmark 5 billions committed instructions have been simulated. In addition to SPEC2006 CPU benchmarks, we use Mediabench [11] benchmark suite.

In recent years, machine learning applications including voice, video and image processing have become very popular. Gaussian Mixture Models (GMMs) [12], [13] and Deep Neural Networks (DNNs) are among the main kernels commonly used in many of these applications, so we add these two benchmarks in addition to the SPEC2006 and Mediabench.

VI. Experimental Results

In this section, we evaluate the proposed register renaming technique explained in Section IV. In our experiments, we consider the overheads of the proposed renaming technique in order to make a fair comparison with respect to the baseline system, including the area of the PRT, register files, issue queue and the predictor. First, we evaluated the area for these units in the baseline and later we reevaluated the area considering the required changes for our renaming technique. In order to make comparisons with the same area, we consider the area of overheads of our technique, and we adjust the number of registers in the register file for our

Units	Configuration	Area (mm^2)
Integer Register File (64-bit registers)	128 Registers	0.2834
Floating-point Register File (128-bit registers)	128 Registers	0.4988
PRT	**Overhead**	5.08 E-04
Issue Queue	**Overhead**	1.48 E-03
Register Predictor	**Overhead**	3.1 E-03
Total Overheads		5.085 E-03

Table II: Area for the register file, register map table, issue queue and the register predictor.

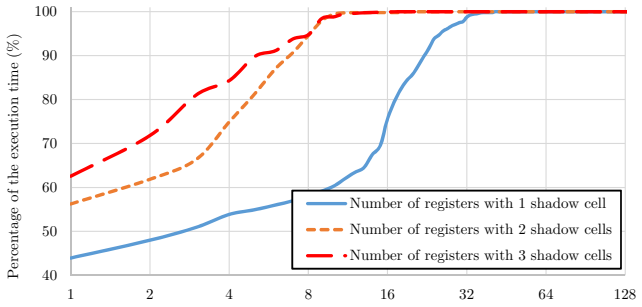


Figure 9: Number of physical registers with 1, 2 and 3 shadow cells needed to cover different percentages of the SPECfp execution time.

renaming scheme in such a way that the total area becomes the same as the baseline.

The additional area required by the shadow cells is independent of the number of register file ports and it becomes relatively smaller as the number of register file ports increases. Besides, since we use shadow cells only for a small percentage of the registers, their area overhead becomes very small (normally in the order of a few physical registers). Unless stated otherwise, comparisons between our technique and the baseline are performed assuming the same total area for both, including the overheads.

A. Size of Different Banks in the Register File

1) Overheads of the proposed scheme: In the first place, we calculate the overheads that our scheme adds to the baseline system. Regarding the issue queue and PRT table, we consider the changes in the sizes of these units with respect to the baseline. Furthermore, the register type predictor has a table with the size of 1K bits. Table II shows the summary of area overheads in different modified units for our scheme. As it is shown, the overheads are small in comparison with the size of the register file.

2) Adjusting the sizes of each bank in the register file: Considering the aforementioned overheads, we performed a sensitivity analysis to identify the most convenient size of the different banks in the register file. For this study, we assumed an unbounded number of registers with up to three shadow cells. Figure 9 shows different number of physical registers with different number of shadow cells to cover different

Register file configuration for the baseline	Register file configuration for our scheme 0-sh, 1-sh, 2-sh, 3-sh			
48	28	4	4	4
56	28	6	6	6
64	36	6	6	6
72	36	8	8	8
80	42	8	8	8
96	58	8	8	8
112	75	8	8	8

Table III: Register File Configuration.

percentages of the SPECfp execution time. Hence, based on this study we tune the number of the registers of each bank in the floating-point register file. Similarly, we tune the number of registers in each bank for the integer register file. Table III shows the equivalent sizes of the register file that we consider. For each particular size of the register file in the baseline system, a hybrid register file configuration of the same area has been considered to evaluate the proposed scheme.

B. Performance Improvement

We present the performance results for the proposed register renaming technique in comparison with the baseline system. We assume a register file with four types of banks as shown in Figure 5. Figure 10 presents the performance improvements with respect to the baseline for different sizes of the register file. Note that the integer and floating-point register files are decoupled. Hence, for integer benchmarks we consider different sizes of the integer register file whereas for floating-point benchmarks we measure performance for different sizes of the floating-point register file.

As Figure 10a shows, for SPECfp benchmarks, the proposed technique provides 12.2%, 7.5%, 3.75%, 1.83% and 0.82% performance improvements on average using a register file of equivalent size for our proposed technique. Similarly, for SPECint benchmarks, see Figure 10b, the proposed technique provides 47%, 6.76%, 2.29%, 0.67% and 0.41% performance improvements on average with respect to the baseline. As the results show, for small register files, the benefits are high. As the register file size increases, the benefits decrease since the register file becomes less critical and a more effective use of it has smaller benefits.

For Mediabench and cognitive computing benchmarks, the proposed scheme provides significant performace improvements, as Figure 10c shows.

Figure 11 shows the average committed instructions per cycle (IPC) for the baseline and the proposed scheme. The X axis represents the number of physical registers in the baseline. For the proposed technique, we assume a register file of equivalent area, taking into account its overheads. As it can be seen, our scheme can achieve the same performance as the baseline with a significantly lower number of registers. For instance, our technique with a floating-point register

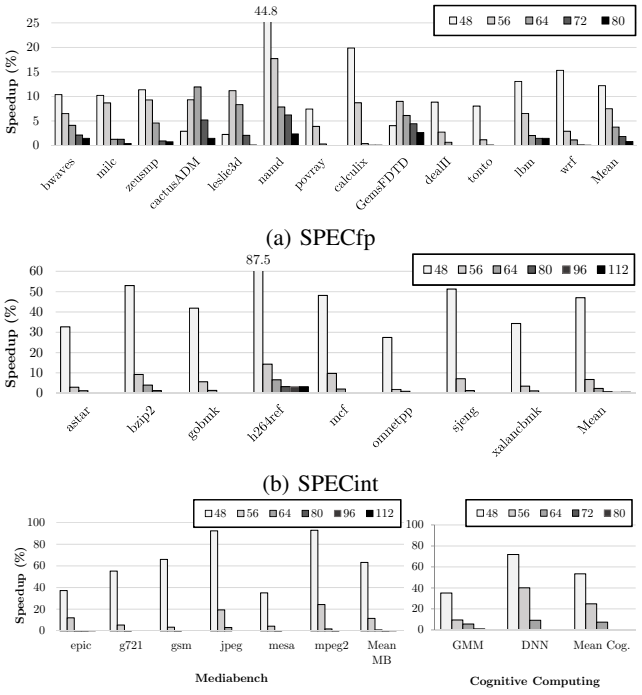


Figure 10: Speedups achieved for each benchmark with respect to the baseline system for different sizes of the register file. The proposed system has a multi-bank register file with 4 banks: a conventional one and banks with 1, 2 and 3 shadow cells. A register can be reused up to 3 times.

file equivalent to 56 registers achieves the same IPC as the baseline with 64, which represents a saving of 13% in area.

C. Analysis on Register Type Predictor

As described in section IV, the proposed technique uses a simple predictor with 512 entries that predicts the most likely reuse for each register and it is the configuration assumed in the experiments of this work. In case the prediction corresponds with the actual number of reuses, we count it as a hit; otherwise it is counted as a miss.

As we discussed in section III, more than 85% of the instructions, on average, require a destination register for which in our scheme we use the register predictor. Therefore, a register may be predicted to be reused correctly or incorrectly. On the other hand, the predictor may predict not to reuse a register which again can be a correct or an incorrect prediction. Figure 12 shows a breakdown for all the instructions of SPECint and SPECfp benchmarks. As this figure shows, despite the instructions that do not have a destination register, the rest will fall into one of these four categories.

Using the predictor two different kind of misprediction may occur. First, there may be a possibility to reuse a single-use register, however, due to an incorrect prediction, the register is not reused. In this case, that represents 2.28%

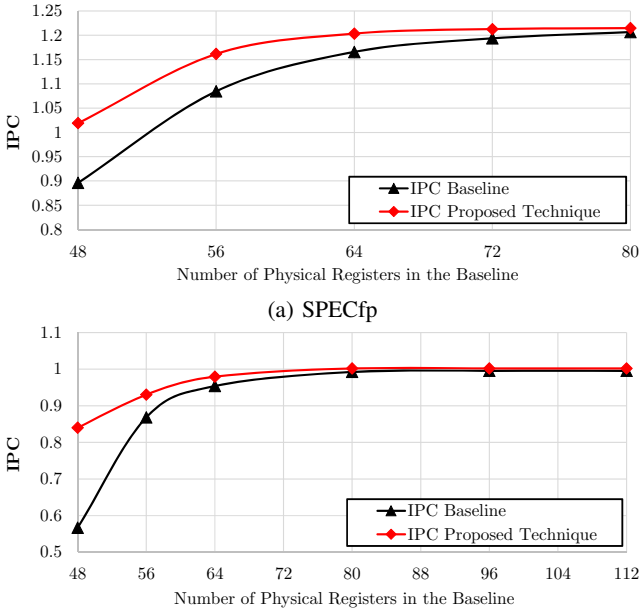


Figure 11: IPC of the proposed scheme and the baseline for different sizes of register files in the baseline and in the proposed technique.

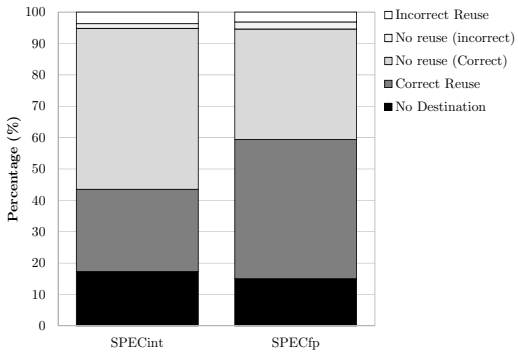


Figure 12: Accuracy of the register type predictor.

of the instructions in SPECfp, an opportunity of reusing a register is lost but no further actions are required. On the other hand, a register may have more than a consumer and it is predicted as a single-use register. In this case, that represents 3.1% of the instructions in SPECfp, as the register is reused incorrectly, the previous value of the register needs to be recovered as discussed in section IV-D1.

D. Complexity of the Proposed Register Renaming Scheme

Our register renaming technique introduces a small overhead. The new hardware structures included are fairly small: the PRT requires up to 384-bits and the register predictor table contains 1-Kbits. Regarding the register file, shadow copies are much cheaper than regular registers as they are not connected to read/write ports. Furthermore, only a small number of registers include shadow copies as shown in Table III. Finally, the issue queue requires 4 additional

bits per entry, which is small compared to the information already stored in modern out-of-order processors.

Regarding the renaming logic, our scheme adds an indirection from Register Map Table to the PRT and it requires an access to the predictor table. Handling dependencies among instructions renamed in the same cycle is not a problem, since out-of-order processors already handle this case and include additional checks and bypasses. Our scheme only requires extra checks to the Read-bit and 2-bit counter. Although this extra complexity in the renaming might impact the total delay of the rename stage, we assume our technique has no impact on cycle time for two reasons. First, some of the latencies can be overlapped, for example Register Map Table and predictor table can be accessed in parallel. Second, renaming is not typically in the critical path of modern out-of-order processors. In the worst case, we can further pipeline the renaming, since adding one stage to the front-end results in negligible impact on the overall IPC as reported elsewhere (see for instance [14]).

Finally, all comparisons in this paper are done for configurations with the same area, in order to show that our renaming scheme is better than simply adding more registers.

VII. Related Work

Using physical registers in a more efficient way has been the goal of many researches in the past. The most similar works to the technique presented in this paper are those that try to delay the allocation of registers or anticipate its release.

Monreal et al. [15] proposed a register renaming technique based on virtual-physical registers [16]. By employing virtual-physical registers, their approach postpones the physical register allocation until the corresponding instruction finishes its execution. Their approach has a significant cost due mainly to the requirement of two separate register map tables, and two mapping operation per each destination register (from logical to virtual-physical, and from virtual-physical to physical).

Several works have been proposed to early release a register. Moudgill et al. [17] suggested to release physical registers, as soon as the last instruction that redefines a register commits. The last-use tracking is based on counters which record the number of pending reads for every physical register. This initial proposal did not support precise exceptions since the counters were not correctly recovered when instructions were squashed. Later, Akkary et al. [18] proposed to improve the Moudgill scheme by adding an unmapped flag for each physical register, which is set when a subsequent instruction redefines that logical register. Then, a physical register can be released once its usage counter is zero and its unmapped flag is set. Moreover, for proper exception recovery of the reference counters, when a check-point is created, the counters of all physical registers belonging to the check-point are incremented. Similarly, when a check-point is released, the counters of all physical registers belonging to the check-point are decremented. In addition to the overheads that these techniques impose, they release a register far later than its actual lifetime.

Monreal et al. [19] proposed two different schemes to release a register. The first scheme waits for a redefining instruction to become non-speculative before releasing the previous version of its logical register. The second adds a new queue with multiple levels corresponding to the unconfirmed branches in the reorder buffer (ROB). Registers are released when the redefining instruction becomes non-speculative and the last instruction using the physical register has committed. On a branch misprediction, the relevant levels in the release queue are squashed. The downside of these techniques is that no recovery mechanism is in place to retain values released early. In the event of an exception or interrupt it would be impossible to reconstruct the precise processor state. They also need to add many large structures to the processor so that the status of redefining and last-use instructions can be maintained, increasing the complexity of the pipeline.

Ergin et al. [3] introduced a check-pointed register file to implement early register release. In their approach, a register is being deallocated immediately after the instruction producing the register's value commits itself and all potential consumers of this value have started execution before the redefining instruction is known to be non- speculative. To support branch misprediction, precise exceptions and interrupts, their proposal save the register value into the shadow bit-cells of the register where it can be accessed easily in such events.

Jones et al. [6] proposed a compiler-based technique for early register release. The compiler defines the points where registers will no longer be used and can be safely released. To guarantee that the state of the processor can be safely recovered after an interrupt or an exception, they used a check-pointed register file similar to the register file proposed in [3]. This scheme requires compiler support and changes in the ISA in addition to the overheads of the shadow cells in the entire register file.

Quinones et al. [20] proposed two techniques to release registers in out-of-order processors with register windows. The proposed techniques are based on the observation that when none of the instructions of a procedure are currently inflight, all mappings of the procedure context are not needed. Therefore, these mappings and their associated physical registers can be released. Although their approach is beneficial, it is specific for processors with register windows, which are not common nowadays.

Note that previous work on early register release like [17] and [19] do not support precise exceptions. Precise exceptions is a must for modern out-of-order processors and, hence, these previous techniques cannot be implemented

on modern CPUs. Work in [20] is specific to processors with register windows which are very uncommon nowadays. Finally, work in [6] requires compiler support and changes to the ISA. ISA extensions become legacy and make the technique less attractive. In comparison, our scheme supports precise exceptions and does not require any ISA extension. In short, previous techniques are not suitable for modern processors and, to the best of our knowledge, our technique is the only one that can reuse a physical register as early as the last use of this register is renamed, as opposed to other techniques that need to wait at least until the last use instruction commits.

VIII. CONCLUSIONS

In this paper, we show that the value generated by more than 50% of the floating-point instructions in SPECfp benchmarks and more than 30% of the integer instructions in SPECint benchmarks are consumed only by one instruction.

Based on this observation, we propose a novel register renaming technique for out-of-order processors that allows to reuse this single-use registers for the destination operand, instead of allocating a new register. To recover the state of the processor in the event of branch mispredictions, exceptions and interrupts, we employ a multi-bank register file in which some banks have registers with integrated shadow cells. A simple register predictor is proposed to allocate the most beneficial type of the register for each instruction. Considering similar costs in hardware, we show that the proposed technique provides 6% speedup on average for the SPEC2006 benchmarks. Alternatively, the same performance as the baseline can be achieved while reducing the area of the register file by 10.5%.

ACKNOWLEDGMENT

This work was supported by the Spanish State Research Agency under grants TIN2013-44375-R and TIN2016-75344-R (AEI/FEDER, EU).

REFERENCES

[1] D. W. Wall, *Limits of instruction-level parallelism*, vol. 19. ACM, 1991.

[2] K. I. Farkas, N. P. Jouppi, and P. Chow, "Register file design considerations in dynamically scheduled processors," in *High-Performance Computer Architecture, 1996. Proceedings., Second International Symposium on*, pp. 40–51, IEEE, 1996.

[3] O. Ergin, D. Balkan, D. Ponomarev, and K. Ghose, "Increasing processor performance through early register release," in *Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings. IEEE International Conference on*, pp. 480–487, IEEE, 2004.

[4] A. Gonzalez, F. Latorre, and G. Magklis, "Processor microarchitecture: An implementation perspective," *Synthesis Lectures on Computer Architecture*, vol. 5, no. 1, pp. 1–116, 2010.

[5] G. S. Sohi, "Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers," *IEEE transactions on computers*, vol. 39, no. 3, pp. 349–359, 1990.

[6] T. M. Jones, M. O'Boyle, J. Abella, A. Gonzalez, and O. Ergin, "Compiler directed early register release," in *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*, pp. 110–119, IEEE, 2005.

[7] J.-L. Cruz, A. González, M. Valero, and N. P. Topham, "Multiple-banked register file architectures," in *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pp. 316–325, IEEE, 2000.

[8] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[9] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP Laboratories*, pp. 22–31, 2009.

[10] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[11] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems," in *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pp. 330–335, IEEE Computer Society, 1997.

[12] H. Tabani, J.-M. Arnau, J. Tubella, and A. Gonzalez, "An ultra low-power hardware accelerator for acoustic scoring in speech recognition," in *Parallel Architectures and Compilation Techniques (PACT), 2017 26th International Conference on*, pp. 41–52, IEEE, 2017.

[13] H. Tabani, J.-M. Arnau, J. Tubella, and A. González, "Performance analysis and optimization of automatic speech recognition," *IEEE Transactions on Multi-Scale Computing Systems*, 2017.

[14] E. Sprangle and D. Carmean, "Increasing processor performance by implementing deeper pipelines," in *ACM SIGARCH Computer Architecture News*, vol. 30, pp. 25–34, IEEE Computer Society, 2002.

[15] T. Monreal, A. González, M. Valero, J. González, and V. Viñals, "Dynamic register renaming through virtual-physical registers," *Journal of Instruction Level Parallelism*, 2000.

[16] A. Gonzalez, J. Gonzalez, and M. Valero, "Virtual-physical registers," in *High-Performance Computer Architecture, 1998. Proceedings., 1998 Fourth International Symposium on*, pp. 175–184, IEEE, 1998.

[17] M. Moudgill, K. Pingali, and S. Vassiliadis, "Register renaming and dynamic speculation: an alternative approach," in *Proceedings of the 26th annual international symposium on Microarchitecture*, pp. 202–213, IEEE Computer Society Press, 1993.

[18] H. Akkary, R. Rajwar, and S. T. Srinivasan, "Checkpoint processing and recovery: Towards scalable large instruction window processors," in *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pp. 423–434, IEEE, 2003.

[19] T. Monreal, V. Viñals, A. González, and M. Valero, "Hardware schemes for early register release," in *Parallel Processing, 2002. Proceedings. International Conference on*, pp. 5–13, IEEE, 2002.

[20] E. Quinones, J.-M. Parcerisa, and A. Gonzalez, "Early register release for out-of-order processors with register windows," in *Parallel Architecture and Compilation Techniques, 2007. PACT 2007. 16th International Conference on*, pp. 225–234, IEEE, 2007.