Virtual-Physical Registers

Antonio González, José González and Mateo Valero

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya,
Jordi Girona 1-3, Edifici D6, 08034 Barcelona, Spain
e-mail: {antonio,joseg,mateo}@ac.upc.es

Abstract

A novel dynamic register renaming approach is proposed in this work. The key idea of the novel scheme is to delay the allocation of physical registers until a late stage in the pipeline, instead of doing it in the decode stage as conventional schemes do. In this way, the register pressure is reduced and the processor can exploit more instruction-level parallelism.

Delaying the allocation of physical registers require some additional artifact to keep track of dependences. This is achieved by introducing the concept of virtual-physical registers, which do not require any storage location and are used to identify dependences among instructions that have not yet allocated a register to its destination operand. Two alternative allocation strategies have been investigated that differ in the stage where physical registers are allocated: issue or write-back. The experimental evaluation has confirmed the higher performance of the latter alternative.

We have performed an evaluation of the novel scheme through a detailed simulation of a dynamically scheduled processor. The results show a significant improvement (e.g., 19% increase in IPC for a machine with 64 physical registers in each file) when compared with the traditional register renaming approach.

1. Introduction

Dynamically scheduled processors are the most common organization nowadays in the marketplace. Most of the latest general purpose microprocessors of major vendors, as well as those announced for the near future, use such organization. A dynamically scheduled processor has the ability of executing instructions out-of-order, and thus, it puts obviously much less constraints on the issue order of instructions than an in-order execution processor. This in general results in much higher instruction-level parallelism (ILP).

However, a dynamically scheduled processor does not have complete freedom to chose the execution order of instructions. In particular, it must obey instruction dependences. These dependences are usually classified into three types [5]:

- Data dependences. They occur when one instruction produces a value that is used by another instruction.
- *Name dependences*. They are caused by the reuse of storage locations, namely registers and memory. However, in this case there is no flow of data between the involved instructions.
- *Control dependences*. They are due to conditional branches. These instructions determine which instructions should be executed later.

Name dependences through registers are usually eliminated by providing multiple storage locations for the same register name and keeping track of which storage location is referred to by each different instance of the same name. This technique is called *dynamic register renaming*. In this context, the name of a register is referred to as a *logical register* whereas the physical location to which it is mapped at a given time is called a *physical register.*

The amount of physical storage devoted to register renaming determines the maximum number of simultaneously live values, and therefore, it limits the instruction window size. Future microprocessors will likely manage a larger instruction window to increase the exploitation of ILP and thus, the register requirements will be higher. Enlarging the physical register file is an obvious solution for a balanced design. However, the hardware cost of the register file is very high mainly because of the large number of ports that it has. In addition, larger register files have a longer access time, and this may increase the critical path length and penalize performance [1].

In this paper we propose a novel register renaming approach that significantly reduces the register pressure. This benefit can be used either to increase the processor performance through the increase of the active instruction window size or to reduce the hardware cost by reducing the amount of storage devoted to register renaming, without loosing performance when compared with the traditional renaming scheme. The novel register renaming approach is based on introducing a new concept that is called *virtual-physical registers*. Virtual-physical registers are names that are used to identify values that will be produced by instructions in the future, and thus, do not use any storage location. Virtual-physical registers are used to keep track of dependences among instructions and therefore, to drive the issue logic. Physical registers are used to store the live values of instructions as in the conventional scheme and thus, virtual-physical registers are mapped to physical registers at some point in time. However, the virtual-physical register organization allocates a physical register for a much shorter interval of time than the conventional scheme, which is the reason for the reduction in register pressure.

The performance evaluation of the virtual-physical register approach for a dynamically scheduled processor with 64 physical registers shows a 19% speedup when compared with the traditional scheme, and about the same performance than the traditional scheme with just about half the number of physical registers for renaming.

The rest of this paper is organized as follows. Section 2 reviews the traditional register renaming approaches. Section 3 presents the novel virtual-physical register scheme. The performance of the new scheme is compared with the traditional one in section 4. Finally, the main conclusions of this work are summarized in section 5

2. Register renaming

Register renaming was first implemented for the floating-point unit of the IBM 360/91 [14]. Register renaming is a key issue for the performance of out-of-order execution processors and therefore, it is extensively used. In this paper we focus on dynamically scheduled processors that implement precise exceptions [9]. In such processors, instructions are committed in-order. After being decoded, instructions are kept in the instruction reorder buffer until they commit. The size of the reorder buffer determines the maximum number of in-flight instructions. These instructions are usually called the instruction window and the size of the reorder buffer is the size of the instruction window. In other words, the instruction window is defined as the set of instructions from the oldest not committed instruction to the latest decoded instruction

The objective of register renaming is to remove name dependences through registers (anti- and output dependences). This is achieved by allocating a free storage location for the destination register of every new decoded instruction. There are two different schemes regarding the approach taken to implement these rename storage locations. In particular, the two following approaches are the most common solutions to provide the rename storage locations:

- The entries of the reorder buffer [11]. In this case, the result of every instruction is kept in the reorder buffer until it is committed. It is then written in the register file. The source operands that are available when an instruction is decoded are read either from the register file or from a reorder buffer entry. Those operands that are not ready at decode are forwarded from the execution units to the corresponding instruction queue entries (reservation stations) when they are produced. When an instruction commits, its result is copied from the reorder buffer to the register file. There is a slight variation that includes a register buffer used just for renaming and avoids to store the result in the reorder buffer(e.g. PowerPC 604 [12]).
- A physical register file. In this case there is a physical register file that contains more registers than those defined in the ISA (instruction set architecture), which are referred to as *logical registers*. By means of a map table, each logical register is mapped to a *physical register* in the decode stage. The destination register is mapped to a free physical register whereas source registers are translated to the last mapping assigned to them. When an instruction commits, the physical register allocated by the previous instruction with the same logical destination register is freed. In this scheme, the operands are always read from the physical register file, which simplifies the operand fetch task when compared with the previous model.

Both register renaming schemes are being used in the latest microprocessors. The first one is used by the Intel Pentium Pro [2], the PowerPC 604 [12], and the HAL SPARC64 [3]. The MIPS R10000 [15], and the DEC 21264 [4] are current implementations of the second approach. In this paper, we focus on the second scheme. A comparison of both approaches in terms of cost-effectiveness could be an interesting study but it is beyond the scope of this paper. However, notice that both approaches have similar renaming storage requirements. In both cases, a new rename storage location is allocated when an instruction is decoded, and a location is released when an instruction commits. Therefore, the main advantage of the virtual-physical register organization, which is the allocation of rename storage locations for a shorter period of time, also applies when compared with the reorder buffer approach.

In the physical register file organization, to take advantage of a given instruction window size a number of physical registers close to the number of logical registers plus the window size is required since most of the instructions have a destination register. This is so because when the instruction window is empty (e.g., after a branch misprediction), each logical register is mapped to a physical register. Thus, the minimum number of physical registers that are used is equal to the number of logical registers. In addition, for every instruction whose destination operand is a register, an additional register is allocated when it enters the window (decode stage) and a physical register is released when it leaves the window (commit stage).

3. Virtual-physical registers

This section describes the virtual-physical register renaming approach. First, the motivations for the new scheme are presented and then, its implementation is detailed.

3.1. Motivation

The motivation for the register renaming approach that is proposed here comes from the observation that the conventional register renaming scheme based on a physical register file allocates a new physical register for every instruction with a destination register. This register is allocated when the instruction is decoded and it is not released until the next instruction that has the same logical destination register is committed.

Notice that this is a conservative approach that is used for simplicity reasons. In fact, the value that a register holds is live for a shorter period of time. The lifetime of the value produced by an instruction extends from the time the execution of the instruction finishes to the time when all the instructions that use such value have read it and are guaranteed to commit.

Thus, the conventional register renaming scheme "wastes" a register for each instruction that is in either of the two following states:

- It has been decoded but its execution has not finished yet (i.e., it is either waiting in the instruction queue to be issued or being executed in its corresponding functional unit).
- It has been committed as well as all the instructions that used the produced value but the next instruction with the same logical destination register has not been committed yet.

As described by other authors [8] [10], the second source of register waste can be eliminated by associating a counter with each physical register that keeps track of the pending read operations. A register is freed whenever the counter is zero, provided that the corresponding physical register has been subsequently renamed to another physical register.

The virtual-physical register renaming scheme eliminates the first factor of register usage waste. Notice that this factor can be very important in the presence of long latency instructions and parts of codes with small amount of ILP. In such circumstances, some instructions spend long time in the instruction queue waiting for their operands and they use (unnecessarily) a physical register for all that period of time. For instance, suppose the following sample code (destination operands are on the left):

```
load f2,0(r6)
fdiv f2,f2,f10
fmul f2,f2,f12
fadd f2,f2,1
```

These four instructions in this code can be fetched and decoded in the same cycle in a four-way superscalar processor. At that time, four different physical registers are allocated to logical register `f2`, each one corresponding to a different instruction. Let us call them `p1`, `p2`, `p3` and `p4` respectively. Assume that in the next cycle the load instruction can start its execution but it produces a cache miss. Assume also that the remaining instructions can be issued as soon as they have all their operands and that they can commit as soon as it execution finishes. Suppose that the cache-miss latency is 20 cycles, the FP division takes 20 cycles, the FP multiplication takes 10 cycles and the FP addition takes 5 cycles.

In the conventional register renaming scheme, `p1`, `p2` and `p3` are used for 42 cycles (i.e. 1 cycle spent in the decode of the load, 20 cycles in the execution of the `load`, 20 cycles in the execution of the `fdiv` and 1 cycle in the commit of the `fdiv`), 52 cycles and 57 cycles respectively. However, if the physical registers were not allocated until the corresponding instruction finished its execution, they would only be used for 21, 11 and 6 cycles respectively. That is, the register pressure would be reduced by 75% (from 151 to 38 cycles) if we measure the register pressure as the sum of the number of cycles that a register is allocated for each produced value. If the physical registers were allocated when the corresponding instructions were issued, they would be used for 41, 31 and 16 cycles respectively, which still implies a reduction of 42% in the register pressure.

Load instructions that miss in cache is a common source of long latency operations. Due to the increasing gap between processor and memory speed, the load miss latency measured in processor cycles may be even higher in future microprocessors. Other source of long latency operations are complex floating point arithmetic instructions such as divide or square root. However, they usually represent a small fraction of executed instructions. In any case, even for short latency operations, the reduction in register pressure can be significant when the code includes long chains of dependent instructions, as it is the case of the above sample code. Finally, note that the amount of time that instructions spend in the instruction window before being executed will grow when the size of the instruction window increases, as it is expected in the future.

Notice that the reason why logical registers are mapped to physical registers at decode stage in the conventional scheme is mainly to keep track of dependences among instructions. In fact, what is just required to keep track of dependences is a tag that identifies the last producer for every logical register. These tags are used to determine from where the source operands are to be read.

3.2. The virtual-physical register renaming

The new organization, which is called *virtual-physical registers*, is based on adding a new type of registers, in addition to the conventional logical and physical types. The registers referenced by the instructions of the ISA are referred to as *logical registers*. When an instruction is decoded, its destination register is mapped to a new tag. Tags are not related to any physical storage location and therefore we will call them *virtual-physical registers (VP registers)*. Later on, when the instructions finishes its execution, it allocates a physical register to store its result. Finally, when the instruction commits, the physical register allocated by the previous instruction with the same logical destination register is freed.

The virtual-physical register renaming scheme can be used for both integer and floating point registers. Thus, the implementation described below is replicated for both register files.

3.2.1 Register map tables. The virtual-physical register organization is implemented by means of two register map tables (see Figure 1). There is a table, which is called the *general map table (GMT)*, that is indexed by the logical register number and contains the following three fields:

- VP register: the last virtual-physical register to which the logical register has been mapped.
- P register: the last physical register to which the logical and the virtual-physical registers have been mapped, if any.
- V bit: indicates whether the P field contains a valid value, that is, whether a physical register has already been allocated to this logical register.

The other table is called the *physical map table (PMT)*. It has an entry for each virtual-physical register and it contains the last physical register to which the virtual-physical register has been mapped. Alternatively, this map table could be implemented by means of a CAM (content-addressable memory) with a number of entries equal to number of physical registers, which is much lower than the number of virtual-physical registers. This approach is used for instance by the DEC 21264 [4] to implement the logical to physical map table.

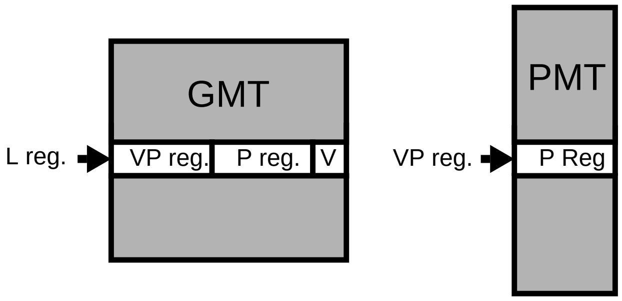

Figure 1. Tables required by the virtual-physical register organization.

In addition, there is a pool of free physical registers, like in the conventional scheme, and a pool of free virtual-physical registers.

The GMT has NLR rows of $\lceil \log_2(\text{NVR}) \rceil + \lceil \log_2(\text{NPR}) \rceil + 1$ bits each, where NLR is the number of logical registers, NVR in the number of virtual-physical registers and NPR is the number of physical registers. The PMT has NVR rows of $\lceil \log_2(\text{NPR}) \rceil$ bits each or NPR rows of $\lceil \log_2(\text{NVR}) \rceil$ bits each if it is implemented through a CAM.

Since virtual-physical registers are not related to any storage location, the number of such registers has a small impact on the hardware cost, especially if the PMT is implemented through a CAM. To guarantee that the processor never stalls because of the lack of them, the NVR must be equal to the number of logical registers (NLR) plus the instruction window size.

3.2.2 Functional description. For each new decoded instruction, its source operands are renamed either to virtual-physical registers or to physical registers if they are available. In particular, for each source register operand, the GMT is looked up. If the V bit is set, the logical register is renamed to the physical register specified in the P register field; otherwise it is renamed to the virtual-physical register. The destination logical register, if any, is renamed to a free virtual-physical register. The corresponding entry of the GMT is updated as follows: the VP register field is modified to reflect the new mapping and the V field is reset. The previous value of the VP register field is kept in the reorder buffer to restore a precise state in case of a branch misprediction or an exception. Then, the instruction is dispatched to the instruction queue,

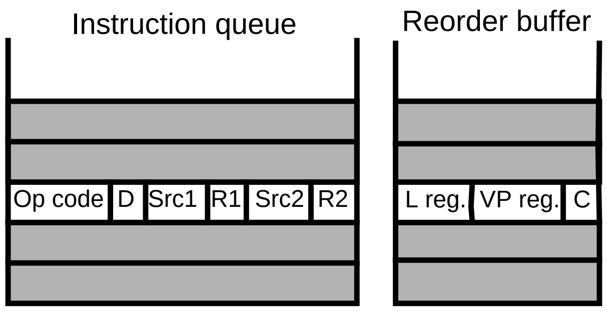

Figure 2. The instruction queue and the reorder buffer.

where it waits until it is issued, and the reorder buffer, where it remains until it is committed.

An entry of the instruction queue has the following fields (see Figure 2):

- Op code: the operation code.
- D: The virtual-physical destination register.
- Src1 and Src2: the identifiers of the two source operands (to simplify the explanation we assume that they are always registers). Each identifier corresponds either to a virtual-physical register or to a physical register
- R_1 and R_2: these are the ready bits of the source operands. When an operand is ready, the Src field contains a physical register identifier. Otherwise it contains a virtual-physical register identifier.

An entry of the reorder buffer has the following fields (see Figure 2):

- L register: the destination logical register identifier.
- C: a single bit that indicates whether the instruction has completed its execution.
- VP register: this field identifies the virtual-physical mapping of the last instruction that had the same logical destination register.

An instruction can be issued when the R fields of both operands are set. This also guarantees that the Src fields contain physical register identifiers. When an instruction is issued, it reads its register operands from the physical register file using the Src identifiers of the corresponding entry in the instruction queue (if the operand is not forwarded from the output of a functional unit).

Every instruction whose destination is a register allocates a new physical register when its execution completes. At this time, a new physical register is taken from a free pool of physical registers (the solution to the lack of free physical registers is considered in the next section; for the sake of simplicity we assume now that this event never happens). Then, the PMT is updated to reflect the new virtual-physical to physical mapping. In addition, the virtual-physical register identifier of the destination operand is broadcast to all the entries in the instruction queue along with the physical register identifier. If there is a match in a Src field whose corresponding R bit is not set, this field is updated with the physical register and the corresponding R bit is set. The virtual-physical register and the associated physical register are also broadcast to the GMT. Each entry then compares its VP register identifier with the one broadcast and if there is a match, the physical register identifier is copied into the P register field and the V flag is set. In this way, any new decoded instruction that uses such logical register will find the corresponding physical register in the GMT. Finally, the C flag of the corresponding entry of the reorder buffer is set.

When an instruction commits, the virtual-physical register allocated by the previous instruction with the same logical destination register is freed. This register is identified by the VP field of the reorder buffer. Besides, the physical register allocated by that instruction is also freed. The identifier of such register is obtained through the PMT, by indexing it with the VP register that is to be freed.

In case of a exception or a branch misspeculation, a precise state can be obtained by undoing the mappings performed by the instructions that follow the offending one. This can be done by popping out the entries of the reorder buffer from the newest until the offending one. For each instruction, the reorder buffer stores the destination logical register and the previous virtual-physical register that was allocated to it. Using the logical register identifier, the GMT is accessed and the current virtual-physical mapping is obtain. In addition, if the V flag of the GMT entry is set, the current physical mapping is also obtained. Both the current virtual-physical register and the physical register (if already allocated) are returned to their corresponding free pools. The VP register field of the GMT entry is restored with the VP field of the reorder buffer (the previous virtual-physical mapping) and the physical mapping associated to such register, if any. Such physical mapping is obtained from the PMT. If the restored virtual-physical register is mapped to a physical register, the V flag is set; otherwise it is reset.

A mechanism based on checkpointing similar to the one used by the R10000 [15] could be used to recover from branches in just one cycle.

Finally, notice that the proposed mechanism does not imply any additional delay to the critical path when compared with the traditional scheme, except for the commit, which may be delayed by one cycle due to the requirement to look up the PMT. The GMT look-up is equivalent to the traditional register mapping task. The allocation of physical registers can be performed during the last cycle of the execution so that it is available at the beginning of the write-back stage.

3.3. Avoiding deadlock

A virtual-physical register organization may be designed with any number of logical, physical and virtual-physical registers. The number of virtual-physical registers has a small impact on the hardware cost, as pointed out above. The number of logical registers is a feature of the ISA and therefore remains fixed for different implementations of the same ISA. On the other hand, the number of physical registers has a very high impact on the hardware cost as discussed in the introduction. In consequence, the number of physical registers will be lower than that of virtual-physical registers.

In this case, it may happen that when a instruction completes there is no a free physical register. The obvious approach to deal with this situation would be to squash such instruction. However, in this situation, the oldest instruction in the window would not be able to commit because when its execution completes it would also find that there is not any free physical register. Under this circumstances, no instruction would be allow to commit and therefore no physical register would be freed, which would result in a deadlock.

This deadlock can be avoided by a slight modification of the register management policy. In particular, it suffices to guarantee that a given number of the oldest instructions that have a destination register will have a physical register for renaming. Let us call this number the *number of reserved registers (NRR)*. In general, this parameter can be different for floating point and integer registers. In this way, the oldest NRR instructions that have a destination register and those instructions in between without a destination register are guaranteed to commit. Since every instruction that consumes a register frees another one when it commits, the next NRR instructions with a destination register and those instructions in between are also guaranteed to commit. Following the same reasoning it can be proved that all instructions are guaranteed to commit and therefore no deadlock occurs.

Such scheme is implemented by means of two pointers in the reorder buffer, one for integer and the other for FP instructions. Such pointers identify the oldest NRR integer/FP instructions that have a destination register and they are called PRR_{int} and PRR_{fp} respectively (see Figure 3 for an example). In addition, there are two registers that indicate the number of instructions below such pointers that have a destination integer/FP register and another two registers that indicate the number of such instructions that have already allocated a physical register. Such counters will be called Reg_{int}, Reg_{fp}, Used_{int} and Used_{fp} respectively.

Every time an instruction with an integer destination register commits PRR_{int} is moved up to the next instruction with an integer destination register. If such instruction has not yet allocated its physical register, Used_{int} is decreased; otherwise it is left unchanged. If the head of the reorder buffer is reached before finding the next instruction, then Reg_{int} is decreased. When a new instructions with a integer destination register is decoded, if Reg_{int} is lower than NRR_{int} then Reg_{int} is increased and PRR_{int} is made to point to such instruction. The same procedure is applied to instructions with an FP destination register and their corresponding pointer and counters.

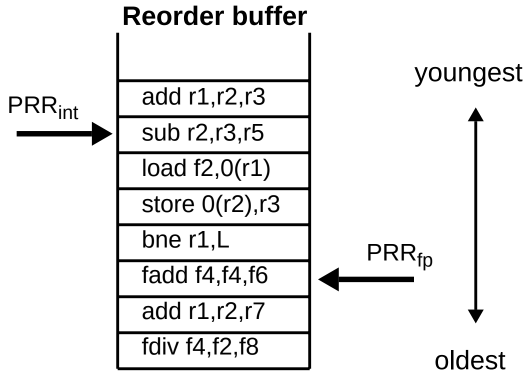

Figure 3. Example of the use of the PRR_{int} and PRR_{fp} for NRR equal to 2.

When an instruction completes, it allocates a new physical register as previously described, provided that the are more free physical registers than $\mathrm{NRR}_{int/fp}$ minus $\mathrm{Used}_{int/fp}$ or it is an instruction not youngest than the one pointed by $\mathrm{PRR}_{int/fp}$. Otherwise, the instruction is squashed and sent back to the instruction queue to be re-executed again.

NRR can take any value from 1 to the number of physical registers minus the number of logical registers. It is difficult to anticipate which is the best value without experimental evaluation. A low NRR implies that the processor has more registers to allocate on demand of completing instructions, which favor a more aggressive out-of-order execution. On the other hand, when the processor runs out of physical registers, the execution can progress using only NRR registers for renaming (those reserved for the oldest instructions) since those younger instructions that have completed and thus allocated a new physical register will not release any register until all previous instructions and themselves have committed.

To be more precise, let us suppose that NRR_{int} is equal to 1, all the instructions have a integer destination register, the number of logical registers is 32, the number of physical registers is 64 and the size of the reorder buffer is 64. Suppose that at a given time the reorder buffer is full; the oldest instruction, which has a long latency, is executing but has not completed yet; the next 32 instructions depend on the oldest one and thus have not been issued and the remaining 31 instructions (the youngest ones) have all executed and completed. Since NRR_{int} is equal to 1, the youngest 31 instructions are allowed to allocate a register when they complete since there is only one register reserved for the oldest instruction. Then, when the oldest instruction completes it allocates the reserved register. Next, it commits an frees a register that is used by the following instruction. When this instruction commits, the register that it frees can be used by the next one and so on. In consequence, until the commit point reaches the youngest 31 instructions, the remaining instruction have only one renaming register available, which forces a sequential execution.

In conclusion, avoiding to allocate some registers to some instructions that cannot issue and giving them to some younger instructions is beneficial because it allows to advance some future work. However, it penalizes the execution of the instructions in between.

Notice that having an NRR equal to the number of physical registers minus the number of logical registers, which could be considered the most conservative configuration, is expected to perform at least as well as the conventional register renaming scheme. In such scenario the virtual-physical register scheme allocates all available physical registers always to the oldest instructions, like the conventional scheme. However, the virtual-physical register scheme has important additional advantages. First, if the processor runs out of a type (integer or FP) of registers, the processor is allow to continue executing instructions of the other type, whereas in the conventional scheme the processor would stall. Second, the processor cannot complete the execution of any instruction beyond the oldest NRR with a destination register, like in the conventional scheme. However, in the virtual-physical register organization the processor is allowed to continue the fetch and decode of further instructions. Finally, those instructions without a destination register will never stall once they have their operands, even if they are beyond the PRR pointer. This may help for an earlier resolution of branch instructions.

The performance achieved by different values of NRR is experimentally evaluated in section 4.

3.4. Alternative allocation policy

One potential drawback of the virtual-register organization described above is the re-execution of instructions that do not have a physical register when they complete. An alternative solution that we have researched is based on allocating physical registers when instructions are issued instead of when they complete. In such scheme, an instruction with a destination register will be allowed to be issued if it has a physical register available. Obviously the drawback of this approach is that it reduces the register pressure when compared with the conventional scheme but not as much as the scheme based on allocating registers when the instructions complete. Section 4 compares both approaches.

4. Performance evaluation

This section presents a performance evaluation of the virtual-physical register organization. The evaluation of the new scheme is performed by comparing the execution time of an aggressive superscalar processor with a conventional register organization with that of the same processor with the virtual-physical register organization. In both cases it is assumed the same amount of physical registers.

4.1. Experimental framework

A trace-driven simulator of a realistic out-of-order superscalar processor has been developed to evaluate the proposed register organization. Two different register organizations have been simulated. The first one is the conventional register renaming scheme used by the R10000 [15] among others, which is based on a physical register file and a map table that translates logical to physical registers. The second one is the virtual-physical register organization proposed in this paper.

The processors can fetch up to eight consecutive instructions every cycle. A perfect instruction cache is assumed. Branch prediction is performed using a 2048 entry Branch History Table with a 2 bit up-down saturated counter per entry. A 128-entry instruction reoder buffer is assumed. There is one physical register file for integer data and another for FP data. Both have 16 read ports and 8 write ports. The number of physical registers has been varied from 48 to 96. Functional units are fully pipelined except for integer and FP division. Table 1 shows the number of functional units and their latency.

Functional Unit	Count	Latency
Simple Integer	3	1
Complex Integer	2	9 multiply 67 divide
Effective Address	3	1
Simple FP	3	4
FP Multiplication	2	4
FP Divide and SQR	2	16 divide

Table 1. Functional units and instruction latency.

Three cache memory ports and the memory disambiguation scheme implemented in the PA-8000 [6] have been assumed in this experiment. Up to 8 instructions can commit every cycle.

The processor has a lookup-free data cache [7] that allows up to 8 pending misses to different cache lines. The cache size is 16 KB, and it is direct-mapped with 32-byte line size. Cache hit latency is 2 cycles and the penalty for a cache miss is 50 cycles. This cache configuration is chosen to stress the penalties caused by the cache memory, as expected in future microprocessors. An infinite L2 cache is assumed and a 64-bit data bus between L1 and L2 is considered (i.e., a line transaction occupies the bus during four cycles).

Our experimentation methodology is trace-driven simulation. The object code, previously compiled with full optimization for a DEC AlphaStation 600 5/266 with a DEC 21164 processor, is instrumented using the Atom tool [13]. The instrumented program is executed and the trace generated feeds the processor simulator. A cycle-by-cycle simulation is performed in order to obtain accurate timing results. Because of the detail at which simulation is carried out the simulator is slow, so we have simulated 50 million of instructions for each benchmark after skipping the 100 million of instructions. Five floating-point (swim, hydro2d, mgrid, apsi, wave5) and four integer (go, compress, li, vortex) SPEC95 benchmarks have been selected for this study. Each program was run with the largest input set available for that benchmark.

4.2. Results

4.2.1 Write-back allocation with maximum NRR. The first experiment evaluates the performance of the virtual-physical register scheme when physical registers are allocated in the write-back stage. Sixty four physical registers (like some current microprocessors have) are considered for each register file. The NRR parameter is set to its maximum value (number of physical register minus number of logical registers: 32) since this configuration is expected to perform at least as well as the conventional scheme.

Table 2 shows the instructions committed per cycle (IPC) for the conventional and the virtual-physical schemes. It can be seen that the virtual-physical register organization provides a significant improvement for all the benchmarks. In average, the increase in IPC is of 19% (12% if the miss penalty is 20 cycles instead of 50) and it goes up to 84% for the best case. It can also be observed, that the improvement is much higher for floating point than for integer programs. Each committed instruction is executed in average 3.3 times. However, this does not hurt performance since re-executions usually spend resources that otherwise would be unused.

4.2.2 Write back allocation for different values of NRR. The next experiment evaluates the effect of the NRR parameter on the performance of the virtual-physical register organization. This parameter determines the number of oldest instructions that are guaranteed to have a physical register. This parameter can be different for integer and FP registers although we consider here the same value for both. As discussed in section 3.3, one can find reasons that favor both high and low values of NRR. For 64 physical and 32 logical registers, NRR can take any value from 1 to 32. Figure 4 shows the speedup achieved by the virtual-physical register organization when compared with the conventional one ($IPC_{virt.}/IPC_{conv.}$) for NRR equal to 1, 4, 8, 16, 24 and 32.

It can be seen in Figure 4 that there are significant differences between integer and FP programs. For the latter, the maximum NRR (32) is almost always the best, except for hydro2d that achieves the best performance for 24. The speedup obtained with NRR equal to 32 is 1.3 in average for the FP programs. Values of NRR between 16 and 24

		Conv. reg.	Virtual-physical reg.	
		IPC	IPC	imp. (%)
int	go	0.73	0.76	4
	li	0.98	1.05	7
	compress	1.75	1.84	5
	vortex	1.14	1.24	9
FP	apsi	1.37	1.76	28
	swim	1.12	2.06	84
	mgrid	1.32	2.09	58
	hydro2d	2.16	2.24	4
	wave5	1.64	1.71	4
	harmonic mean	1.23	1.46	19

Table 2. Instruction completion rates of the conventional and virtual-physical register organizations.

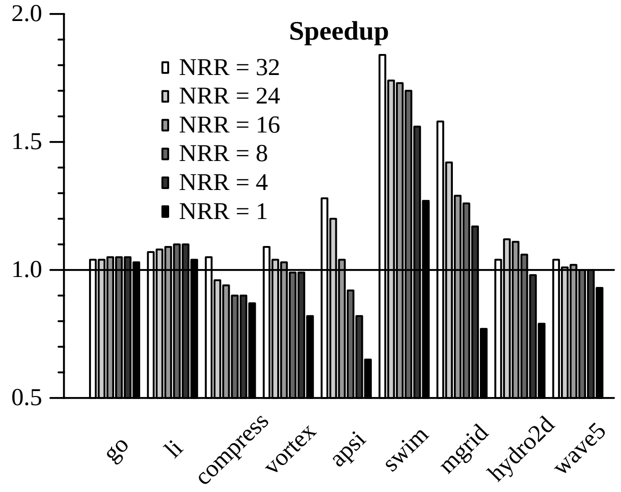

Figure 4. Speedup of the virtual-physical register organization with register allocation at write-back.

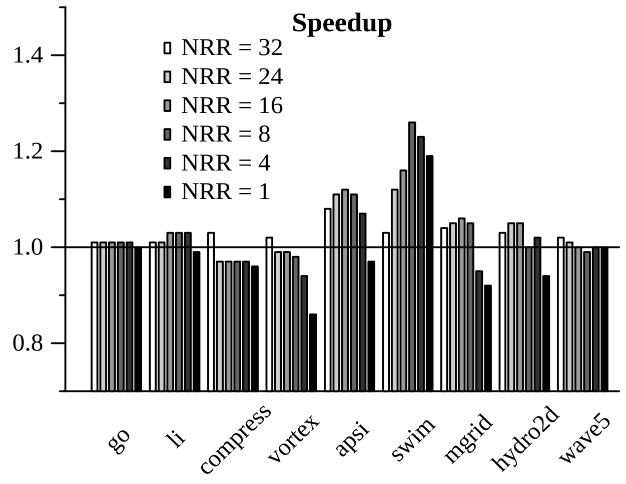

Figure 5. Speedup of the virtual-physical register organization with register allocation at issue.

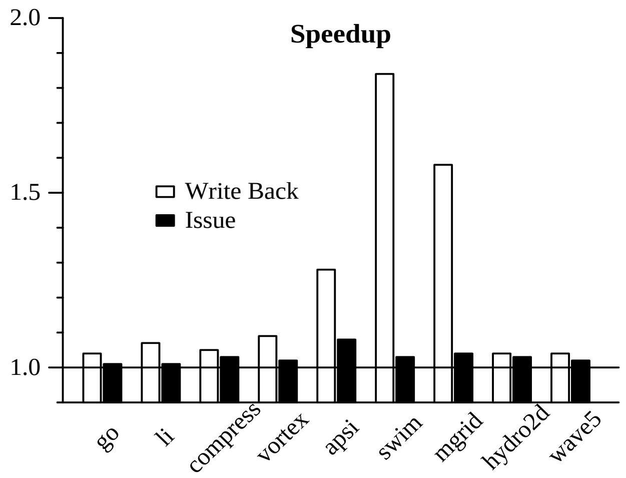

Figure 6. Write-back versus issue register allocation.

provide a better performance than the conventional scheme for all FP programs but the performance decreases when NRR decreases. Finally, very small values of NRR are not adequate for any FP programs. In this case, the virtual-physical register organization can perform worse than the conventional scheme. It is remarkable the good performance of the new scheme for the swim program for any value of NRR. The speedup for this benchmark range from 1.27 to 1.84.

For integer benchmarks the speedup of the virtual-physical register scheme is lower but still significant. In this case only NRR equal to 32 provides an improvement for all the benchmarks. Decreasing NRR provides a slight improvement for two programs (go and li) but a significant detriment for the other two.

4.2.3 Issue allocation versus write-back allocation. As discussed in section 3.4, an alternative implementation of the virtual-physical register organization could allocate the physical registers in the issue stage instead of the write-back. This will avoid re-executions of instructions but will not be as effective to reduce the register pressure. Figure 5 shows the speedup of the virtual-physical register organization with this alternative register allocation scheme. In this case, the optimal value of NRR is 32 (24 has the same average performance), which provides an improvement of 4% over the traditional register mapping scheme.

Figure 6 compares the two alternative schemes to allocate physical registers in the virtual-physical register organization. In each case, the optimal value of NRR observed in the previous experiments is considered (32 for both). It can be seen that allocating registers in the write-back stage significantly outperforms the other scheme.

4.2.4 Varying the number of physical registers. We have also evaluated the virtual-register scheme for a different number of physical registers. In addition to the size of each register file previously considered (64) we have also evaluated the performance of the virtual-physical register organization for 48 and 96 registers. Figure 7 shows the IPC of the virtual-physical organization with allocation in the write-back stage and NRR equal to 16, 32 and 64 respectively, compared with that of the conventional renaming scheme. It can be seen that the virtual-physical organization always outperform the conventional one. The improvement increases when the number of physical register decreases, as one could expect since the new organization reduces the register pressure. In average, the improvement of the virtual-physical scheme is 31%, 19% and 8% for 48, 64 and 96 registers respectively.

Another conclusion that can be drawn from these results is that the virtual-physical register organization can reduce the size of the register file without penalizing performance when compared to the conventional scheme. For instance, the average IPC of the virtual-physical register organization with 48 registers (1.17) is about the same as that of the conventional scheme with 64 registers (1.23); thus, the new organization provides a 25% register saving.

5. Conclusions

We have presented a novel register renaming scheme for dynamically scheduled processors. The key idea behind the new organization is to delay the allocation of physical registers, instead of doing it in the decode stage, in order to reduce the register pressure.

The new scheme is based on introducing a new concept that is called virtual-physical registers. Virtual-physical registers are not related to any storage location but they

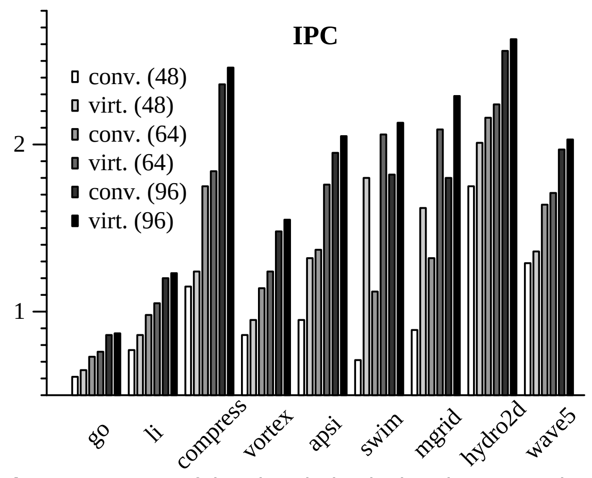

Figure 7. IPC of the virtual-physical register organization and the conventional renaming scheme for a varying size of each register file.

are merely tags that are used to keep track of register dependences.

We have investigated two alternative realizations of the virtual-physical register scheme that differ in the time when physical registers are allocated. We have shown that the scheme that allocates them in the write-back stage is more effective than the scheme that allocates them in the issue stage, in spite of the large number of instruction re-executions that the former scheme implies. Besides, both schemes outperform the traditional register renaming organization.

We have also researched the most critical design parameter of the novel organization, that is the number of oldest instructions in the instruction window that are guaranteed to have a physical register. This feature is necessary to avoid deadlocks in a precise exception processor.

We have shown that the new renaming approach provides significant improvements for a different number of physical registers. When compared with the conventional scheme, the virtual-physical registers provides an increase in IPC of 31%, 19% and 8% for 48, 64 and 96 physical registers. In general, the improvement for FP programs is higher than for integer benchmarks. We have also shown that the new scheme with 48 registers provide about the same performance than the traditional one with 64.

Finally, it is important to point out that the benefits of reducing the register pressure can be even much more beneficial for future architectures with a larger instruction window and thus, a much higher register pressure. For instance, we believe that in the context of multithreaded architectures the benefits of the virtual-physical register organization will be more important than those observed for a superscalar processor. We plan to explore this scenario in future work.

Acknowledgments

We would like to thank Jim Smith for his comments and suggestions on early versions of this paper.

This work has been supported by the Spanish Ministry of Education under grant CICYT TIC 429/95 and the Direcció General de Recerca of the Generalitat de Catalunya under grant 1996FI-03039-APDT.

The research described in this paper has been developed using the computing resources of the CEPBA.

References

[1] K.I. Farkas, N.P. Jouppi and P. Chow, “Register File Design Considerations in Dynamically Scheduled Processors”, in *Proc. of the Int. Symp. on High Perf. Computer Architecture*, pp. 40-51, 1996.

[2] L. Gwennap, “Intel’s P6 Uses Decoupled Superscalar Design”, *Microprocessor Report*, 9(2), Feb. 1995

[3] L. Gwennap, “HAL Reveals Multichip SPARC Processor”, *Microprocessor Report*, 9(3), March 1995

[4] L. Gwennap, “Digital 21264 Sets New Standard”, *Microprocessor Report*, 10(14), Oct. 1996

[5] J.L. Hennessy and D.A. Patterson, *Computer Architecture. A Quantitative Approach*. Morgan Kaufmann Publishers, San Francisco CA, 1996.

[6] D. Hunt, “Advanced Performance Features of the 64-bit PA-8000”, in *Proc. of the CompCon’95*, pp. 123-128, 1995.

[7] D. Kroft, “Lockup-free instruction fetch/prefetch cache organization”, in *Proc. 8th International Symposium on Computer Architecture* (1981) pp. 81-87

[8] M. Moudgill, K. Pingali and S. Vassiliadis, “Register Renamin and Dynamic Speculation: an Alternative Approach”, in *Proc. of Int. Symp. on Microarchitecture*, pp. 202-213, 1993.

[9] J.E. Smith and A.R. Pleszkun, “Implementing Precise Interrupts in Pipelined Processors”, *IEEE Tranactions on Computers*, 37(5), pp. 562-573, May 1988.

[10] J.E. Smith and G.S. Sohi, “The Microarchitecture of Superscalar Processors”, *Proceedings of the IEEE*, 83(12), pp. 1609-1624, Dec. 1995.

[11] G.S. Sohi, “Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers”, *IEEE Transactions on Computers*, 39(3), pp. 349-359, March 1990

[12] S.P. Song, M. Denman and J. Chang, “The PowerPC 604 Microprocessor”, *IEEE Micro*, 14(5), pp. 8-17, Oct. 1994

[13] A. Srivastava and A. Eustace, “ATOM: A system for building customized program analysis tools” , in *Proc of the 1994 Conf. on Programming Languages Design and Implementation*, 1994.

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

[15] K.C. Yeager, “The MIPS R10000 Superscalar Microprocessor”, *IEEE Micro*, 16(2), pp. 28-40, April 1996