Delaying Physical Register Allocation Through Virtual-Physical Registers

Teresa Monreal†, Antonio González*, Mateo Valero*, José González* and Victor Viñals†

†Departamento de Informática e Ing. de Sistemas
Centro Politécnico Superior - Univ. de Zaragoza
e-mail: {tmonreal,victor}@posta.unizar.es

*Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
e-mail: {antonio,mateo,joseg}@ac.upc.es

Abstract

Register file access time represents one of the critical delays of current microprocessors, and it is expected to become more critical as future processors increase the instruction window size and the issue width. This paper present a novel physical register management scheme that allows for a late allocation (at the end of execution) of registers. We show that it can provide significant savings in number of registers and thus, it can significantly shorter the register file access time. The approach is based on virtual-physical registers, which we presented in a previous work, extended with a new register allocation policy. This policy consists of an on-demand allocation in order to maximize the register usage, combined with a stealing mechanism that prevents older instruction from being delayed by younger ones. This shortens the average number of cycles that each physical register is allocated, and allows for an early execution of instructions since they can obtain a physical register for its destination earlier than with the conventional scheme. Early execution is especially beneficial for branches and memory operations, since the former can be resolved earlier and the latter can prefetch their data in advance.

1. Introduction

Dynamically-scheduled superscalar processors exploit instruction-level parallelism (ILP) by overlapping the execution of instructions in an instruction window. In spite of being able to execute instructions out-of-order, the amount of ILP that current superscalar processors can exploit is significantly constrained by data dependences, especially for non-numeric codes. The number of instructions that can be executed in parallel is highly dependent on the instruction window size and thus, wide issue processors require a large instruction window [15]. However, a large instruction window has some implications in other critical parts of the microarchitecture, such as the complexity of the issue logic [10] and the size of the physical register file [3]. In this work we are concerned with the latter problem.

The access time of a register file is significantly affected by its size, as well as its number of ports [3]. Since the current trend of increasing the issue width and the instruction window size has direct consequences on the number of ports and registers respectively, it is very likely that the register file access time will become one of the longest delays of forthcoming microprocessors. In this case, it will determine the clock cycle and thus, it will have a severe impact on the processor performance, unless it is pipelined.

However, pipelining a register file is not trivial and besides, it has significant effects on the processor. In particular, a multi-stage register file increases the branch misprediction penalty and requires extra levels of bypass logic [14]. Both issues, are critical for the performance of superscalar processors.

On the other hand, current superscalar processors require many more registers than those strictly necessary to store the values of a program. This is due to the fact that registers are allocated too early and released too late. Every instruction allocates a physical register for its destination operand much before its result is available (at decode), and this register is released much after its last consumer commits (when the following instruction with the same logical destination register commits). In this paper, we focus on the waste due to the former factor. Figure 1 shows the average number of physical registers used by a superscalar processor (written+non-written), and the number of registers that are actually wasted because of the early allocation (non-written). We assume here a processor with 160 physical registers in each file. For other details about the evaluation framework refer to Section 4.1. On average, the early allocation of registers increases the register requirements by 45% for integer and by 40% for FP; for some programs such as *li* it is responsible for an 53% average increase and in some particular cycles of the execution this figure can be as much as 500%.

This paper focuses on a novel register management scheme that allows the processor to delay the allocation of physical registers until the values that they store are available (at the end of the execution stage). We proposed this scheme in previous work [4] [5] and referred to it as virtual-physical registers. We observed on that work that the approach to allocating physical registers was critical to performance. In this paper, we present a novel register allocation approach that outperforms the former scheme. We show that virtual-physical registers with this allocation

1072-4451/99 $10.00 © 1999 IEEE

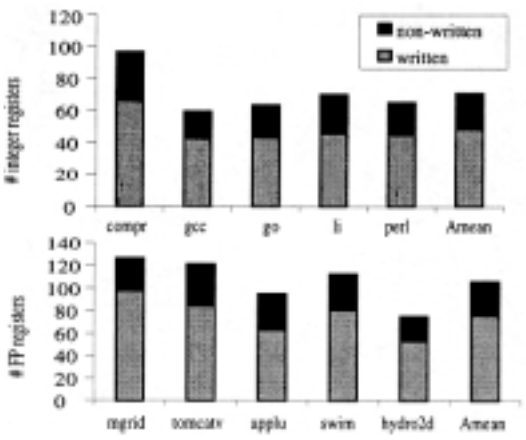

Figure 1. Register usage (written + non-written) and register waste (non-written) due to early allocation.

scheme provide a significant saving in physical registers. For instance, we show that for 5 SpecFP95 benchmarks, a virtual-physical register organization with 77 FP registers achieves about the same performance as a conventional register organization with 101 registers, in terms of instructions committed per cycle (IPC). Note that if the processor cycle is determined by the register file access time, the reduction in number of registers will imply an increase in instruction throughput.

The rest of this paper is organized as follows. Section 2 reviews the virtual-physical register renaming scheme. The new allocation policy for this scheme is presented in Section 3. Section 4 discusses the performance of the proposed approach. Finally, Section 5 outlines some related work and Section 6 summarizes the main conclusions of this work.

2. Virtual-Physical Registers

The virtual-physical register architecture is motivated by the fact that instructions do not require a storage location for their results until they are available. However, current dynamic register renaming schemes allocate such a storage in the decode stage, much earlier than the result is available. The reason is that a physical register is used for two different purposes: to store a value and as an identifier of the result, in order to keep track of dependences. Only the latter objective requires the register to be allocated at decode. Thus, the allocation of a storage location can be delayed if the processor uses another artifact to keep track of dependences. Such an artifact is what we call *virtual-physical* registers (VP registers for short).

VP registers are merely tags and do not require any physical storage. When an instruction is decoded, its destination logical register is mapped to a VP register obtained from a pool of free VP registers. Later on, when the instruction is in the last cycle of the execution stage, the VP register is mapped to a physical register taken from the pool of free physical registers. When an instruction commits, the VP and physical registers allocated by the previous instruction with the same logical destination register are released to their respective free pools.

This register management scheme requires two map tables: the general map table (GMT), which indicates for each logical register its latest VP association and the latest physical mapping of this VP register, if any, and the physical map table (PMT), which denotes for each VP register its latest physical mapping, if any. More details about the operation of this approach can be found in [4] [5].

Since VP registers are just tags, a processor should typically be provided with the maximum required amount, namely the number of logical registers plus the maximum number of instructions in-flight. However, the physical register file should be dimensioned to the smallest size that provides a performance not much lower than an infinite number of registers in order to keep its access time low. It is thus not guaranteed that every instruction in-flight finds a free physical register when it finishes the execution stage. Therefore, the policy used to allocate physical registers to instructions is critical to performance.

In fact, a conventional renaming scheme could be defined as a scheme that allocates the available physical registers in program order and forces the processor to stall the decoding of instructions when it runs out of physical registers. The virtual-physical register scheme could achieve the same behavior with an adequate policy for register allocation, i.e., by allocating physical registers to the oldest instructions in the window, with the additional advantage that it does not require the decode of instruction to be stalled when physical registers have been used up. However, other allocation policies are possible, and may be more effective.

When an instruction finishes its execution and there is not any free physical register, the instruction is kept in the instruction queue and executed later on, with the expectation that some physical registers will have been freed in the meantime. However, if this instruction was the oldest instruction in the window, no instruction would commit and thus, no physical register would be freed. To avoid this deadlock situation the allocation policy proposed in [4] guarantees that a given number of oldest instructions in the window (NRR) will obtain a register when they reach the write-back stage, where NRR is an implementation parameter. In other words, that allocation scheme assigns NRR registers to the oldest instructions in the window that have a destination operand, whereas the rest of physical registers are allocated on-demand, that is, they are assigned to the instructions that first reach the write-back stage.

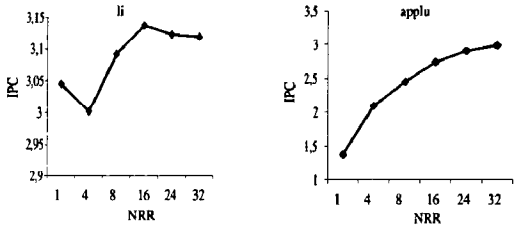

Figure 2. Effect of varying NRR for two particular programs.

3. A New Register Allocation Approach for Virtual-Physical Registers

We realized in our previous work that the performance of the processor was very sensitive to the value of NRR. The optimal value of NRR for different programs is quite distinct, and even for a single program, varying the value of NRR across different sections of the execution may provide significant benefits. Such a scheme for a dynamic tuning of the NRR parameter was concluded to be critical and this is what has motivated this work. For instance, Figure 2 shows the IPC for *li* and *applu* when NRR is varied from 1 to 32, assuming 64 physical registers in each file. Other details about the evaluation framework can be found in section 4.1. Note that the optimal value for NRR for *li* is 16 whereas for *applu* it is 32.

For the whole benchmark suite, the value of NRR that has the best average performance is 32, which is its maximum value for 64 physical registers. A virtual-physical scheme with maximum NRR is conceptually similar to the renaming scheme of the Power3, as discussed in Section 5, and it is one of the schemes that we use for comparison in this work.

Note that in fact this previous register allocation scheme is not the only approach that may guarantee a deadlock avoidance. On the other hand, finding the optimal register allocation policy seems to be an unsolvable problem even with a perfect knowledge of future register references. We have then to rely on some heuristics that try to approximate such an optimal scheme. The approximation we propose is based on the following two rules:

- Registers should be allocated to the instructions that can use them earlier. In this way, the average number of unused registers is minimized.
- Given any two instructions, if the execution of one of them should be delayed by the lack of registers, the most appropriate candidate is the youngest instruction, since delaying the execution of the oldest would delay its commit, which in turn would also delay the commit of the youngest one.

These two criteria can be met by the following scheme. Every instruction allocates a physical register in the last cycle of the execution stage (just before write-back) if there are free registers. This meets the first criterion since

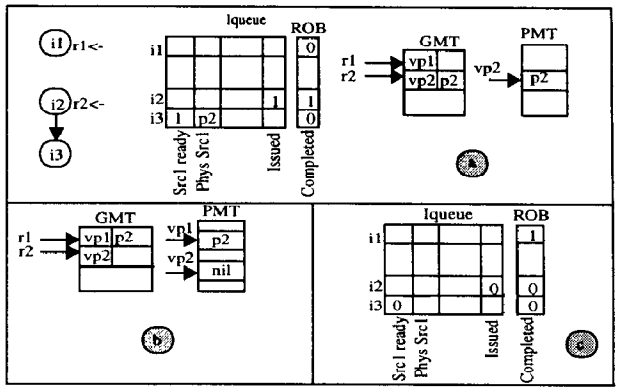

Figure 3. Example of virtual-physical renaming with DSY allocation.

registers will be allocated by the instructions that first finish execution. If an instruction reaches the last cycle of the execution stage and there is not any free physical register, it is checked whether there is any younger instruction that has already allocated a register. If this is the case, it is better to stall the younger instruction rather than the older one, based on the second criterion. As suggested in [7], this can be achieved by stealing the register allocated by the younger instruction and assigning it to the older one. If there are more than one younger instruction with a register already allocated, the youngest one will be chosen. We refer to this register allocation scheme as *on-Demand with Stealing from Younger* (DSY).

3.1. Implementing the DSY Register Allocation Scheme

Identifying whether there is a younger instruction with an allocated register is done by inspecting the reorder buffer, searching for any younger entry with the execution-complete bit set. If several are found, the youngest one is chosen. Let us refer to the instruction that demands a register as i_1, and to the instruction from which the register is stolen as i_2. The VP register identifier allocated by i_2 is obtained from the reorder buffer (this additional field in the reorder buffer is an overhead of the virtual-physical policy) and the physical register identifier is obtained from the PMT table. Let us refer to the VP destination register of i_1, the VP destination register of i_2 and the physical destination register of i_2 as VP_1, VP_2 and P_2 respectively (see Figure 3.a). When i_1 steals the physical register of i_2, the PMT table is updated to reflect that now VP_1 is mapped to P_2 and VP_2 is not longer associated to P_2 (see Figure 3.b).

The instruction i_2 must be re-executed in the future. A simple approach to achieving this is to keep instructions in the instruction queue until they retire, with a flag that indicates whether they have been issued. By marking i_2 as not issued, the issue logic will choose it again for issue in the future (Figure 3.c).

Since i_2 has been executed in the past, the consumers of VP_2 (i.e. instructions with a source operand renamed to VP_2) have marked this operand as ready. However, it is not ready anymore since its associated physical register has been stolen. Turning this operand into not ready can be done by using the buses that are used to awake instructions, as described below.

In a conventional processor, when an instruction completes execution[1], it broadcasts the identifier of its physical destination register to all the entries in the instruction queue. Every entry checks if any of its source operand identifiers correspond to the broadcast one, and those that match are marked as ready. For virtual-physical registers, each entry in the instruction queue identifies each source operand by means of both a VP register and a physical register identifiers. When an instruction completes, both the VP and physical identifiers of the destination register are broadcast to the instruction queue. Each entry compares the VP identifiers of its source operands against the broadcast one, and in case of a match, the broadcast physical identifier is copied in the corresponding field of the matching operand, and this operand is marked as ready. On the other hand, when the physical destination register of an instruction is stolen, its corresponding VP tag (VP_2 in the example) is broadcast to the instruction queue in order to mark as NOT ready any matching source operand (Figure 3.c).

Note that some of the instructions that have VP_2 as a source operand may have been issued at the time when this operand becomes not ready. These instructions have read a correct source operand and thus, the result that they will produce will be correct. At the time they finish, if there are free physical registers they will be able to store their result and dependent instructions will be allowed to be issued. However, instruction i_2 will eventually be executed again and will allocate a new physical register (let us denote it by P_3) for its destination. At that moment, the VP_2 and P_3 identifiers will be broadcast to the instruction queue, and those instructions that consume VP_2 will copy the new physical mapping and will be re-issued if not yet completed (i.e. it has been executed and allocated a physical register for its destination). The same happens to consumers of these re-issued instructions: they will be issued no matter if they have already been issued in the past.

3.2. Implicit Benefits of the DSY scheme

As described above, the DSY scheme may cause multiple executions of some instructions and all the re-executions of the same instruction will produce the same result. In this section, we point out that these multiple executions implicitly have a very positive effect on the control speculation mechanism as well as the data cache memory.

Among the multiple times that an instruction is executed, all of them except the last one could be regarded as premature executions, that would not occur at that time if the processor allocated physical registers from oldest to youngest instructions. Indeed, the physical register of an instruction i is stolen only when in the instruction window there are more instructions older than it that require a physical register. A conventional renaming mechanism would assign all physical registers to the older instructions and instruction i would have not even been fetched. On the other hand, with virtual-physical registers, this instruction gets a physical register because it finishes execution earlier than some older instructions. However, when an older instruction completes its execution and finds no free physical registers, it steals the register from i, which forces its later re-execution.

The preliminary execution of some instructions have two important benefits:

- A preliminary execution of a branch instruction will validate its prediction and in case of misprediction, the instructions of the wrong path will be squashed and the fetching from the correct path will be started immediately. In the conventional renaming scheme, such validation would occur much later, since the instruction would be delayed by the lack of registers.
- A preliminary execution of a load instruction that misses in cache would fire the fetching of the data. When the instruction is definitely re-executed this data element may be already in cache and result in a cache hit. In other words, the early (preliminary) execution of memory instructions acts as a small-distance data prefetching scheme, hiding the memory latency of some cache misses.

Finally, note that the latency as well as resource consumption of re-executed instructions can be significantly reduced by means of a reuse mechanism [11]. Instructions that are to be re-executed could keep their results in a reuse buffer and later on, when physical registers are available, these results could be directly copied into the new physical registers. In this way, re-executed instructions would not increase the contention in the functional units. In the analysis presented in this paper, such an instruction reuse mechanism is not considered.

4. Performance Evaluation

4.1. Experimental Framework

The virtual-physical register renaming approach has been evaluated by means of a cycle-based timing simulator of a dynamically-scheduled processor derived from the SimpleScalar v3.0 tool set [2]. The sim-outorder simulator has been modified to include physical register files (integer

1. In fact, it is done some cycles before in order to overlap the latency of the issue and read logic with the last cycles of the execution.

and FP) where the results of all instructions are stored (instead of temporarily storing them in the RUU and moving them to the architected register file at commit). This is the approach used by some current microprocessors such as the MIPS R10000 and the Alpha 21264. The main parameters of the microarchitecture we use in our simulations are presented in Table 1. We refer to such microarchitecture as a conventional processor. Then, the simulator has been extended to include the proposed virtual-physical renaming, leaving the remaining architectural parameters unchanged.

Table 1. Processor microarchitectural parameters

Parameter	Value
Fetch width	8 instructions (up to 2 taken branches)
L1 I-cache	32 KB, 2-way set-associative, 32 byte lines, 1 cycle hit time
Branch predictor	18-bit Gshare with speculative updates
Window size	128 entries
Functional units (latency)	8 Simple int (1); 4 int mult (7); 6 simple FP (4); 4 FP mult (4); 4 FP div (16); 4 load/store
Load/Store queue	64 entries with store-load forwarding
Issue mechanism	out-of-order issue. Loads may execute when prior store addresses are known
Physical registers	48-160 int / 48-160 FP
L1 D-cache	32 KB, 2-way set-associative, 64 byte lines, 1 cycle hit time
L2 unified cache	1 MB, 2-way set-associative, 64 byte lines, 12 cycles hit time
Main memory	infinite size, 50 cycles access time
Commit width	8 instructions

We used ten benchmarks from the Spec95 suite: five integer programs (*compress*, *gcc*, *go*, *li* and *perl*) and five FP programs (*mgrid*, *tomcatv*, *applu*, *swim* and *hydro2d*). All programs were simulated to completion, excepting *tomcatv*, for which the initial part that reads a huge input file was skipped. Table 2 lists the inputs and the number of executed instructions. The programs were compiled with the Compaq/Alpha Fortran and C compilers with the maximum optimization level (-O5).

Table 2. Benchmarks

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

4.2. Performance Statistics

Figure 4 shows the average number of instruction committed per cycle (IPC) for each benchmark as well as the harmonic mean for integer and FP programs, assuming 64 physical registers in both the integer and FP files. Three different register management schemes are compared: the conventional one (conv), virtual-physical registers with the original register allocation policy (vp-ori), and virtual-

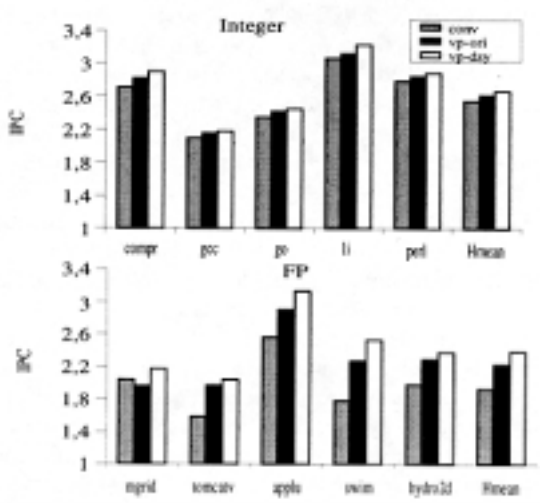

Figure 4. Performance of the virtual-physical renaming versus the conv. scheme for a 64 physical registers. Two different register allocation policies are shown for virtual-physical renaming: the original and the DSY.

physical registers with the DSY allocation policy presented in this paper (vp-dsy). We can observe that the benefits of virtual-physical registers for FP codes are much more significant than for integer programs. This is an expected result since FP programs in general cause a much higher register pressure. The virtual-physical organization significantly outperforms the conventional organization, providing an average speed-up of 5% and 24% for integer and FP codes respectively. The difference between the original and the DSY allocation policies is noticeable, DSY provides an average speedup of 2% and 7% for integer and FP codes respectively.

For the DSYconfiguration, the percentage of instructions that have their destination physical register stolen is 5.26% for integer programs and 11.76% for FP codes, which results in a 9.79% and a 57.75% of instructions re-executed respectively. This is due to the different behavior of these applications. FP programs exhibit more ILP and a low branch miss rate, and thus the instruction window is usually filled up. Thus the VP scheme can assign physical registers to instructions far away from the oldest instructions in the window, increasing the ILP extracted and thus the performance, at the expense of a higher number of re-executions. On the other hand, integer applications experience a much higher branch miss ratio, which implies that the instruction window cannot be completely filled up, and then, the VP scheme cannot yield so much performance benefits by exploiting ILP.

Figure 5 shows the impact of preliminary executions of load instructions. It depicts the percentage of loads that miss in cache and among them, the percentage that have been re-

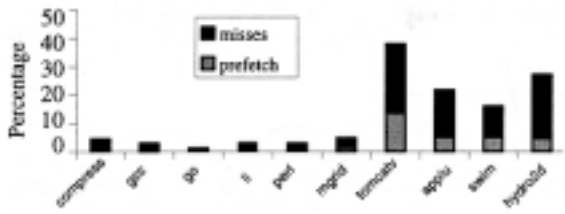

Figure 5. Percentage of loads that miss on cache and their data is prefetched through re-execution.

executed. We can observe that FP applications experience a reasonable degree of prefetch (28% on average), especially the *tomcatv*, which also obtains the best speedup (it re-executes the 35% of loads that miss in cache).

Analyzing the effect of a varying number of registers on the processor performance may be more interesting than just looking at a particular register file size. In general a processor designer would be interested in finding the best trade-off between number of registers and performance.

Figure 6 illustrates how the processor performance varies as a function of the number of physical registers for both the conventional and the virtual-physical organization. For the latter, the original register allocation policy as well as the DSY one are shown. Virtual-physical registers with DSY allocation is always better than virtual-physical registers with the original allocation policy, which in turn is better than the conventional renaming scheme. Note also that the difference among the three schemes is more significant for FP codes and increases as the number of physical registers decreases. We can observe that virtual-physical registers can provide significant savings in number of registers. A candidate design point for the number of registers to be implemented in a processor would be the lowest number of registers that provides a performance close to that of an infinite size register file (i.e. as many registers as reorder buffer entries plus number of logical registers). For instance, if we could afford about 10% IPC degradation with respect to the maximum IPC, we would choose 61 integer and 101 FP registers for the conventional scheme, whereas for virtual-physical registers 45 and 77 would suffice respectively. This implies a saving of 26% and 24% respectively, which directly translates in savings in the register file access time and area, since both are significantly affected by the number of registers [3].

5. Related Work

Register renaming is an old concept that first appeared in the FP unit of the IBM 360/91 [13]. Nowadays it is used by all dynamically scheduled processors. Different schemes basically differ in the organization of the storage location for the register values. Some processors keep non-committed values in the reorder buffer and copy these values to the register file at commit (e.g. Intel P6 [6]); others

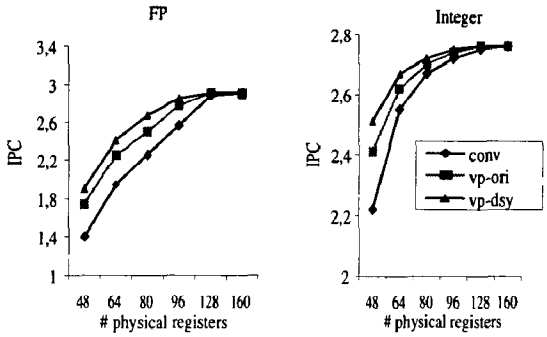

Figure 6. Performance as a function of number of physical registers for the conv. register organization and the virtual-physical scheme with the two different allocation policies.

have a register file for non-committed values and another for committed results (e.g. PowerPC 620 [9]); finally, some processors have a single register file (one for integer and another for FP) that holds both non-committed and committed data (e.g. Alpha 21264 [8]). The virtual-physical organization that we have proposed in this paper can be applied to the two last schemes and we have assumed the last one for the presented evaluation.

The virtual-physical register renaming scheme presented in this paper builds upon the approach that we presented in [4]. The main contribution of this paper is a novel register allocation scheme that allocates physical registers at the end of the execution stage, using an on-demand policy with stealing from younger instructions.

Another approach for delaying the allocation of physical registers was proposed by Wallace and Bagherzadeh [16]. Their motivation was to have a multiple-banked organization with just one write port per bank. Delaying the allocation of physical registers until result write time allowed the processor to avoid conflicts in the banks. Their scheme had the same type of deadlock hazard as virtual-physical registers have. Their solution was based on shifting the processor to a special mode when the oldest instruction was unable to issue or complete. In this special mode, only the oldest instruction was allowed to execute and its result was stored in the register that this instruction would release at commit. Note that this is very similar to our former approach with one reserved register (NRR=1). In fact, this scheme does not reserve any register, but uses one that is sure to be released right away.

Finally, the Power3 implements what they call virtual renaming [1] [12]. Like the PowerPC 620 [9], this processor has two register files: one for committed values, which is referred to as architected register file, and another for non-committed values that is called rename buffers. In this processor, there are 16 rename buffers for integer and 24 for FP data. However, an operand is identified by one bit more than those required by a rename buffer identifier. This additional bit is called the virtual bit. This allows up to two

in-flight instructions to use the same rename buffer for its result. The older assignment is considered to be the 'real' one whereas the younger is called the 'virtual' one. They are distinguished by the value of the virtual bit. Only instructions with real operands (source and destinations) are allowed to be issued. When an instruction commits, its destination rename buffer is switched from real to virtual, and that of the younger instruction that uses the same physical buffer is switched from virtual to real. The processor allows up to 32 instructions in-flight (due to the size of the reorder buffer) but, unlike our proposal, only the 16/24 oldest instructions with an integer/FP destination register respectively are allowed to be issued. In fact, these scheme is conceptually very similar to our original proposal when the number of reserved registers is set to the number of rename buffers.

To summarize, Wallace and Bagherzadeh and the Power3 schemes represent two extreme points in the design space. The former allocates physical registers almost on demand, with the exception of the oldest instruction, whereas the latter assigns all physical buffers to the oldest instructions that have a destination operand. The former can execute instruction far beyond the actual commit point much earlier than the latter. However, when the oldest instructions run out of registers, the former scheme has very low performance since instructions are executed sequentially. Our proposal, virtual-physical registers with DSY allocation, can be as aggressive as the latter, but when it realizes that old instructions are progressing slowly due to the lack of registers, it steals some registers from the younger instructions and give them to the older ones.

An orthogonal approach to reducing the register pressure was proposed by Jourdan et al. [7]. Their scheme takes advantage of instruction repetition. The idea is to identify instructions that produce the same result and allocate the same physical register for all of them.

6. Conclusions

In this paper we have presented a novel register renaming scheme that allows for the late allocation of physical registers. In particular, physical registers are allocated at the end of the execution stage, rather than at decode time as conventional processors do. The direct advantage of this scheme is a significant reduction in the register pressure. For instance, we have evaluated that it can provide a 26% and 24% reduction in the number of integer and FP registers, and achieve the same IPC rate as a conventional scheme. This reduction in number of registers translates into a shorter access time to the register file, which is likely to be a critical issue of forthcoming microprocessors, and thus, it may significantly increase performance.

The proposed scheme has also important indirect advantages. In particular, it allows branches to be resolved earlier and load/store instructions to prefetch their data. In addition, the re-execution of instructions caused by the stealing feature can be done very effectively by means of a reuse mechanism.

The novel register allocation policy has been shown to be more effective than previous proposals for late register allocation. The on-demand allocation with stealing from the younger provides maximum look-ahead when ILP is limited but this look-ahead never penalizes older instructions.

7. Acknowledgments

This work has been supported by contracts CYCIT TIC98-0511 and ESPRIT 24942, by the Programa Europa de Investigación (CAI CONSI + D), by the grant 1996FI-03039-APDT, and by the CEPBA.

8. References

[1] P. Bose and J. Moreno. *Private communication*, June 1999

[2] D. Burger and T.M. Austin. "The SimpleScalar Tool Set v2.0", Technical report 1342, University of Wisconsin-Madison, CS Department, June 1997

[3] K.I. Farkas, N.P. Jouppi and P. Chow. "Register File Considerations in Dynamically Scheduled Processors", in *Proc. of 2nd. Int. Symp. on High-Performance Computer Architecture*, pp. 40-51, 1996

[4] A. González, J. González and M. Valero. "Virtual-Physical Registers", in *Proc. IEEE 4th. Int. Symp. on High-Performance Computer Architecture*, pp. 175-184, 1998

[5] A. González, M. Valero, J. González and T. Monreal. "Virtual Registers", in *Proc. of Int. Conf. on High-Performance Computing*, pp. 364-369, 1997

[6] L. Gwennap. "Intel's P6 Uses Decoupled Superscalar Design", *Microprocessor Report*, pp. 9-15, Feb. 1995

[7] S. Jourdan et al. "A Novel Renaming Scheme to Exploit Value Temporal Locality through Physical Register Reuse and Unification", in *Proc. of 31st. Int. Symp. on Microarchitecture*, pp. 216-225, 1998

[8] R.E. Kessler. "The Alpha 21264 Microprocessor", *IEEE Micro*, 19(2):24-36, March 1999

[9] D. Levitan, T. Thomas and P. Tu. "The PowerPC 620 Microprocessor: A High-Performance Superscalar RISC Microprocessor", in *Proc. of 40th. IEEE Computer Society International Conference*, pp. 285-291, 1995

[10] A.S. Palacharla, N.P. Jouppi and J.E. Smith. "Complexity-Effective Superscalar Processors", in *Proc. of 24th. Int. Symp. on Computer Architecture*, pp. 206-218, 1997.

[11] A. Sodani and G.S. Sohi. "Dynamic Instruction Reuse", in *Proc. of 24th. Int. Symp. on Computer Architecture*, pp. 194-205, 1997

[12] P. Song. "IBM's Power3 to Replace P2SC", *Microprocessor Report*, 11(15): 23-27, Nov. 1997

[13] R.M. Tomasulo. "An Efficient Algorithm for Exploiting Multiple Arithmetic Units", *IBM Journal of Research and Development*, 11(1), pp. 25-33, Jan. 1967.

[14] D.M. Tullsen et al. "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor", in *Proc. of the 25th. Int. Symp. on Computer Architecture*, pp. 191-202, 1996

[15] D.W. Wall. "Limits of Instruction-Level Parallelism", Technical Report WRL 93/6 Digital Western Research Laboratory, 1993.

[16] S. Wallace and N. Bagherzadeh. "A Scalable Register File Architecture for Dynamically Scheduled Processors", in *Proc. 1996 Conf. on Parallel Architectures and Compilation Techniques*, pp. 179-184, 1996