Microarchitectural Support for Speculative Register Renaming

Jesús Alastruey[1], Teresa Monreal[1], Víctor Viñals[1], and Mateo Valero[2]

[1]Universidad de Zaragoza
Dept. Informática e Ingeniería de Sistemas-I3A
Zaragoza, Spain
{jalastru, tmonreal, victor}@unizar.es

[2]Universitat Politècnica de Cataluña and BSC
Dept. d'Arquitectura de Computadors
Barcelona, Spain
mateo@ac.upc.edu

Abstract

This paper proposes and evaluates a new microarchitecture for out-of-order processors that supports speculative renaming. We call speculative renaming to the speculative omission of physical register allocation along with the speculative early release of physical registers. These renaming policies may cause a register operand not to be kept in the Physical Register File (PRF). Thus, we add a low-ported Auxiliary Register File (XRF) located outside the processor core that keeps the values absent in PRF and supplies them at higher latency. To support the location of register operands being either in PRF or XRF, we use virtual registers. We consider omission and release policies directed by hardware prediction. Namely, we will use a single Last-Use Predictor that directs both speculative omission and release. We call this mechanism SR-LUP (Speculative Renaming based on Last-Use Prediction). Two Last-Use predictor designs of incremental complexity and performance are analyzed. In a 256-ROB, 8-way processor with an 80int+80fp PRF, SR-LUP with an 11-port 256int+256fp XRF, speeds up computations up to 11.5% and 29% for INT and FP SPEC2K benchmarks, respectively. For FP benchmarks, if the PRF limits the clock frequency, a conventionally managed 128int+128fp PRF can be replaced using SR-LUP by a 64int+64fp PRF backed up with a 10-port 224int+224fp XRF, showing 19% IPS gain.

1. Introduction

Out-of-order superscalar processors working at high clock frequencies often rely on Physical Register Files which merge committed and non-committed versions of logical registers. This is the case of current processors such as Intel Pentium4 or IBM Power5 [14][16].

The Physical Register File (PRF) is located in a critical path and can limit the processor clock frequency [2][11], so several directions have been proposed to reduce its access latency, such as reducing the number of ports [2][5][9][12][24], reducing the number of physical registers [3][4][10][20][22][28], or using a few, narrow entries to encode frequent values [18].

In order to reduce the number of physical registers without losing performance it is required to act on the renaming policy to reduce the time a physical register is allocated. This paper follows this approach and details the design of a microarchitecture that supports speculative renaming, that is, to omit allocation and to perform the release of physical registers in a speculative way.

The baseline to improve is what we call the conventional renaming mechanism. Register renaming is a common technique used to increase the Instruction Level Parallelism in processors that have a centralized Register File [13][14][17][29]. Renaming removes false dependencies by allocating a physical place in the PRF to every instruction writing to a logical register. Therefore, a physical register p serves two functions: p acts as an identifier for tracking dependencies and addressing PRF (p identifier), and also p acts as a value container (p value). Releasing a p identifier under conventional renaming only proceeds after making sure that the p value is not going to be read anymore (p identifier is un-mapped by the next instruction writing to the same logical register, and such instruction reaches the commit stage [23]). This strict condition is easy to track and only involves returning p identifier to a Physical Free List, being the dead p value still retained in PRF until p identifier is allocated to another instruction that writes a new p value, overwriting the previous one. This way, conventional renaming supports precise exceptions and its implementation is simple, but for tight PRFs (# physical registers << # logical registers + # entries in ROB) performance can raise quite a lot if dead values are not retained in PRF [21].

Several improvements have been proposed aimed at relaxing the release conditions in order to recycle a physical register identifier quite before the redefining instruction commits [1][2][3][10][15][19][22]. We can distinguish between safe and speculative policies. In the first group, either software or hardware approaches exist. The former takes advantage of the limited compiler knowledge in order to safely release a register read by its only consumer [15]. Any hardware approach monitors program execution so as to safely release a physical register provided that some conditions are met, such as register dependences, conditional branch outcomes or capability to raise exceptions [19][22].

On the other hand, speculative policies have been proposed to release registers earlier, to the extent that a physical register can be released before all its consumers have read it [1][2][3][10]. This kind of policies has to set the conditions under which a physical register can be released early, and also needs recovery resources to retrieve those incorrectly early released values. The referenced work addresses the first issue either by tracking continuously such conditions [2][3][10] or by predicting which instruc-

1-4244-0910-1/07/$20.00 ©2007 IEEE.

tion uses a given physical register for the last time [1]. For the recovery issue, different back-up structures are used, namely, an Auxiliary Register File [1], a two-level Register File [2] or a Checkpointed Register File [3][10].

Related to the allocation of physical registers, there are also safe and speculative policies. The first one delays the allocation of physical registers until the execution stage [20]. The second one omits the allocation of a physical register under some circumstances: either it is predicted that a destination register will be never read [1][7] or it is predicted that a destination register with a single consumer will be supplied by the bypass network (a short-lived value) [4].

In this work we propose a microarchitecture that supports any kind of speculative renaming. We call speculative renaming (SR) to the omission of physical register allocation and the early release of physical registers. Both policies are speculative because they rely on prediction and so they may require recovery actions in case of a misprediction. Our proposal relies on the management of a low-ported Auxiliary Register File (XRF) located outside the processor critical paths. Instruction results not allocated to physical registers are written directly to XRF, whereas early released registers are transferred from PRF to XRF. To support the location of register operands being either in PRF or in XRF, the dependence tracking is decoupled from physical register identifier through the concept of virtual registers [20].

This SR support is orthogonal with the specific policies that identify the values to be sent to the XRF. As far as we know, this is the first proposal of a microarchitecture that supports any policy of allocation omission and release of physical registers, no matter it comes from compiler hints or from a hardware predictor (the case we evaluate later).

The main microarchitectural modifications and extensions for supporting SR are the following:

Auxiliary Register File. XRF keeps values until it is safe to discard them, therefore supporting precise exceptions as well as control speculation. Both an instruction result lacking physical register allocation and an early released physical register that has been overwritten are supplied from XRF to the instructions reading them (unexpectedly), paying a penalty of some cycles but without restoring it to the PRF.

Dependence tracking. Under SR, operands can be read from PRF or from XRF. To solve this double operand location, the dependence tracking is decoupled from physical register identifiers through the use of virtual registers [20].

Speculative issue. In order to benefit the frequent case, we predict for all instructions having source physical registers that their operands still reside in PRF. In parallel with the PRF access, it is verified that the physical register read has not been prematurely released and then overwritten by a younger instruction reusing it. Source operands without allocated physical registers are read directly from XRF.

Misprediction recovery. When an instruction is reading a physical register and it realizes that such physical register has been overwritten by a younger instruction, the instruction itself and all its already issued dependent instructions have to be squashed and issued again. This is accomplished by means of chained recovery, a selective recovery policy proposed to cope with latency predictions [27].

To evaluate the SR microarchitecture we choose a Last-Use Predictor (LUP) because it can direct at once allocation omission and early release. Other choices are possible, but the potential of this predictor is quite attractive when considering ideal components (oracle LUP, unlimited XRF and free recovery) [1].

In this paper we consider two LUP designs: the "Sticky" LUP (SLUP) predictor proposed in [1] and a new one LUP based on the concept of register Degree-of-Use [8]. SLUP is a simple predictor in which a last-use prediction can not be untrained, that is, once made it "sticks" until the whole prediction entry is replaced. The Degree-of-Use LUP (DULUP) leverages from the work of Butts and Sohi in [8], offering better performance but greater complexity than SLUP.

The paper is structured as follows. Section 2 details the SR microarchitecture and the processor model. Section 3 describes the implementation of the SR-LUP mechanism. Section 4 presents the experimental methodology and analyzes the results. Section 5 comments related research. Finally, we summarize the conclusions in Section 6.

2. SR Microarchitecture

We first present the SR microarchitecture. Section 2.2 describes the processor model. Section 2.3 details our proposal on how to use virtual registers in order to manage PRF and XRF values. Finally, Section 2.4 describes allocation and release of physical, auxiliary and virtual registers.

2.1 SR rationale

Under SR, one or more registers of an instruction can be tagged with speculative renaming predictions. More detailed, source registers can receive an early release of physical register prediction (ERPRpred) and the destination register can receive an omission of physical register allocation prediction (OPRApred). For instance, in Figure 1, let's suppose the predictor tags the physical register p of an instruction I. If p is a source register, after reading its value from PRF, I itself returns the p identifier to the Physical Free List and also copies the p value into the XRF (**A** circuit in Figure 1). If p is a destination register, I is going to produce a dead value, so now the p value is written directly into the XRF (**B** circuit in Figure 1). Besides, in this case it is not needed to waste a p identifier for the logical register destination of I, being sufficient the allocation of an auxiliary register (x id in Figure 1).

A misprediction or a late read due to out-of-order execution may require a value without an allocated physical register or a prematurely released value. Unexpected uses of speculatively omitted registers will read their operands from XRF, whereas unexpected uses of speculatively released registers will read the required values either from PRF (if the value has not yet been overwritten) or from XRF (**C** circuit in Figure 1). Virtual registers will help us in dealing with this double value location [20] (Section 2.3).

2.2 Processor Model

We model a superscalar processor with a ten-stage pipeline (see top of Figure 2). It supports back-to-back execution, so a dependent instruction is speculatively woken up based on the issue of their parent instructions. Concerning the memory model, a load is not issued until all previous

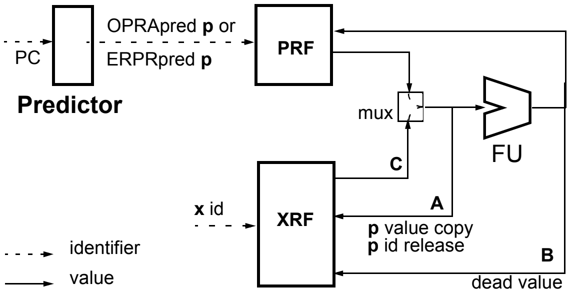

Figure 1. Basic components and circuits required in SR microarchitecture.

stores have computed their addresses and there is no address match. Besides, load instructions are issued with the blind latency prediction of L1 data cache hit, so every load remains in the Issue Queue until the hit is verified. In case of a cache miss, the load and its issued dependent instructions are squashed by means of a simple chained recovery mechanism used in the context of cache bank prediction [27]. It is selective (only its dependent instructions are reissued) and starts recovery as soon as the misprediction notification reaches the IQ. The recovery is called chained because every issue cycle all the instructions to be cancelled are squashed except the ones that are being selected.

To cope with misspeculations, SR leans on the same recovery approach. The IQ schedules dependent instructions assuming all source register operands lie in PRF. If any source value is not in PRF, the offending instruction has to be reissued and all its direct dependents re-scheduled at the higher latency of accessing XRF. If one source operand lies in XRF and the other in PRF, the functional unit is reserved from the time the first value arrives from PRF until the second value arrives from XRF.

2.3 PRF and XRF Register Value Management

Let´s consider two instructions A and B, A executing before B, whose logical destination registers will become consecutively allocated to the same physical register pd. Let's also assume a third intervening instruction between A and B, I, that speculatively early releases pd. After instruction A executes, the computed value is only present in the pd register of PRF, but after the Register Read stage of I, the A value is also in XRF. The A value remains in PRF until instruction B completes execution. After the overwritten moment, the pd register in PRF stores the B value, whereas A value is again only present in a single place, the XRF. So, under SR, an instruction result can be either only in PRF, both in PRF and XRF or only in XRF.

This SR feature makes physical identifiers unable to locate register operands. We suggest to overcome this problem by using the concept of virtual registers, previously proposed in the context of physical registers allocation [20].

A pair of identifiers is allocated to every logical destination register: a physical register and a virtual register, let's call them the <p, v> pair. Figure 2 presents the structures implementing the SR circuit. The new double mapping is supported by a number of mapping tables that manage speculative renaming and provide a way to locate values. The Global Map Table (GMT) supports the double mapping. A Physical to Virtual (P2V) Table, a Virtual to Auxiliary (V2X) Table, a match structure (set of comparators), and a mux selecting between PRF and XRF (belonging to the bypass network) allow value location. The P2V Table is accessed in parallel with PRF and keeps the current virtual mapping associated to a given physical register. This P2V Table and the match structure verify whether the value read from PRF is the correct one or not. The V2X Table acts as an Auxiliary Map Table and stores the mapping between a virtual identifier and the auxiliary register that holds a value with no physical register allocated or an early released value. The Virtual and Auxiliary Free Lists manage virtual and auxiliary free identifiers, respectively.

For every instruction, its predictions (OPRApred and/or ERPRpred) are stored into the RAM Payload. In the Rename stage, an instruction reads from the GMT the <p, v> pair corresponding to its source registers. Also in the Rename stage, the GMT is updated with a new pair of identifiers <p, v> that map the logical destination register. An OPRApred set in the destination register forces the allocation of only a virtual identifier, so no physical register identifier is reclaimed. For every source and destination register coming with a prediction, an auxiliary register x from the Auxiliary Free List is reclaimed. The V2X Table stores such mappings (virtual to auxiliary register identifiers). After that, instructions are dispatched to the IQ and all the relevant information is stored into the RAM Payload (opcode, physical registers, predictions, etc.). Under SR, dependence tracking in IQ is managed by virtual identifiers.

At the same time a value is written into PRF (register p), its virtual identifier is stored in the P2V Table (P2V[p]=v). Concerning to speculative releasing, an instruction with some source register tagged as ERPRpred performs three actions in the Register Read stage: i) reads the register value from PRF, ii) writes it into XRF, and iii) releases the register identifier to the Physical Free List. In case of a destination register tagged as OPRApred, the ALU or cache output writes it directly into XRF.

Under SR, except for operands with no physical register allocated, source register locations are not known at issue time, so every selected instruction uses the <p, v> pair to locate their operands. The p identifier is used to access in parallel PRF and P2V Table, looking for the current virtual mapping of p (v_current=P2V[p]). The v identifier is compared with the v_current mapping; if they match, the p value from PRF is the right one, but when the virtual check fails, the operand should be read from XRF and recovery actions undertaken. Reading XRF requires an access to V2X Table to find the auxiliary identifier that contains the saved value. As regards recovery actions, the offending instruction is reissued and its dependent instructions rescheduled to the XRF latency (Section 2.2). Note that recovery is only needed after the speculative early release of a register p whose p value has been overwritten in PRF by the instruction that reallocated the released p identifier.

2.4 Virtual, Physical and Auxiliary Register Releasing

Under SR up to three kinds of register identifiers have to return to three different Free Lists (Virtual, Physical and

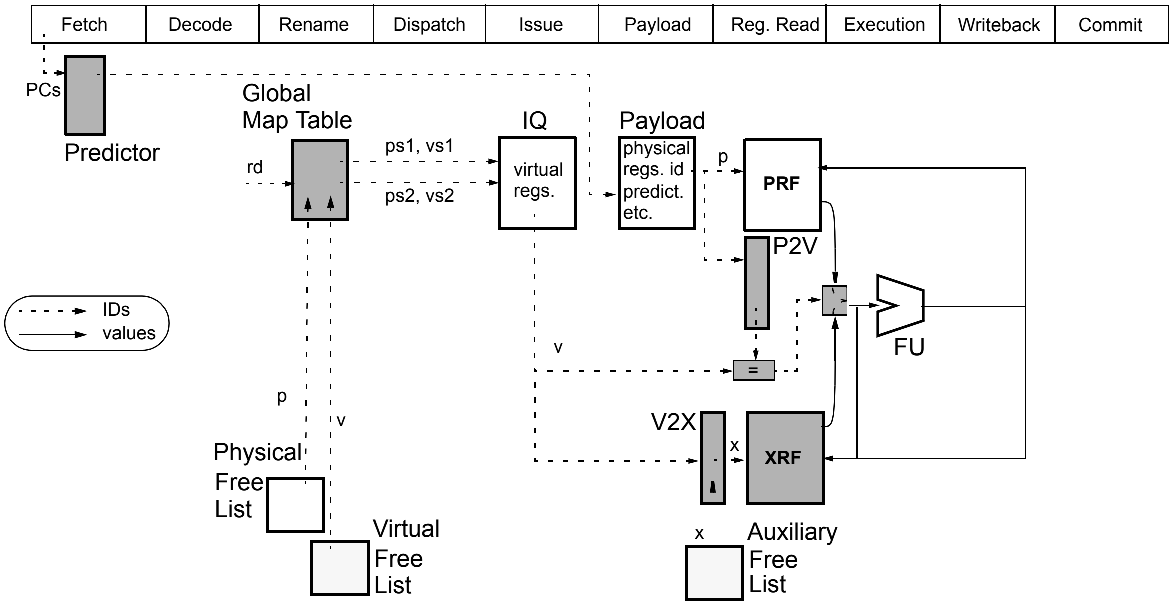

Figure 2. SR microarchitecture. Hardware components (dark grey) are placed below their corresponding stage in the pipeline.

Auxiliary Free Lists in Figure 2). Both virtual and auxiliary identifiers are managed in conventional way, being released at the Commit stage of redefining instructions. On the other hand, a physical identifier experiences either an early or a conventional release. In order to distinguish the kind of release two bit-vectors are needed. These bit-vectors have several functionalities: permitting the conventional release of physical registers not tagged with ERPR predictions, controlling that early released registers will not be released twice and canceling ERPR predictions to registers yet previously tagged.

We call Speculative Renaming Prediction List (SRP) to the bit-vector that informs about what values have received an OPRA or ERPR prediction. The SRP List is indexed by virtual identifier and it is set in the Rename stage for every received OPRA or ERPR prediction. So, for every instruction, a bit set in the SRP entry for its destination register cancels the conventional release of the physical identifier and activates the conventional release of the auxiliary one. Besides, a SRP bit set in a source register inhibits new ERPR predictions received by that register. The SRP bit is unset when a virtual register is allocated. As for GMT, an SRP copy has to be done after each predicted branch.

The Committed Early Release List (CER) is a bit-vector used to perform the SRP recovery. The CER informs whether any speculative release scheduled during a mispredicted path has finished or not (the value is in the XRF or not). The CER List is also indexed by virtual identifier. An entry is set when the speculatively released value related to the virtual identifier is written in the XRF. With this, only such SRP entries that are not set in the CER List are recovered in case of a branch misprediction. Any SRP bit recovered also implies the conventional release of the auxiliary identifier and the CER bit unsetting.

3. An SR-LUP Mechanism Design

In SR-LUP, the Last-Use predictor (LU predictor) directs both speculative omission and early release of physical registers. We use LU predictors indexed with program counter whose entries have partial tags. For every matching instruction, the LU predictor predicts a pattern of use, a bit-vector telling for each register operand whether further reads, in program order, are going to appear or not. Accordingly, when a register operand has a prediction of no further use we call it a Last-Use prediction (LUpred).

SR-LUP takes advantage of a Last-Use prediction on a source register (ERPRpred) by performing an early release just after reading it. A Last-Use prediction on a destination register (OPRApred) is processed earlier, in the rename stage, by omitting the allocation of a physical register.

Two LU predictors of incremental complexity and performance are used in this work. The first one is the Sticky LU predictor (SLUP) suggested by Alastruey et al. in [1]. We propose a second predictor based on the degree-of-use concept introduced by Butts and Sohi [8] (we call it Degree of Use LU predictor, DULUP). Both LU predictors benefit from enhancements such as replacing tags by branch-aware signatures or adding confidence bits, but in order to keep complexity low in the prediction part, we will use the following simplified designs.

3.1 SLUP: Sticky Last-Use Predictor

The SLUP consists in the predictor itself and an In-Order Last-Use table (IO-LUT) -see Figure 3-. The predictor is organized as an associative cache with partial tags. Each entry accumulates the last-use operands observed for an instruction over all past executions (LUpred bits s1, s2, and d), and untraining is only possible by means of entry replacement. The predicted pattern of use flows across the pipeline, filling the RAM payload when the instruction is

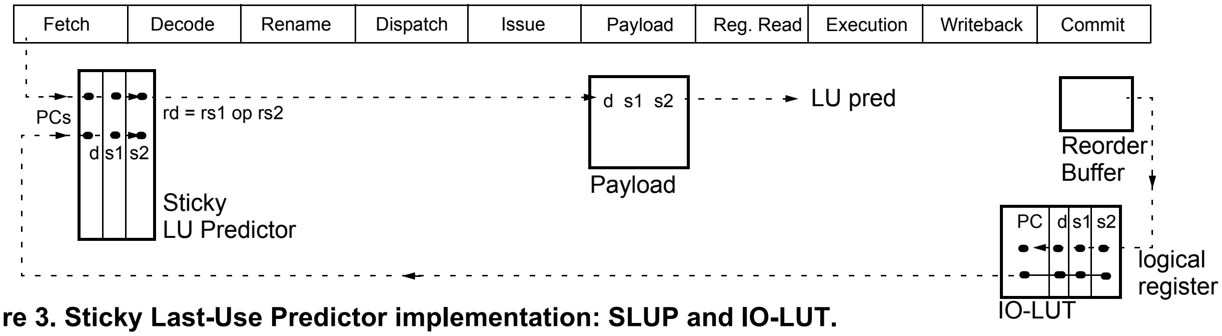

Figure 3. Sticky Last-Use Predictor implementation: SLUP and IO-LUT.

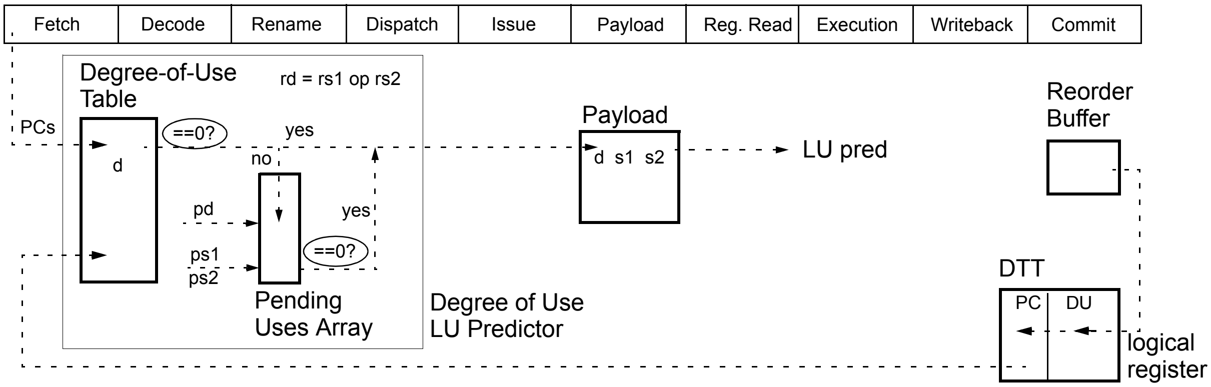

Figure 4. Degree of Use Last-Use Predictor implementation: Degree-of-Use Table, Pending Uses Array and DTT.

dispatched. An LUpred bit set for any source register drives the early release of that register. A destination register with the LUpred bit set forces the Rename stage not to allocate a physical register and the Writeback stage to write only into XRF. Whenever a prediction is not available for an instruction, all register values are assumed alive.

The IO-LUT keeps the last use, in program order, experienced by every logical register. So, each IO-LUT entry has the program counter and the operand type (destination, source 1 or 2) of the last instruction referencing a given logical register. Every time an instruction having a destination register commits, a previous instruction becomes the last-use of such destination register. By reading the IO-LUT entry corresponding to the destination register, the PC of the last-use instruction is used to train the SLUP, setting the proper LUpred bit in the pattern of use. If the last-use instruction was already in the predictor, this action does not change the remaining LUpred bits, being this the reason that once a prediction is made it “sticks” during the whole entry lifetime.

3.2 DULUP: Degree of Use Last-Use Predictor

DULUP derives from the degree of use predictor proposed in [8]. The predictor starts with a Degree-of-Use Table (DUT) organized as an associative cache with partial tags -see Figure 4-. For an instruction with a destination register, an entry in the DUT has the expected number of readings such a destination register is going to experience. Every time a new instance of the instruction commits, a new training is performed, overwriting the previous one.

By looking up DUT in parallel with Fetch, it can be known for the found instructions the expected number of uses their destination registers have, storing next such numbers in the Pending Uses Array (PUA), which is indexed by physical register (pd in Figure 4). From now on, every time an instruction is renamed, a PUA counter is decremented for each one of its physical source registers (ps1 and ps2 in Figure 4). After that, a source register achieving a zero-value counter in PUA receives an LUpred, and the corresponding instruction will proceed to an speculative early release. Note that decrementing PUA counters is done at the Rename stage so an accurate management requires value restore after a branch misprediction. As before, a zero degree-of-use set for a destination register tells the Rename stage to omit physical register allocation and the Writeback stage to write only into the XRF.

Training DUT is accomplished by means of a Dynamic Training Table (DTT) that tracks the number of committed readings performed to every logical register.

4. Experimental Results and Analysis

First, we describe the experimental framework and the methodology used. Section 4.2 analyzes the performance of several configurations of SLUP and DULUP predictors. Next, we analyze the performance of SR microarchitecture directed by the two LU predictors.

4.1 Simulation Environment and Methodology

In all experiments we have used a detailed cycle-based simulator derived from SimpleScalar v3.0 [6] to model the

Table 1. Processor parameters.

Parameter	Value	Parameter	Value
Fetch, decode and rename width	8 instructions. Up to 2 taken branches fetched	L1 I-cache	64 KB, 4-way set-associative 32-byte line size, 1-cycle hit time
Branch prediction	Hybrid predictor: bimodal + gshare 16-bits with speculative update	L1 D-cache	32 KB, 2-way set-associative 64-byte line size, 2-cycle hit time
Functional Units (*latency*)	8 simple int (*1*), 2 mult int (*7*), 4 load/store, 4 simple FP (*4*), 4 FP mult/div (*4/16*)	L2 U-Cache	256 KB, 8-way set-associative 128-byte line size, 10-cycle hit time
Issue Queue size	int+mem: 64 entries; fp: 32 entries	L3 U-Cache	4 MB, 4-way set-associative 128-byte line size, 16-cycle hit time
Reorder Buffer	256 entries	Main Memory	unbounded, 200-cycle access time
Load/Store Queue	128 entries, store-load forwarding	TLB	256 entries, 4-way set-associative
Issue width	8int + 4fp	PRF	36-288 int / 36-288 fp (32 int / 32 fp logical)
Commit width	8 instructions	XRF	32-288 int / 32-288 fp, 3-cycle access time

processor described in Section 2.2. The main parameters of the simulated microarchitecture are given in Table 1. As workload we use all the benchmarks of SPEC2000. We use the Alpha binaries compiled by C. Weaver and simulate a contiguous run of 100M instructions from the points suggested by Sherwood et al. [26].

4.2 The LU predictor design

In this section we change the size and associativity of SLUP and DULUP, and analyze the variation of two metrics that correlate with performance: last-use mispredictions (incorrect last-use register identifications that produce Unexpected Uses, UUs) and opportunity loss (last-use registers that have not been identified). Figure 5 shows these metrics for several sizes and associativities of both predictors.

As can be seen, the opportunity loss rate decreases as sizes increase in both predictors. When capacity increases, conflicts decrease and therefore there are more available last-use predictions. On the other hand, last-use mispredictions increase with predictor sizes: the more entries the predictor has, the more last-use predictions will be made, and so more last-use mispredictions. These two effects are more noticeable for SLUP. As expected, associativity increasing behaves as size increasing: opportunity losses decrease and last-use mispredictions increase. This effect is more noticeable for integer benchmarks.

If we compare both predictors, we can observe that SLUP suffers more last-use mispredictions than DULUP for all predictor configurations. This is because SLUP keeps last-use predictions to an instruction having a last-use register in any of its past executions, even when in its very last execution its pattern of use does not show any last-use register. In contrast, DULUP due to its different update and prediction mechanism, is able to catch changes in the pattern of use. Concerning opportunity losses, they are less frequent for DULUP, specially with the smaller predictors. In spite of more instructions entering DULUP (68% of dynamic instructions have destination register vs. 58% that have a last-use register), the SLUP simple training and update policy makes it less efficient managing its entries.

In the following sections we will use direct-mapped, 5-bit tagged 1k-entry LU predictors. For DULUP, the Degree-of-Use Table and the Pending Uses Array contain 3-bit counters, and the latter will not be restored when recovering from a branch misprediction. We choose these simple configurations (hardware budgets of around 1 KB for the predictors and 0.1 KB for their training mechanisms) as a good point in the complexity/performance trade-off. As expected from Figure 5, performance (IPC) increases with larger predictors (up to 4k entries, 4-way set-associativity for SLUP and 2k-entries, 2-way set-associativity for DULUP). We have not included these experiments by lack of space.

4.3 Performance sensitivity to the Auxiliary Register File design

The goal of this section is to find a realistic design of XRF that provides good performance without compromising the processor complexity. We have considered an XRF with an access time of one cycle and two more cycles to drive the data to and from it. Therefore, we have rejected any XRF design (ports, size) exceeding the access time of the considered PRF. Next, we detail the effects of restricting the number of entries and ports of XRF and evaluate several XRF configurations in order to find XRFs that exploit well SR-LUP.

XRF size. As stated in before, at the Rename stage there should be supplied as much auxiliary registers as the total number of LUpred registers identified by the LU predictor. The lack of auxiliary registers supposes an LUpred cancellation, and possibly a loss opportunity of omitting a physical register allocation or an early register release. To exploit at maximum the potential of SR-LUP, the number of auxiliary registers must be enough to make the number of LUpred cancellations negligible. The maximum theoretical XRF size for our processor is 288 entries, close to the maximum observed (274 for XRFint and 270 for XRFfp).

Number of ports in XRF. The XRF write ports allow the copy of the LUpred register values. So, their number is determined by the number of source LUpred registers (instructions in the Register Read stage) and destination LUpred registers (instructions in the Writeback stage). Lack of write ports causes contention when selecting in IQ instructions with LUpred registers, and therefore, the delay

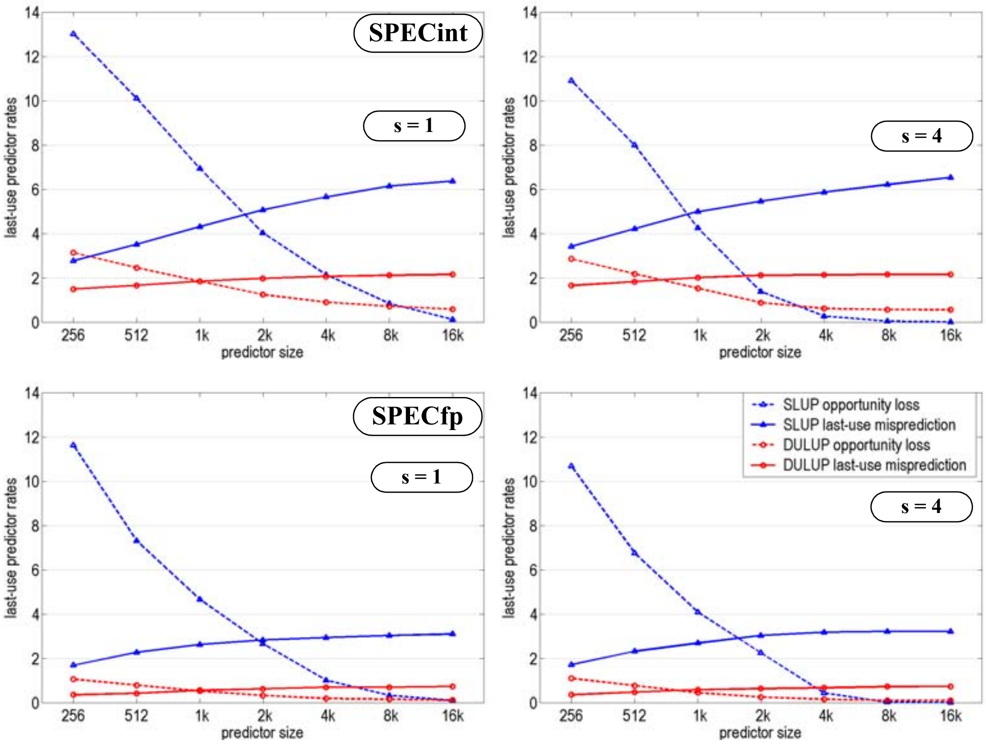

Figure 5. Last-use predictor rates for several sizes and associativities of SLUP and DULUP predictors. The two figures on the left/right correspond to direct-mapped (s=1), and 4-way set associative (s=4) predictors.The two figures above/below show last-use predictor rates for int/fp benchmarks. Last-use prediction rates are split into opportunity loss (dashed lines) and last-use mispredictions (solid lines). An 80int+80fp PRF and an unbounded XRF have been used.

of the scheduled speculative omission and release of physical registers (up to three predictions in the same instruction). To avoid this contention, the number of write ports should not be a limiting factor (a maximum of 30 write ports in XRFint, 16+14 for sources and destinations, respectively).

The XRF read ports supply values not kept in the PRF to the functional units when executing UU instructions. The number of needed ports comes from the amount of register reads from the XRF due to values that have not been provided by the PRF and omission of physical register allocation mispredictions that have not been provided by the bypass network. Lack of read ports involves contention when issuing UU instructions, so delaying the advance of this kind of instructions (additionally penalized with two cycles due to the XRF access). To avoid this contention, the number of read ports should allow all possible UUs to read simultaneously from XRF (a maximum of 16 ports for XRFint).

According our experiments with 1k LU predictors, the real use of the read and write ports observed suggests that their number may be significantly reduced with negligible contention rates.

XRF size and number of ports trade-offs. Summarizing, we need an XRF with enough entries not to cancel LU predictions, and with enough ports to write LUpred registers and to read UU registers. To explore a reasonable amount of design points, we have proceeded by selecting several XRF sizes, and several combinations of read and write ports with similar UU and LUpred issue contention rates. Then we have mixed together sizes and ports to obtain different XRF configurations. For each PRF size, we have discarded the XRFs with more access time than the considered PRF. As a result, we obtain different XRF configurations for each PRF size (from highly-ported XRFs with few entries to low-ported XRF with many entries). In order to reduce the experimentation points, for each XRF size we have chosen the most highly-ported, i.e. XRFs performing the most (likewise, for each number of ports, we have chosen the largest XRF). We have considered XRFint and XRFfp biased to the most limiting one (the XRFint). Finally, we have evaluated the obtained XRFs configurations in order to find the best performing one.

For each PRF size, Table 2 shows the XRF configurations that obtain the best performance (IPC) among all the selected samples. Table 2 stops at PRFs of sizes greater than 160 because larger PRFs do not benefit from SR-LUP. The number of write ports have been broken down into source and destination ports (s, d).

We observe that the number of XRF ports has been proportionally more reduced than the number of entries (for the XRF with 160-entries and 7-ports, almost an 85% reduction in the number of ports, and less than 45% entries). From

Table 2. XRFint and XRFfp biased configurations that best adapt to SR-LUP (PRF sizes bigger than 160 entries are not shown because they do no benefit from SR-LUP).

	Physical PRF							
	36	**40**	**48**	**64**	**80**	**96**	**128**	**160**
XRF size	160	160	192	224	256	256	256	256
XRF total ports	7	8	8	10	11	14	18	22
XRF read ports	2	2	2	3	3	5	6	8
XRF write ports (source+destination)	5 (4s+1d)	6 (5s+1d)	6 (5s+1d)	7 (5s+2d)	8 (6s+2d)	9 (7s+2d)	12 (10s+2d)	14 (12s+2d)

these results and with the performance goal in mind, it is preferable to delay the issue of either UU or LUpred instructions rather than to cancel last-use predictions. Although the read and write contention avoids the advance of UUs and LUpreds, the issue stage is not blocked, since other instructions can advance. Opposite from this, the cancellation of an LUpred due to lack of XRF registers may imply a lost opportunity of early release. With tight PRFs, this may produce a rename stall due to lack of physical registers, what negatively impacts performance. For the smallest PRFs (up to 80 registers), it can be observed that 2 or 3 read ports are enough to UU reads, whereas LUpreds (both source and destination) can be written with only 5 to 8 ports. These read and write ports numbers are far from the theoretically needed to avoid contention (a maximum of 16 and 30 ports for XRFint).

4.4 SR-LUP performance (IPC and IPS)

Figure 6 shows for every PRF size the IPC harmonic mean obtained by the XRFs listed in Table 2 for both SLUP and DULUP predictors. Conventional (conv) and SR-LUP with an oracle-based predictor (SR-OB) are shown as the lower and upper performance limits, respectively. We can observe how the benefits of SR-LUP for floating point codes are much more significant than for the integer ones. This is an expected result since FP programs in general cause a much higher register pressure. SR-LUP significantly outperforms the conventional renaming and reaches gains close to the limits. For tight PRFs of 36 and 40 registers and any kind of benchmark, performance is more than doubled for both SR-SLUP and SR-DULUP configurations.

A combination of an 80-entry PRF and a 256-entry XRF of 11 ports with SR-SLUP obtains speedups of 8.6% and 25.3% for INT and FP benchmarks, respectively. The same figures with SR-DULUP are of 11.5% and 29.0%. With PRF sizes larger than 128/160 registers for INT/FP benchmarks and any SR-LUP configurations, speedups decrease up to be negligible or even negative. As regards the limits, SR-DULUP supported by an 80-entry PRF and a 256-entry XRF of 11 ports reaches 98% and 96% of the SR-OB IPC for integer and floating point benchmarks, respectively. Besides, it reaches 97% and 96% of the IPC obtainable with unlimited number of physical registers.

When the processor cycle time is constrained by PRF access time, a reduction in the PRF size may lead to a clock frequency increase and to an added performance improvement. Figure 7 shows the harmonic mean of the instructions per second (IPS) executed by processors working with the four configurations under study (conv, SR-SLUP, SR-DULUP and SR-OB, all supported by the XRFs listed in Table 2). Cycle times have been obtained according to the 0.18μ Rixner model [25].

For integer benchmarks, the best performance for conventional renaming is obtained with a 96-entry PRF. Applying SR-SLUP/SR-DULUP and an 192-entry XRF of 8 ports, PRF size is reduced to 48 also achieving almost the same IPC and 13%/14% of IPS increase. For FP benchmarks, the best performance for conventional renaming is now obtained with a 128-entry PRF. A combination of a 64-entry PRF supported by a 224-entry XRF of 10 ports reports about the same IPC and IPS gains of 16%/19% (SR-SLUP/SR-DULUP). The additional structures (extra registers, tables and bit vectors) needed to support these SR configurations require around 4 KB (3.80 KB for 48PRF+192XRF and 4.14 KB for 64PRF+224XRF), a reasonable hardware cost for current processors.

5. Related Work

The potential of an ideal last-use prediction was studied in [1], but this work lacked the design of any real implementation. Unlike that approach, our paper shows up the main aspects about the design of an entire SR microarchitecture and proposes feasible and real SR-LUP designs that reach gains close to the limits.

Borch and Tune in [5] propose a way of reducing the issue-to-execute latency in the pipeline. Their Distributed Register Algorithm (DRA) moves the entire PRF out of the issue-to-execute path and places it in the decode-issue path. Instead of that approach, our SR-LUP design relies on the movement of dead values to a non critical and farther XRF without compromising any other processor critical paths.

As regards other early register release proposals, either speculative [2][3][10] or not [19][22], all of them still check the un-mapping condition used in the conventional release conditions. Besides, to identify dead values, all these proposals use array-counters that keep track of the pending consumers for every physical register. For the mechanisms leading to correct program execution, those counters must be restored when recovering from a branch misprediction. Under SR-LUP, there is no need to wait for the renaming instruction in order to identify a last-use instruction. Even more, and different from the other techniques, physical registers can be released far before their unmapping instructions are fetched. In the SR-SLUP design, both the hardware to check the unmapped register condition and the array counters are replaced by a simpler LU predictor. For

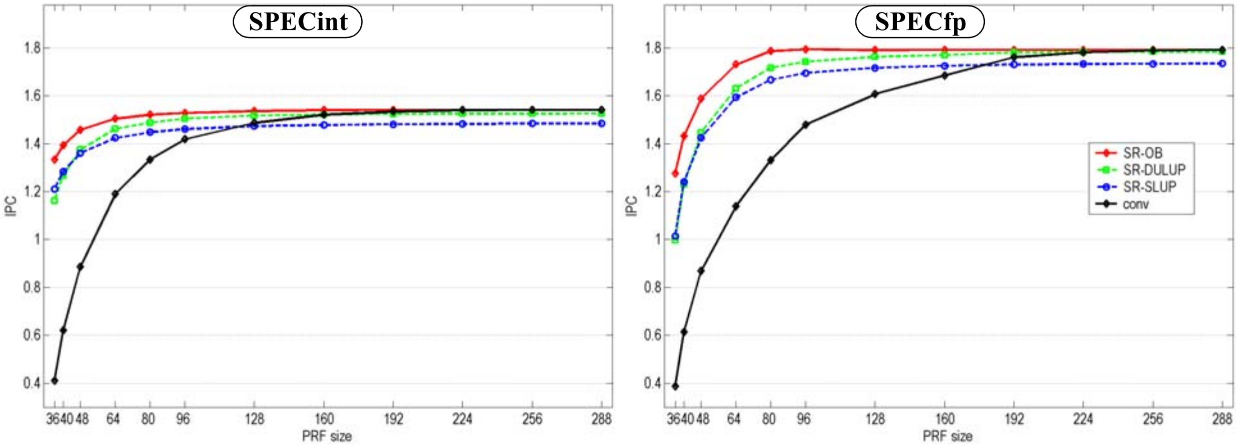

Figure 6. IPC harmonic mean vs. PRF size for conventional, SR-SLUP, SR-DULUP and SR-OB configurations. For every PRF, the corresponding XRF dimensions as shown in Table 2.

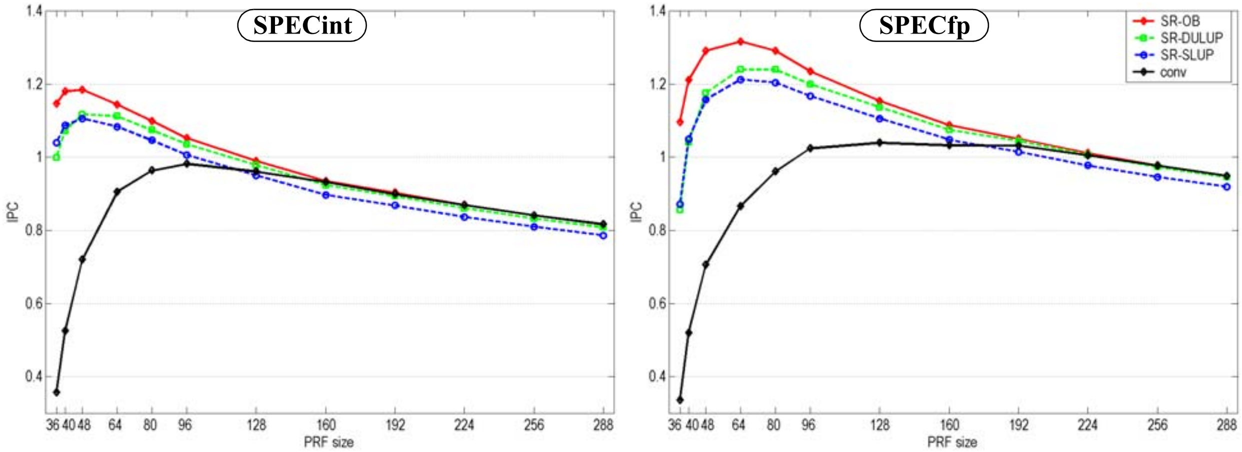

Figure 7. IPS harmonic mean vs. PRF size for conventional, SR-SLUP, SR-DULUP and SR-OB configurations. For every PRF, the corresponding XRF dimensions are shown in Table 2.

the SR-DULUP design, the check hardware does not exist either, and the mechanism can execute a program correctly without restoring the pending uses array.

Related on the speculative proposals, Balasubramonian et al. change the PRF organization by a two-level PRF whose second-level (L2) acts as a container for the detected dead values [2]. Ergin et al. [10] and Barkan et al. [3] use an entire copy of the PRF (the Checkpointed Register File, CRF) to keep early released values. Different from the SR virtual register management, both proposals return all the incorrectly released values back to the PRF, even if they are not going to be read in the future. Besides, with CRF only two instances of a physical register can be alive at the same time. Even more, the CRF size is constrained to the PRF size. As we have analyzed in Section 4.3, restrictions in the back-up structure size seriously damage performance due to the cancellation of predictions.

With respect to register allocation, Butts and Sohi suggest not allocating a physical register to the instructions predicted to produce dead values, and even not executing such dead instructions [7]. The predictor used is a precursor of the DUP [8] that we have used for developing DULUP. More recently, Balkan et al. do not allocate a physical register to a short-lived value with only one consumer that is predicted to read it from the bypass network [4]. Another restrictive condition is the absence of branches between the producer of the value and the instruction redefining its corresponding logical register. Because those values are dropped in case of correct predictions, this mechanism requires additional hardware to support precise exceptions and interrupts. To relax such requirements, this policy could take advantage of our SR microarchitectural support. Besides, it could be combined with other policies such as LUP.

6. Conclusions

This paper presents a microarchitectural design that supports speculative renaming, proposing all the structures and control required to exploit such renaming alternative. Under SR, register values are available either from a low-ported XRF located outside the processor core or from PRF even many cycles after their register identifiers have been released. A double mapping of physical and virtual register identifiers resolves the correct location of register operands without charging the IQ complexity.

A Last-Use predictor that implements both speculative omission and release of physical registers is used to evaluate the SR design, but it is also open to other predictive renaming policies that could be proposed, either hardware or software.

The microarchitecture has been analyzed with two different LU predictors of incremental complexity and performance. With the simple predictor, a 256-ROB processor with a 80int+80fp PRF shows a speed-up of 8.6% and 25.3% for integer and floating point benchmarks, respectively, when it is enhanced with an SR-LUP with a low-ported 256int+256fp XRF. The former figures translate into 11.5% and 29.0% with the best predictor. As regards the limits and with the same figures and the best predictor, a 98% of the SR-OB IPC is reached for integer benchmarks. Also with the best predictor and for FP benchmarks, a conventionally managed 128int+128fp PRF can be replaced in a processor with SR-LUP by a 64int+64fp PRF backed up with a low-ported 224int+224fp XRF, showing 1.2% IPC gain. If PRF is limiting clock frequency, the former figures translate into 16% and 19% IPS gain for the simple and complex predictors, respectively.

7. Acknowledgments

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

8. References

[1] J. Alastruey, T. Monreal, V. Viñals and M. Valero, “Speculative Early Register Release”. *ACM Int’l Conference on Computing Frontiers (ICCF 06)*, May 2006, pp. 291-302.

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

[3] D. Balkan, J. Sharkey, D. Ponomarev and A. Aggarwal, “Address-Value Decoupling for Early Register Deallocation”, *Proc. 35th Int’l Conf. on Parallel Processing (ICPP-06)*, Aug. 2006, pp. 337-346.

[4] D. Balkan, J. Sharkey, D. Ponomarev and K. Ghose, “SPARTAN: Speculative Avoidance of Register Allocation to Transient Values for Performance and Energy Efficiency”, *Proc. 15th Int’l Conf. Parallel Architectures and Compilation Techniques (PACT 06)*, Sept. 2006, pp. 265-274.

[5] E. Borch, E. Tune, S. Manne and J. Emer, “Loose Loops Sink Chips”, *Proc. 8th Int’l Symp. High-Performance Computer Architecture (HPCA 02)*, Feb. 2002, pp. 299-310.

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

[7] J.A. Butts and G. Sohi, “Dynamic dead-instruction detection and elimination”, *Proc. 10th Int’l Conf. Architectural Support for Programing Languages and Operating Systems (ASPLOS 02)*, Oct. 2002, pp. 199-210.

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

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

[10] O. Ergin, D. Balkan, D. Ponomarev and K. Ghose, “Early Register Deallocation Mechanisms Using Checkpointed Register Files”, *IEEE Transactions on Computers*, vol. 55, no. 9, Sept. 2006, pp. 1153-1166.

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

[12] K. Farkas, P. Chow, N. Jouppi and Z. Vranesic, "The Multicluster Architecture: Reducing Cycle Time Through Partitioning”, *Proc. 30th Int’l Symp. Microarchitecture (MICRO 97)*, Dec. 1997, pp. 149-159.

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

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

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

[16] R. Kalla, B. Sinharoy, and J.M. Tendler, “IBM Power5 chip: a dual-core multithreaded processor,” *IEEE Micro*, vol. 24, no. 2, Mar.-Apr. 2004, pp. 40-47.

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

[18] M.H. Lipasti, B.R. Mestan, and E. Gunadi, “Physical Register Inlining”, *Proc. 31st Int’l Symp. Computer Architecture (ISCA 04)*, June 2004, pp. 325-335.

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

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

[21] T. Monreal, V. Viñals, J. González, A. González and M. Valero, "Late Allocation and Early Release of Physical Registers", *IEEE Transactions on Computers*, vol. 53, no. 10, Oct. 2004, pp. 1244-59.

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

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

[24] I. Park, Michael D. Powell, and T. N. Vijaykumar, “Reducing Register Ports for Higher Speed and Lower Energy”, *Proc. 35th Int’l Symp. Microarchitecture (MICRO 02)*, Nov. 2002, pp. 171-182.

[25] S. Rixner, W. Dally, B. Khailany, P. Mattson, U. Kapasi and J.Owens, “Register Organization for Media Processing” *Proc. 6th Int’l Symp. High-Performance Computer Architecture (HPCA 00)*, Jan. 2000, pp. 375-386.

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

[27] E. Torres, P. Ibáñez, V. Viñals and J.M. Llabería, “Counteracting Bank Misprediction in Sliced First-Level Caches,” *Proc. 9th Int’l Conf. on Parallel and Distributed Computing (Euro-Par 03)*, Aug. 2003, pp. 586-596.

[28] S. Wallace, and N. Bagherzadeh, “A Scalable Register File Architecture for Dynamically Scheduled Processors,” *Proc. 5th Int’l Conf. Parallel Architectures and Compilation Techniques (PACT 96)*, Oct. 1996, pp. 179-184.

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