# A Power-Aware Hybrid RAM-CAM Renaming Mechanism for Fast Recovery

S. Petit, R. Ubal, J. Sahuquillo and P. López
*Department of Computer Engineering (DISCA)*
*Universidad Politécnica de Valencia, Spain*
spetit@disca.upv.es, raurte@gap.upv.es, {jsahuqui, plopez}@disca.upv.es

***Abstract*— Modern superscalar processors implement register renaming by using either RAM or CAM tables. The design of these structures should address their access time and misprediction recovery penalty. While direct-mapped RAMs provide faster access times, CAMs are more appropriate to avoid recovery penalties. Although they are more complex and slower, CAMs usually match the processor cycle in current designs. However, they do not scale with the number of physical registers and the pipeline width.**

**In this paper we present a new hybrid RAM-CAM register renaming scheme, which combines the best of both approaches. In a steady state, a RAM provides the current mappings quickly; on mispeculation, a low-complexity CAM enables immediate recovery and further register renaming. Compared to an ideal CAM in a 4-way state-of-the-art superscalar microprocessor, and for almost the same performance (1% slowdown) and area (95% of the ideal CAM size), the proposed scheme consumes about 90% less dynamic energy.**

## I. Introduction

High performance microprocessors implement out-of-order and speculative execution to increase performance. In this context, many mechanisms have been devised aimed at enhancing the amount of instructions executing concurrently. These microarchitectural mechanisms require register renaming techniques in order to overcome *write after read* (WaR) and *write after write* (WaW) data hazards.

Register renaming techniques distinguish two kinds of registers: logical and physical registers. Logical or architected registers refer to those used by the compiler, while physical registers are those actually implemented in the machine. Typically, the number of physical registers is quite larger than the number of logical registers. When an instruction that produces a result is decoded, the renaming logic allocates a *free* physical register. Then, the destination logical register is said to be mapped to that physical register. After that, subsequent data dependent instructions rename their source logical registers to access this physical register. In addition, to deal with program execution correctness, the register renaming circuitry must also recover the register mapping table on mispeculation and release the physical registers allocated to mispeculated instructions. To this end, some microprocessors perform checkpoints of the renaming table [1][2][3].

Two types of memory structures have been traditionally used for register renaming: static RAMs or CAMs. Both of them present advantages and shortcomings, and the industry does not show a predominant trend for either of them. For example, the MIPS R10000 [1] and the Pentium4 [4] use the RAM approach, while the Power4 [3] and the Alpha 21264 [2] include a CAM with a large number of checkpoints.

TABLE I
RECOVERY CYCLES IN A TYPICAL RAM-BASED APPROACH.

| Triggered at | SpecInt | SpecFP | Average |
|---|---|---|---|
| Commit stage | 12.4 | 53.5 | 33.0 |
| Writeback stage | 3.8 | 29.4 | 16.6 |

To rename a source logical register, its identifier is used to obtain the current mapping. This function is performed faster and more efficiently in terms of energy by a RAM structure. The reason is that RAMs are organized as direct-mapped structures, while CAMs are fully associative tables. This means that the RAM table is directly indexed by a source logical register, whereas this register is compared against all current mappings in the CAM. The associative search is a major concern in CAM-based approaches, not only because of the long access time it involves, but also because it hinders scalability with the number of physical registers [5].

Concerning renaming of destination logical registers, RAM-based approaches require at least two major structures: the renaming table to keep current mappings, and the free register queue, which is used to track the set of free physical registers. In contrast, CAM-based approaches have the proper layout to easily allocate new mappings without additional assistant structures [6][7].

Regardless of the approach used, checkpoints allow quick recovery of the correct mappings after mispeculation. Thus, checkpoints of the renaming tables are implemented in both RAM-based [1] and CAM-based [2][3] processors. However, when the number of checkpoints surpasses a certain limit, CAM checkpointing becomes faster and more energy-efficient than RAM checkpointing [5]. In addition, only a CAM is able to perform in constant time the reclamation of registers previously allocated by mispeculated instructions (see Section II-B).

Alternatively, in RAM-based approaches the reclamation of these registers is performed by either waiting for the offending instruction to reach the reorder buffer (ROB) head [4] or scan the ROB when a misprediction is raised at the $writeback$ stage [1]. Table I shows the recovery penalty time (in processor cycles) in a RAM-based state-of-the art microprocessor[1]. As observed, the penalty of triggering

[1]These results were obtained with the baseline processor configuration described in Section IV

978-1-4244-5028-2/09/$25.00 ©2009 IEEE

recovery at *commit* is, on average, twice as large as doing it at the *writeback* stage. But even in the latter case, the amount of cycles is not negligible.

In this paper, we propose a new scheme that tries to take the best of each implementation, that is, fast register renaming, fast register reclamation, and fast recovery. To this end, we propose a hybrid approach that uses both a RAM and a CAM. During correct path execution, the RAM is in charge of providing most of the mappings, and can be seen as a cache of the CAM. The CAM is checkpointed whenever a branch is decoded, enabling quick mispeculation recovery, which is followed by an invalidation of the RAM contents. After recovery, the RAM is progressively refilled with correct mappings while new instructions enter the pipeline.

The advantages of the hybrid design come from two main facts. On one hand, processors work in a non-speculative mode in the common case, so RAM invalidations are unusual. On the other hand, frequently executed instructions (e.g., loops) only use a small subset of the architected register file, so only few RAM updates suffice to recover the steady state. Thus a reduction of the CAM complexity in terms of number of ports is possible without hurting performance, causing lower power consumption, area, and access time. This makes the proposed mechanism more scalable with the number of physical registers than typical CAM-based approaches.

Experimental results show that the hybrid scheme achieves almost the same performance than a CAM-based approach in a 4-way state-of-the-art superscalar processor, while reducing the number of CAM searches by about 95%. Consequently, the dynamic energy consumption drops by about 90%. Likewise, the area is reduced by between 5% and 37%, depending on the hybrid configuration.

The remainder of this paper is organized as follows. Section 2 discusses typical renaming mechanisms. Section 3 describes the hybrid RAM-CAM approach. Section 4 presents experimental results. Section 5 discusses some related work. Finally, Section 6 draws some concluding remarks.

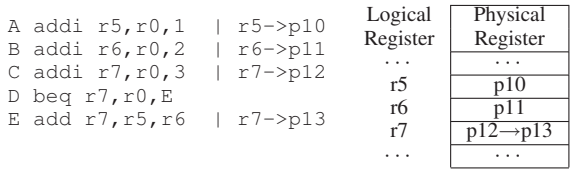
## II. Background

Renaming techniques involve renaming source and destination logical registers. Renaming source registers is straightforwardly performed by looking up the current mapping. Renaming a destination logical register requires allocating a free physical register and updating this mapping in the renaming table. Below, we describe how these mappings are tracked in both purely RAM- and CAM-based mechanisms.

### A. RAM approach

Let us see how a typical RAM-based approach works through a working example. Figure 1(a) shows a code consisting of five instructions, which rename three logical registers (`r5`–`r7`) to four physical registers (`p10`–`p13`).

Figure 1(b) shows the RAM contents involved at the time instruction `E` enters the rename stage. At this point, its source logical registers are renamed to `p10` and `p11` (contents of entries `r5` and `r6`, respectively). In addition, a free physical register (`p13`) is allocated to the destination register `r7`, and this entry is updated accordingly (as the arrow indicates).




a) Code example b) RAM table excerpt

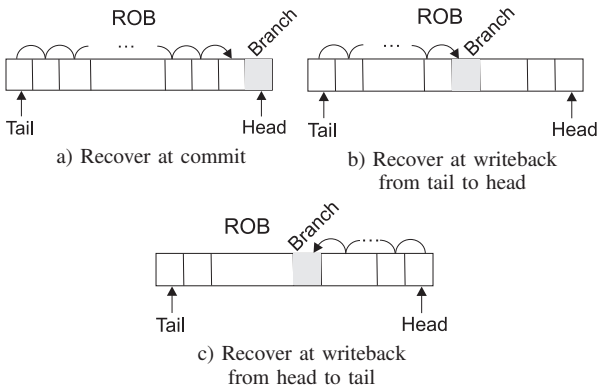
Fig. 1. RAM working example.

To allocate a free physical register, RAM approaches use an additional free fegister queue (FRQ), that is, a FIFO structure tracking the identifiers of the physical registers that are not being used by instructions in flight. A free register is obtained from the FRQ each time a producer instruction enters the rename stage [6]. After that, subsequent instructions having a data dependence with `r7` will be renamed to `p13`. The direct-mapped memory organization allows the mappings to be rapidly performed while taking up small area. Later, at the commit stage, physical registers are released by placing their identifiers back into the FRQ.

Since the RAM table is updated at rename, it can be updated by either non-speculative or speculative instructions. On mispeculation, those changes performed by mispeculated instructions must be canceled. That is, the RAM state must be restored to its previous state at the time the offending instruction (e.g., a mispredicted branch) entered the rename stage. In addition, registers allocated to mispeculated instructions must be placed back into the FRQ.

The simplest way to do this is to wait until the mispredicted branch reaches the Reorder Buffer (ROB) head. The ROB is a FIFO structure where instruction metadata is stored in program order in the corresponding entry until retirement. Among this information, ROB entries contain the previous mapping (e.g., `p12` for instruction `E`). On mispeculation, the correct RAM state can be restored by scanning the ROB once the offending instruction reaches the ROB head. From now on, we will refer to this technique, depicted in Figure 2(a), as *recover at commit*.

*Recover at commit* incurs a penalty with two main components: i) the time elapsed since the misprediction is known until the mispredicted instruction reaches the commit stage,




a) Recover at commit b) Recover at writeback from tail to head

c) Recover at writeback from head to tail

Fig. 2. Recovery schemes.

and ii) the time required to restore the correct mappings. The second component can be reduced by using two RAMs, a front-end RAM (FRAM) and a retirement RAM (RRAM), as implemented in some modern microprocessors [4]. The FRAM is updated at rename, as usual, while the RRAM is updated as non-speculative instructions leave the ROB. Thus, the RRAM contains a delayed validated copy of the FRAM. In this context, a simple way to implement the recovery mechanism is waiting until the offending instruction reaches the ROB head, and then copying the RRAM contents into the FRAM. The main shortcoming of this implementation is that the first component of the penalty time is not reduced at all. Performance is adversely impacted, for instance, if the mispredicted instruction is preceded by a long latency instruction, such as an L2 miss.

To reduce this component, recovery should be triggered before the commit stage. This can be done by recovering the renaming table as soon as the misprediction is known, i.e., at the writeback stage. This approach will be referred to as *recover at writeback*. In this case, the overall penalty time is reduced with respect to *recover at commit*, since recovery starts immediately and only a fraction of the ROB needs to be walked. In this context, if two RAMs are implemented, the renaming table can be restored by walking the ROB either from its tail towards the offending instruction (see Figure 2(b)), or from its head towards the offending instruction (see Figure 2(c)).

Although it is possible to perform checkpoints of the FRAM to accelerate recovery, there are two important reasons for not using them extensively. First, the ROB still needs to be walked in order to release registers allocated by mispeculated instructions. Second, RAM tables scale worse than CAM tables as the number of checkpoints increases [5]. As a matter of fact, RAM-based commercial implementations support a small number of checkpoints. For instance, the MIPS R10000 [1] only supports 4 checkpoints, while the Pentium4 [4] does not implement checkpoints at all.

### B. CAM approach

CAM structures have as many rows as the number of physical registers. Each row maintains information for renaming, recovery and register reclamation as shown in Figure 3. The first two columns are involved in the renaming process. The first one indicates the mapped logical register, whereas the second column specifies whether this mapping is or is not the current one. Physical registers not mapped to any logical register have the *current mapping* bit clear.

Assume a simple processor where register mappings are checkpointed every time a branch instruction is decoded, and just one unresolved branch is allowed to be in flight. Figure 3 shows an excerpt of a CAM table containing the current mappings and one branch checkpoint. The state corresponds to the code studied above (Figure 1(a)) at the time instruction `E` enters the rename stage. Again, arrows in the table represent content transitions.

| Physical Register | Logical Register | Current Mapping | Branch Checkpoint | Free |
|---|---|---|---|---|
| ... | ... | | | |
| P10 | R5 | 1 | 1 | 0 |
| P11 | R6 | 1 | 1 | 0 |
| P12 | R7 | 1→0 | 1 | 0 |
| P13 | R7 | 0→1 | 0 | 1→0 |
| P14 | – | 0 | 0 | 1 |
| ... | ... | | | |

Fig. 3. CAM table excerpt.

The source registers of `E` (`r5` and `r6`) are renamed to `p10` and `p11`, respectively, by searching the current mappings in the CAM. In addition, the destination logical register `r7`, previously mapped to `p12`, is remapped to `p13`, which is obtained by means of a priority encoder (PE) connected to column *free*. Then, this mapping is updated in the corresponding entries (logical register and current mapping) of the CAM. At the same time, the *current mapping* entry of `p12` must be reset. Finally, the branch checkpoint keeps a copy of the *current mapping* column performed at the time branch `D` was decoded. Depending on the implementation, a checkpoint can be performed either for each instruction [2] or just for some of them, such as low-confidence branches [8][9]. To this aim, only the *current mapping* column needs to be copied, which is a fast and unexpensive process.

Let us analyze how this organization provides fast register reclamation and recovery. Regarding register reclamation, a physical register is assumed to be free when its *current mapping* bit is clear and it is not used in any checkpoint. In the example, `p10`, `p11`, and `p13` are currently mapped, so they cannot be released. Also, although `p12` is not currently mapped, it cannot be released until the checkpointed branch instruction `D` is resolved and known to be non-speculative, since `p12` could become the current mapping for `r7` again if speculation for `D` fails. Finally, `p14` is free because it was not mapped before branch `D`. Thus, free registers can be straightforwardly obtained by simply *nor*-ing the *current mapping* and the *branch checkpoint* bits. In practice, the same behavior can be accomplished with lower hardware cost when a checkpoint is released, by testing if it is the last checkpoint including a given physical register [7]. The latter technique has a constant complexity independent of the the number of checkpoints.

To recover the processor state on misprediction, it suffices to copy the branch checkpoint column into the current mapping column in order to return the CAM to its state at the time the branch was decoded. This can be done as soon as the misprediction is resolved, i.e. at the writeback stage. In the example, `p13` would be automatically released, since the restored column has its corresponding entry clear. In summary, the CAM approach returns to a previous machine state by just copying a bit column.

The cited advantages make checkpoints much more appealing in CAM-based implementations than in RAM-based ones. For example, the Alpha 21264 [2] and the Power4 [3] implement 80 and 20 checkpoints, respectively. Moreover, supporting a high number of checkpoints offers significant performance gains, especially for large instruction windows, because execution can be recovered with a small penalty from any of its intermediate states.

## III. HYBRID RAM-CAM

The proposed mechanism combines the advantages of both approaches described above. On one hand, it provides fast register renaming with direct-mapped lookups as done in the pure RAM scheme; on the other hand, it allows fast recovery on mispeculation, as provided by pure CAM-based implementations. The hybrid scheme uses two tables: $i$) a CAM containing all register mappings up to date, and $ii$) a RAM acting as a cache of the CAM, which contains a subset of the renaming information stored in the CAM. The CAM table can be indexed both directly by a physical register or associatively by a logical register, while the RAM is indexed by a logical register. Unlike typical RAMs, an entry may or may not contain a valid copy of the current mapping, which is indicated by an additional $valid$ bit attached to each entry. If this bit is clear for a given entry, a RAM miss is said to occur when this entry is accessed.

Register renaming is performed by just accessing the RAM as long as valid entries are hit. On a RAM miss, the CAM is used to retrieve the current mappings. On mispeculation, the RAM contents are invalidated and progressively restored each time a register mapping is obtained from the CAM. A complete invalidation of the RAM is overconservative, since the mappings performed by instructions previous to the offending one are correct. Nevertheless, the slight impact on performance shown in our experiments allows us to avoid a (probably more complex) selective RAM invalidation.

Let us see how the previous working example behaves in the hybrid RAM-CAM proposal. As instructions in Figure 1(a) enter the rename stage, destination registers are mapped to physical registers allocated by the PE both in the RAM and the CAM. Thus, when instruction `E` is being renamed the RAM and CAM tables are in the state shown in Figure 1(b) and Figure 3, respectively. The source registers are renamed by means of the RAM because its mappings are valid.

In this scenario, assume that branch `D` is resolved as mispredicted. Instruction `E` becomes then mispeculated, and consequently the transition of entry `r7` from `p12` to `p13` must be undone. In addition, `p13` must be deallocated. Recovery in the hybrid RAM-CAM approach works as follows. First, the contents of the RAM are completely invalidated by resetting all $valid$ bits in the RAM entries. Second, the CAM is recovered by simply restoring the current mapping column to the branch checkpoint performed when the branch instruction `D` was decoded. Notice that both operations are simple and only involve updating a bit column in each table. Figure 4 presents this CAM transition. In this way, `p12` becomes again the current mapping of `r7`, but this information (in fact, all the renaming information) is only available in the CAM.

| Physical Register | Logical Register | Current Mapping | Branch Checkpoint | Free |
|---|---|---|---|---|
| ... | ... | | | |
| P10 | R5 | 1 | 1 | 0 |
| P11 | R6 | 1 | 1 | 0 |
| P12 | R7 | 0→1 | 1 | 0 |
| P13 | R7 | 1→0 | 0 | 0→1 |
| ... | ... | | | |

Fig. 4. CAM mispeculation recovery.

In addition, the checkpoint restoration automatically frees register `p13`, since its entry in the $free$ column is set.

Immediately after recovery, renaming source registers will likely cause RAM misses, because all entries are initially invalid. However, both CAM lookups and new register allocations from the PE will cause the RAM to be progressively updated, hence quickly reducing the probability of subsequent RAM misses.

### A. Implementation

Figure 5 depicts a pipelined implementation of the hybrid scheme, where each box represents a table lookup. Lookups in the RAM are always direct mapped (d.m.), while CAM lookups can be both direct mapped or associative searches. The criterion to separate circuit stages is that no pair of table lookups are performed sequentially in a single stage. This causes our proposal to be pipelined in three stages, though only two of them are on the critical path towards the instruction queues and the ROB.

Advanced circuit layouts may affect the criterion above [3][5] —e.g., by allowing two sequential table lookups in a single stage—, but the physical implementation of the hybrid RAM-CAM is out of the scope of this work. Nevertheless, the access time results obtained from the models devised in section IV-B limit the clock frequency to a reasonable 2Ghz bound. For lower frequencies (e.g., 1Ghz) the number of stages in the critical path can be reduced to 1.

The hybrid scheme representation is horizontally divided into three parts. The lower part shows the source register renaming, while the rest of them deal with destination registers. Specifically, the upper third of the figure illustrates the clearing of previous destination mappings, while the central portion represents the allocation of new mappings. Each of these parts are detailed next.

**Clearing of previous destination mappings.** CAM entries corresponding to the previous destination mappings are cleared. In the first stage, all previous destination mappings are looked up in the RAM. For those RAM entries in a valid state, physical register identifiers are obtained, which are then used in the second stage to directly index the CAM and clear the current mapping entries. In contrast, RAM misses cause previous mappings to be associatively cleared.

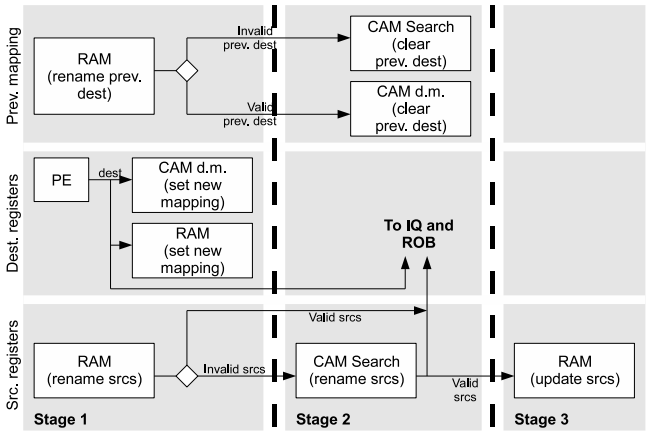



Fig. 5. Hybrid RAM-CAM renaming stages.

**Destination register renaming.** Free physical registers are mapped to destination registers. To this aim, the PE provides physical register identifiers from a set of free entries in the CAM. These identifiers are used to directly index the CAM and set the new mappings. The new mappings are also updated in the RAM, which is indexed with the identifiers of the destination logical registers.

**Source register renaming.** Mapped physical register identifiers are obtained for source registers. In the first stage, the RAM is accessed for valid mappings. On a hit, physical registers are available right away. Otherwise, an associative CAM search is performed in the second stage. Finally, those mappings retrieved from the CAM are updated in the RAM in the third stage to avoid future RAM misses.

Notice that previous mappings are cleared in the CAM in the second stage, while new mappings are set in the first stage. Thus a hazard arises when an associative CAM lookup in the second stage for a given instruction accesses a mapping allocated by the same instruction in the first stage. This hazard can be avoided by flagging the new mapping entries at the end of the first stage in an additional single bit column in the CAM. The flags are reset at the end of the second stage.

## IV. Experimental Evaluation

A performance evaluation has been carried out on top of the SimpleScalar toolset [10], which has been extensively modified to model the hybrid proposal and typical RAM- and CAM-based register renaming approaches. A baseline out-of-order 4-way superscalar processor has been modeled with the architectural parameters summarized in Table II, and results have been obtained from the execution of the entire SPEC2000 benchmark suite including both integer and floating-point benchmarks. For evaluation purposes five schemes have been studied, referred to as follows: $Commit$) RAM-based approach that triggers recovery at commit, $Writeback$) RAM-based approach that triggers recovery at writeback from head to tail, $Writeback\text{-}fwalk$) RAM-based approach that triggers recovery at writeback from tail to head, $Ideal\ CAM$) pure CAM-based approach, and $Hybrid$) proposed approach. In addition, we will apply the suffix -$Iw$ to the $Ideal\ CAM$ and $Hybrid$ schemes that, in order to reduce CAM complexity, constrain to $I$ the number of instructions that can be renamed each cycle.

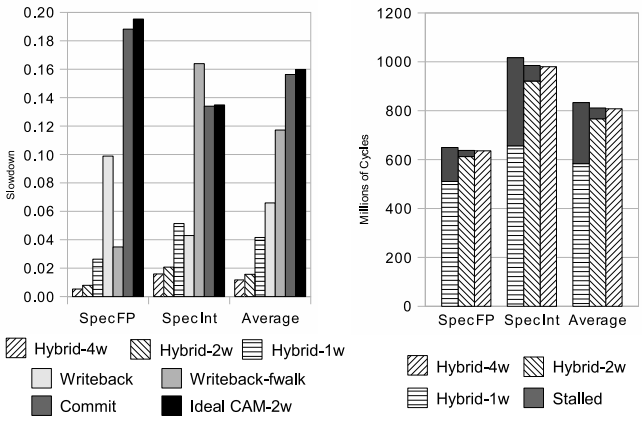
For purely RAM- or CAM-based approaches, register renaming has been assumed to take one pipeline stage. On the contrary, register renaming in the Hybrid proposal is pipelined in three stages, as detailed above. The number of cycles incurred by misprediction recovery has been accurately modeled, taking into account the ROB position of the mispredicted instruction and the number of pipeline stages. Finally, notice that $Ideal\ CAM$ imposes an upper performance bound for the remaining models, since it takes just one processor cycle for both register renaming and misprediction recovery.

### A. Analyzing CAM Requirements

The Hybrid approach associatively searches the CAM only on RAM misses. Therefore, the CAM complexity (which is a major concern for CAM-based approaches) can be sensibly reduced. This section explores the impact on performance of reducing the CAM complexity in the Hybrid approach. Figure 6(a) shows the performance slowdown of the studied renaming schemes with respect to an $Ideal\ CAM\text{-}4w$. For instance, a bar height of 0.2 denotes a scheme performing 20% more slowly than the ideal CAM. Three configurations of the $Hybrid$ approach with different number of CAM ports have been evaluated. Results show that, on average, the hybrid approach slows down the ideal CAM by just about 1.2%, 1.6%, and 4.2% for 4-, 2-, and 1-way CAMs, respectively. The $Hybrid\text{-}4w$ scheme incurs some slowdown because it has a longer pipeline than the ideal CAM. As observed, the other designs perform worse than the hybrid approaches. $Writeback$ and $Writeback\text{-}fwalk$ behave differently for integer and floating-point benchmarks. The reason is that the position of the mispredicted branch inside the ROB is usually farther away from the ROB head for floating-point benchmarks than for integer ones. The $Commit$ design performs worse, on average, since its recovery penalty is usually higher. Finally, for comparison purposes, a 2-way CAM-based design ($Ideal\ CAM\text{-}2w$) is

TABLE II
MACHINE PARAMETERS.

| | |
|---|---|
| **Microprocessor core** | |
| Issue policy | Out of order |
| Fetch, issue, commit bandwidth | 4 instructions/cycle |
| Pipeline depth | 7 stages (RAM- and CAM-based)<br>9 stages (Hybrid) |
| Branch predictor type | gShare/bimodal:<br>Gshare has 16-bit global history<br>plus 64K 2-bit counters.<br>Bimodal has 2K 2-bit counters.<br>Choice predictor has 1K 2-bit counters. |
| # of Integer ALU's, | 4 |
| # of multiplier/dividers | 1 |
| # of FP ALU's | 2 |
| # of FP multiplier/dividers | 1 |
| **Memory hierarchy** | |
| Memory ports available (to CPU) | 2 |
| L1 data cache | 32KB, 4 way, 64 byte-line |
| L1 data cache hit latency | 2 cycles |
| L2 data cache | 512KB, 8 ways, 64 byte-line |
| L2 data cache hit latency | 10 cycles |
| Memory access latency | 100 cycles |




a) Performance slowdown with respect to *Ideal CAM-4w*

b) Execution time with stalled cycles

Fig. 6. Performance of recovery mechanisms.

TABLE III

RELATIVE CAM ACCESSES FOR HYBRID SCHEMES.

| | SpecInt | SpecFP | Average |
|---|---|---|---|
| **Hybrid-1w** | 7.2% | 2.7% | 5% |
| **Hybrid-2w** | 7.6% | 2.6% | 5.1% |
| **Hybrid-4w** | 7.8% | 2.5% | 5.1% |

included in the figure to show the impact on performance of reducing the CAM complexity by blindly halving the renaming bandwidth.

The lower the number of ways in the Hybrid approach the worse performance is achieved. The main reason is that a higher number of ways incurs a higher number of associative searches in the CAM. This metric is represented in Table III as a percentage of the number of searches performed in a pure 4-way CAM. On average, this value lies around 5%. It is especially low for floating-point benchmarks (below 3%), which means that stalls due to lacking CAM ports occur less often. This drastic reduction of associative CAM lookups makes it worth comparing the hybrid scheme with the high-performance, power-hungry pure CAM approach in terms of energy (Section IV-C).

Figure 6(b) presents the total execution time of integer and floating-point benchmarks. The plotted bars include a grey portion that represents the amount of time the rename stage was stalled due to a lack of free ports in the CAM. As observed, this time is higher in integer benchmarks. On average, the total bar heights for $Hybrid$-$2w$ and $Hybrid$-$4w$ are very similar, which means that stalled renaming cycles for $Hybrid$-$2w$ are scarce enough to prevent performance from dropping.

Finally, Figure 7 shows performance results for individual benchmarks. This plot compares the $Commit$, $Writeback$, $Hybrid$-$2w$, and $Ideal\ CAM$-$4w$ schemes. For most benchmarks, an ascendent performance trend can be observed in the mentioned order. In some of them ($facerec$, $fma3d$, or $wupwise$), performance is especially affected by misprediction latencies, and a clear difference shows up between $Commit$ or $Writeback$ designs and $Hybrid$-$2w$ or $Ideal\ CAM$-$4w$ approaches. The latter two provide mostly a similar performance.

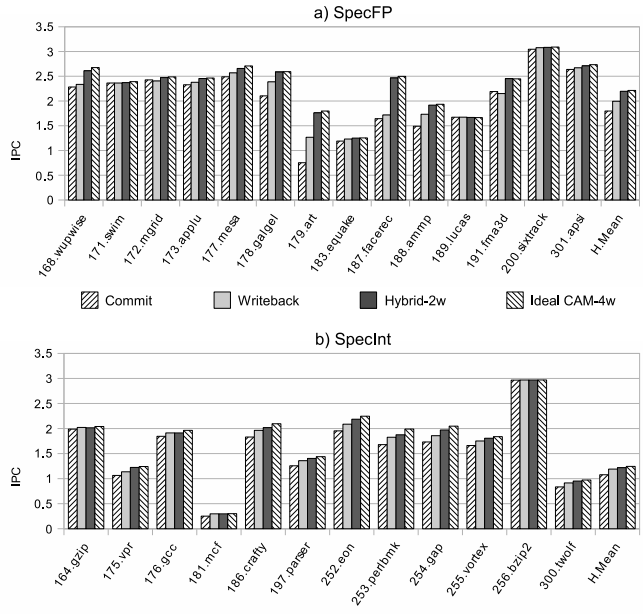



Fig. 7. Detailed IPCs for individual benchmarks.

TABLE IV

HARDWARE COMPLEXITY.

| | Priority Encoder | FRQ | RAM | RRAM | CAM Assoc. | CAM d.m. | Area |
|---|---|---|---|---|---|---|---|
| **Ideal CAM-4w** | yes | – | – | – | 4w+8r | 4w | 0.079 |
| **Commit** | – | 4r+4w | 4rw+8r | 4w | – | – | 0.068 |
| **Writeback** | – | 4r+4w | 4rw+8r | 4w | – | – | 0.068 |
| **Writeback - fwalk** | – | 4r+4w | 4rw+8r | – | – | – | 0.064 |
| **Hybrid-1w** | yes | – | 4rw+8r+2w | – | 1w+2r | 8w | 0.049 |
| **Hybrid-2w** | yes | – | 4rw+8r+4w | – | 2w+4r | 8w | 0.074 |
| **Hybrid-4w** | yes | – | 4rw+8r+8w | – | 4w+8r | 8w | 0.154 |

### B. Hardware Complexity

The hardware used in the evaluated renaming schemes is represented in Table IV, where rows correspond to different models, and columns represent hardware components. In case a component is present in a renaming scheme, a given cell contains the number of read ($r$), write ($w$), and read-write ($rw$) ports it requires for the baseline 4-way processor. For CAM components, the number of direct-mapped (d.m.) ports is also shown. The area values (in $mm^2$) in the last column of the table are obtained by adding the areas of the individual constituting components. Both area and energy results have been calculated by means of the Cacti tool [11] for a 65nm technology.

Let us explore the hybrid design based on the implementation shown in Figure 5 from a complexity point of view. In the first stage, the corresponding RAM table entries are read for previous destination mappings, while the PE allocates new mappings at the same entries ($4rw$ RAM ports). Source operands are renamed by also accessing the RAM ($8r$ RAM ports). Finally, the RAM is updated in the third stage with those source registers involved in previous RAM misses ($2w$, $4w$, and $8w$ RAM ports for $Hybrid$-$1w$, $Hybrid$-$2w$, and $Hybrid$-$4w$, respectively).

Associative CAM ports in the hybrid designs are used in the second stage to rename sources ($2r$, $4r$, or $8r$ CAM ports) and to clear previous destination mappings ($1w$, $2w$, or $4w$ CAM ports) that missed in the RAM. However, if the physical register indexes are correctly provided by the RAM, clearing previous destinations can be performed with a direct-mapped access. Furthermore, d.m. ports are much less complex than associative ports, so different hybrid configurations keep a constant number of them ($4w$ d.m. CAM ports). The remaining $4w$ d.m. ports (overall $8$ d.m. CAM ports) are used in the first stage to allocate the new mappings provided by the PE.

Regarding the ideal CAM design, free physical registers are also provided by the PE. Therefore, as in the hybrid design, the CAM is indexed by means of direct-mapped ports to allocate new destinations ($4w$ d.m. CAM ports). On the other hand, the CAM is associatively searched to rename

source registers and clear previous destination mappings (overall $8r$+$4w$ associative CAM ports).

RAM-based designs (i.e., $Commit$, $Writeback$, and $Writeback$-$fwalk$) use the RAM to rename source registers ($8r$ RAM ports), as well as to look up previous destination mappings and update them with new values ($4rw$ RAM ports). In addition, $Commit$ and $Writeback$ use an RRAM ($4w$ ports) where destination mappings are updated when instructions commit. This table is superfluous for $Writeback$-$fwalk$, since it recovers from the RAM, instead of the RRAM.

Finally, all RAM-based designs use an FRQ to allocate ($4r$ FRQ ports) and release ($4w$ FRQ ports) physical registers. Notice that even if a PE is used instead of an FRQ, the ports are still needed to update the free bit column read by the PE, since unlike checkpointed CAMs, RAMs cannot provide an updated version of the free bit column at any time.

### C. Energy

Figure 8 shows the total dynamic energy budget used for register renaming, and details the fraction spent by accesses to each individual hardware component (FRQ, RAM, RRAM, direct-mapped CAM lookups, and associative CAM searches). Only dynamic energy is shown because the memory structures are small and frequently accessed, so it becomes the dominant fraction of the total spent energy.

The $Ideal\ CAM$ design consumes about one order of magnitude more than the other schemes, since the CAM is associatively accessed for all mappings in every cycle. As also discussed in previous works [12], the main drawback of CAM-based approaches is their high power consumption, which makes $Ideal\ CAM$ stand out from the rest as far as energy is concerned.

Energy dissipation is drastically alleviated by the hybrid designs. Besides providing a performance close to $Ideal\ CAM$, some of their configurations can consume even less than RAM-based designs (e.g., $Hybrid$-$1w$ and $Hybrid$-$2w$ for SpecFP). In general, an increase of the CAM width in hybrid designs causes better performance and higher energy cost. The reason is that the number stalls due to limited CAM bandwidth is reduced, but this also implies a higher number of useless accesses to the CAM when speculative execution takes place, and a higher cost per access due to more costly RAM and CAM structures.

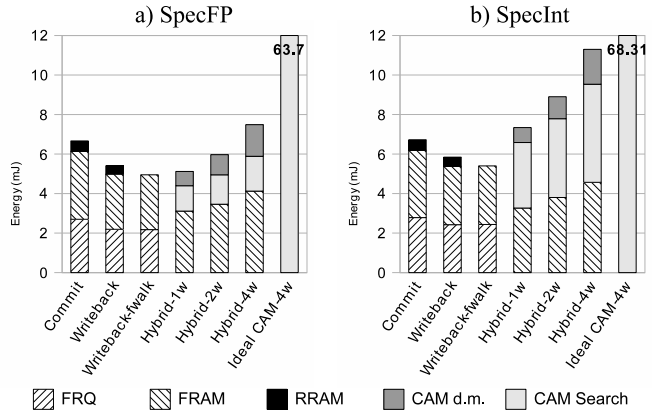



Fig. 8. Dynamic energy spent by each renaming scheme.

As the used tool does not provide with an accurate model of the PE, its energy has not been included in the overall consumption of the CAM designs. However, this limitation does not degrade the comparison —the consumption of the PE is a negligible fraction of the total CAM consumption [13]—, nor it changes the main conclusion of this experiment. Namely, the hybrid designs reduce the power consumption of a purely CAM-based approach by an order of magnitude, closing the gap with the implementations that consume less but perform worse.

## V. Related Work

The complexity of the memory structures (RAM or CAM) used to tackle register renaming in superscalar processors is a major concern because of the large amount of read/write ports required to support the high decode bandwidth. For instance, in a 4-way processor, up to 12 logical register mappings can be looked up in a single clock cycle. Therefore, many research works have aimed at reducing renaming complexity, renaming performance, or both. Existing proposals can be classified in two categories according to the memory structure they use: RAM- and CAM-based approaches.

Regarding the RAM approach, Moshovos [14] proposes to reduce the number of ports in the front-end RAM by detecting those instructions that do not use the maximum number of source and destination register operands. Along the same line, Kucuk et al. [15] further reduce the number of accesses to the front-end RAM by forwarding results of previous accesses performed by nearby instructions.

Some research works have addressed the reduction of the recovery penalty time incurred by RAM-based schemes. In [9], Moshovos proposes an out-of-order release checkpoint mechanism which reduces the number of RAM checkpoints to about one third. In [16], Akl et al. propose a ROB-like structure to accelerate checkpoint recovery. Such structure allows mispeculation recovery from specific branches. Similarly, a selective checkpoint mechanism to recover mispredictions and support large instruction windows is proposed in [17].

In a closely related work [18], Zhou et al. propose a mechanism which enables instructions to be renamed immediately after a misprediction detection, without restoring the front-end RAM table. This RAM-based approach differs from the one proposed in this work in three main aspects. First, it does not deal with register reclamation. Second, this mechanism needs additional logic to correctly manage branch checkpoint queues, instruction queues, and the issue stage. Finally, to obtain correct mappings, it relies on either waiting for the branch instruction to reach the commit stage or scanning the ROB. In contrast, the hybrid RAM-CAM approach proposed in this paper is a simpler mechanism that relies on well known microarchitecture components and does not require any kind of ROB scanning.

Although the RAM approach, which requires one or two renaming tables and a free register queue, has been used in many commercial processors [1][4], the CAM approach has also been used in aggressive designs [3], since it only

requires a single memory structure that supports fast recovery and a large number of checkpoints [5]. Thus, it becomes a major research concern to reduce the access time incurred by CAMs. In this sense, Buti et al. [3] detail how this problem is addressed in the case of the IBM Power4 processor from a technological point of view. In addition, Liu and Lu [19] explore the effect of circuit-level speculation to speed up the response of a CAM renaming table.

In a recent work [5], Safi et al. compare the energy and latency of RAM and CAM approaches. They conclude that, when the number of checkpoints exceeds a given bound, CAM approaches become more efficient and faster. They also propose to selectively disable CAM entries in order to optimize CAM energy consumption.

Finally, although the implementation and goals are quite different and do not affect the novelty of this work, Wallace and Bagherzadeh also propose a hybrid RAM-CAM design [20]. Their goal is to reduce the RAM complexity and access time in RAM-based renaming schemes. To this end, they implement a small ROB-like FIFO queue located before the RAM which is also associatively addressable by a logical register identifier. This table reduces the number of required RAM ports (much like our design reduces the number of required CAM ports, which are more costly), and allows recovery of the correct mappings in one cycle only when mispredicted instructions have not updated the RAM. However, register release, pipelining, and other complexity issues are not tackled. In addition, since this is a proposal focused on reducing RAM complexity, it is perfectly compatible with the present work.

## VI. Conclusions

Current superscalar microprocessors rename registers aggressively by using either a RAM or a CAM-based approach. Each approach presents some advantages and shortcomings. For instance, RAM-based approaches rename registers in a fast and efficient way but they incur a higher performance penalty during recovery. On the contrary, CAM-based approaches have higher access time and energy consumption, but provide fast recovery regardless of the number of checkpoints. Any of these approaches has been implemented in commercial microprocessors, depending on the design goals.

In this paper, we have presented a hybrid renaming mechanism consisting of a RAM table and a low-complexity CAM table. Experimental results show that a 2-way hybrid approach presents small performance slowdowns (about 2% and 1% for integer and floating-point benchmarks, respectively) with respect to a 4-way CAM-based renaming mechanism that is able to recover in one clock cycle. In contrast, a RAM-based renaming mechanism triggering recover at commit presents higher slowdowns than the hybrid approach (about 13% and 19%), while recovering at writeback only reduces the relative slowdowns to 4% and 10%.

The small slowdowns of the hybrid approach are accomplished by performing only 8% and 3% of the original associative searches carried out in the CAM-based approach. This reduces the dynamic energy to 13% and 9%, relative to the original CAM consumption, and it does not affect the occupied area. Thus, the hybrid approach closes the dynamic energy consumption gap between CAM and RAM approaches, and can even consume less than the simplest, not checkpointed RAM approach for some configurations.

## Acknowledgements

This work was supported by Spanish CICYT under Grant TIN2006-15516-C04-01, by Consolider-Ingenio 2010 under Grant CSD2006-00046, by Explora-Ingenio under Grant TIN2008-05338-E, and by Generalitat Valenciana under Grant GV/2009/043.

## References

[1] K. C. Yeager, "The MIPS R10000 Superscalar Microprocessor," *IEEE Micro*, vol. 16, no. 2, pp. 28–40, 1996.

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

[3] T. N. Buti, R. G. McDonald, Z. Khwaja, A. Ambekar, H. Q. Le, W. E. Burky, and B. Williams, "Organization and implementation of the register-renaming mapper for out-of-order IBM POWER4 processors," *IBM Journal of Research and Development*, vol. 49, no. 1, pp. 167–188, 2005.

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

[5] E. Safi, A. Moshovos, and A. Veneris, "A Physical Level Study and Optimization of CAM-based Checkpointed Register Alias Table," in *Proc. of the 13th International Symposium on Low Power Electronics and Design (ISLPED)*, 2008.

[6] J. E. Smith and G. Sohi, "The Microarchitecture of Superscalar Processors," *Proc. of the IEEE*, vol. 83, no. 2, Dec. 1995.

[7] M. Moudgill, K. Pingali, and S. Vassiliadis, "Register Renaming and Dynamic Speculation: an Alternative Approach," in *Proc. of the 26th International Symposium on Microarchitecture*, Dec. 1993.

[8] H. Akkary, R. Rajwar, and S. T. Srinivasan, "Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors," in *Proc. of the 36th International Symposium on Microarchitecture*, Dec. 2003.

[9] A. Moshovos, "Checkpointing Alternatives for High-Performance, Power-Aware Processors," in *Proc. of the International Symposium on Low Power Electronics and Design (ISLPED)*, Aug. 2003.

[10] D. C. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," *Computer Architecture News*, vol. 25, no. 3, 1997.

[11] P. Shivakumar and N. Jouppi, "An Enhanced Access and Cycle Time Model for On-Chip Caches," *DEC WRL technical report number 93/5*, 1994.

[12] K. Pagiamtzis and A. Sheikholeslami, "A Low-Power Content-Addressable Memory (CAM) Using Pipelined Hierarchical Search Scheme," *IEEE Journal of Solid-State Circuits*, vol. 39, pp. 1512–1519, 2004.

[13] B. Agrawal and T. Sherwood, "Modeling tcam power for next generation network devices," in *Proc. of the 2006 IEEE International Symposium on Performance Analysis of Systems and Software*, Mar. 2006, pp. 120–129.

[14] A. Moshovos, "Power-Aware Register Renaming," *Technical Report, Computer Engineering Group, University of Toronto*, 2002.

[15] G. Kucuk, O. Ergin, D. Ponomarev, and K. Ghose, "Reducing Power Dissipation of Register Alias Tables in High-Performance Processors," in *IEEE Proceedings on Computers and Digital Techniques*, Nov. 2005.

[16] P. Akl and A. Moshovos, "Turbo-ROB: A Low Cost Checkpoint/Restore Accelerator," *Lecture Notes in Computer Science*, vol. 4917/2008, pp. 258–272, 2008.

[17] H. Akkary, R. Rajwar, and S. T. Srinivasan, "An Analysis of a Resource Efficient Checkpoint Architecture," *ACM Trans. Archit. Code Optim.*, vol. 1, no. 4, pp. 418–444, 2004.

[18] P. Zhou, S. Önder, and S. Carr, "Fast Branch Misprediction Recovery in Out-of-Order Superscalar Processors," in *Proc. of the 19th International Conference on Supercomputing (ICS)*, 2005.

[19] T. Liu and S. L. Lu, "Performance Improvement with Circuit-Level Speculation," in *Proc. of the 33rd International Symposium on Microarchitecture*, 2000.

[20] S. Wallace and N. Bagherzadeh, "A scalable register file architecture for dynamically scheduled processors," in *Proc. of the 1996 Conference on Parallel Architectures and Compilation Techniques*. Washington, DC, USA: IEEE Computer Society, 1996, p. 179.