Dynamic Register Renaming Through Virtual-Physical Registers

Teresa Monreal[†]
Antonio González*
Mateo Valero*
José González^{††}
Víctor Viñals[†]

TMONREAL@POSTA.UNIZAR.ES
ANTONIO@AC.UPC.ES
MATEO@AC.UPC.ES
JOSEG@DITEC.UM.ES
VICTOR@POSTA.UNIZAR.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 dynamic register renaming scheme that delays the allocation of physical registers until a late stage in the pipeline. We show that it can provide important savings in number of physical registers so it can significantly shorter the register file access time. Delaying the allocation of physical registers requires some artifact to keep track of dependences. This is achieved by introducing the concept of virtual-physical registers, which are tags that do not require any storage location. The proposed renaming scheme 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 restricted 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 (Wall, 1993). However, a large instruction window has some implications in other critical parts of the microarchitecture, such as the complexity of the issue logic (Palacharla, Jouppi & Smith, 1997) and the size of the physical register file (Farkas, Jouppi & Chow, 1996). 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 (Farkas, Jouppi & Chow, 1996). Since the current trend of increasing the issue width

[†]Departamento de Informática e Ing. de Sistemas. Centro Politécnico Superior - Univ. de Zaragoza, Zaragoza, Spain.

^{*}Departament d' Arquitectura de Computadors. Universitat Politècnica de Catalunya, Barcelona, Spain.

^{††}Departamento de Ingeniería y Tecnología de Computadores. Universidad de Murcia, Murcia, Spain.

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 (Tullsen et al., 1996). 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 5.1 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%.

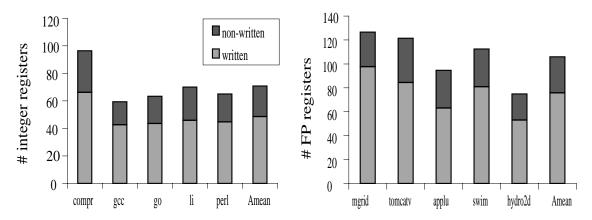


Figure 1. Register usage (written + non-written) and register waste (non-written) due to early allocation.

This work presents a novel register renaming 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 (González, González & Valero, 1998; González et al., 1997) proposed this scheme and referred to it as virtual-physical registers, and later we extended the scheme with a novel register allocation approach (Monreal et al., 1999).

We show in this paper that virtual-physical registers 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 conventional register renaming scheme. The virtual-physical renaming scheme is presented in Section 3. Two different variations of the scheme are analyzed in Section 4, which differ in the approach taken when there are not free physical registers. Section 5 analyzes the performance of the

virtual-physical renaming scheme. Finally, Section 6 outlines some related work and Section 7 summarizes the main conclusions of this work.

2. Register Renaming

Register renaming was first implemented for the floating-point unit of the IBM 360/91 (Tomasulo, 1967). 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 (Smith & Pleszkun, 1998). 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 youngest 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. In particular, the two following approaches are the most common solutions to provide the rename storage locations:

- The entries of the reorder buffer (Sohi, 1990). 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 (Song, Denman & Chang, 1994)).
- 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 (Gwennap, 1995), the PowerPC 604 (Song, Denman & Chang, 1994), and the HAL SPARC64 (Gwennap, 1995). The MIPS R10000 (Yeager, 1996), and the DEC 21264 (Gwennap, 1996) 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 this scheme are presented and then, its implementation is detailed.

3.1 Motivation

The motivation for the proposed register renaming approach 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.

Note 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 (Moudgill, Pingali & Vassiliadis, 1993; Smith & Sohi, 1995), 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 logical register has been subsequently renamed to another physical register. In order to support precise exceptions, the register cannot be freed until the last instruction that reads it commits.

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 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 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 £2, 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 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.

Note 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

This 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 2.a). The first one is called the *general map table (GMT)*, it 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 second one 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 the 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 (Gwennap, 1996) to implement the logical to physical map table.

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

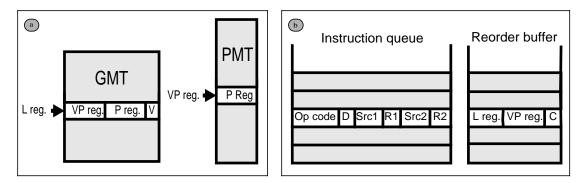


Figure 2. Tables required by the virtual-physical register organization. The instruction queue and the reorder buffer.

The GMT has NLR rows of $\lceil \log_2(NVR) \rceil + \lceil \log_2(NPR) \rceil + 1$ bits each, where NLR is the number of logical registers, NVR is the number of virtual-physical registers and NPR is the number of physical registers. The PMT has NVR rows of $\lceil \log_2(NPR) \rceil$ bits each or NPR rows of $\lceil \log_2(NVR) \rceil$ 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 due to 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 and 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, 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.b):

- *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.b):

- 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 (solutions 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 used to update the GMT. The entry corresponding to the logical destination register is checked for a match with the VP register identifier, and if so, 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 misprediction, 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 (Yeager, 1996) could be used to recover from branches in just one cycle.

Finally, note 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 they are available at the beginning of the write-back stage.

3.3 Alternative Allocation Policy

One potential drawback of the virtual-register organization described above is the reexecution of instructions that do not have a physical register when they complete, as it is
described in the next section. 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, a ready instruction with a destination register will be allowed to be issued only
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. Another hybrid
solution that we have evaluated is based on allocating registers for high latency instructions
(loads) when they complete and for the rest of instructions when they are issued. Although
the number of re-executions decreases with this hybrid solution, they usually produce a
slight decrease in instructions committed per cycle (IPC), except for some cases in FP
benchmarks such as *swim* for which it increases IPC by 4% (assuming 64 physical registers
in each file).

4. 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 and cycle time 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.

We have proposed to alternative approaches to avoid this deadlock, which are described in the following sections.

4.1 Reserving Registers for the Oldest Instructions

Our first solution to avoid deadlock consists of 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 refer to this parameter as 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. We will refer to this register allocation scheme as *vp-NRR*.

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

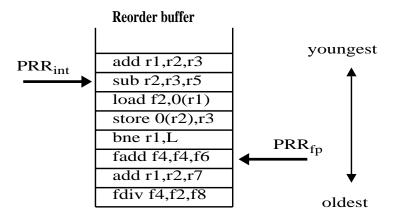


Figure 3. Example of the use of the PRR_{int} and PRR_{fp} for NRR equal to 2.

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} and Use_{int} are decreased. When a new instructions with an 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.

When an instruction completes, it allocates a new physical register as previously described, provided that the are more free physical registers than NRR_{int/fp} minus Used_{int/fp} or it is an instruction not younger than the one pointed by 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 an 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. Suppose also that all 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, the remaining instructions have only one renaming register available, which forces a sequential execution, until the commit point reaches the youngest 31 instructions.

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

Note that having an NRR equal to the number of physical registers minus the number of logical registers 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, it is allowed 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 instructions 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 of the processor is very sensitive to the value of NRR. For instance, Figure 4 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 5.1. Note that the optimal value for NRR for li is 16 whereas for applu 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 6, and it is one of the schemes that we use for comparison in this work.

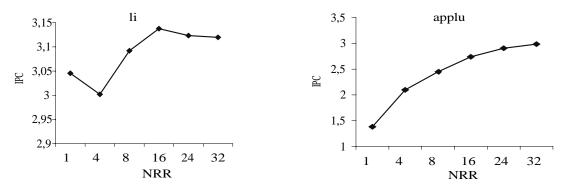


Figure 4. Effect of varying NRR for two particular programs.

For the vp-NRR register allocation scheme, 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 is critical and this is what has motivated the next approach.

4.2 Stealing Registers from Younger Instructions

The 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 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 [9], 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).

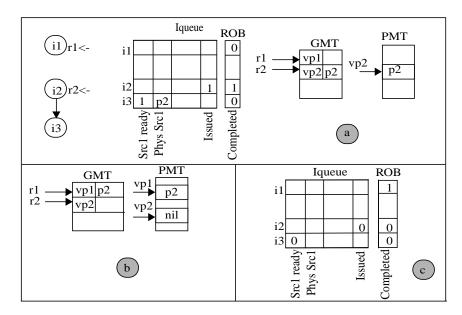


Figure 5. Example of virtual-physical renaming with DSY allocation.

4.2.1 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 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 VP₁, VP₂ and P₂ respectively (see Figure 5.a). When i_1 steals the physical register of i_2 , the PMT table is updated to reflect that now VP₁ is mapped to P₂ and VP₂ is not longer associated to P₂ (see Figure 5.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 commit, 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 5.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 the 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 (VP₂ in the example) is broadcast to the instruction queue in order to mark as NOT ready any matching source operand (Figure 5.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, their results 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.

4.3 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 on 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 there are more instructions older than it in the instruction window requiring 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 advanced execution of some instructions have two important benefits:

 An advanced execution of a branch instruction will validate its prediction and in case of misprediction the recovery actions will start sooner (wrong path squashing and correct path fetching). In the conventional renaming scheme, such validation would occur much later, since the instruction would be delayed by the lack of

^{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.

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.

The effect of the advanced execution of instructions is analyzed in section 5.3.

Finally, note that the latency as well as the resource consumption of re-executed instructions can be significantly reduced by means of a reuse mechanism (Sodani & Sohi, 1997). 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 them. In this way, re-executed instructions would not increase the contention in the functional units. In our analysis, such an instruction reuse mechanism is not considered.

5. Performance Evaluation

5.1 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 (Burger & Austin, 1997). The sim-outorder simulator has been modified to include physical register files (integer 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 (Kessler, 1999). 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	
Funtional 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-assoc., 64 byte lines, 12 cycles hit time	
Main memory	unbounded 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 for Fortran codes and -O4 -migrate for C codes).

Table 2. Benchmarks

Program	Input	#dyn. inst.(M)
compress	40000 e 2231	170
gcc	genrecog.i	145
go	99	146
li	7 queens	243
perl	scrabbl.in	47
mgrid	test (modifying the two first lines to 5 & 18)	169
tomcatv	test	191
applu	train (modifying dt=1.5e-03; nx=ny=nz=13)	398
swim	train	431
hydro2d	test (modifying ISTEP=1)	472

5.2 Performance Statistics

Figure 6 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 renaming schemes are compared: the conventional one (conv), virtual-physical registers with the NRR deadlock-avoiding approach described in Section 4.1 (vp-nrr), and virtual-physical registers with the DSY approach presented in Section 4.2 (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 NRR and the DSY allocation policies is noticeable, DSY provides an average speed-up of 2% and 7% for integer and FP codes respectively.

For the DSY configuration, 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.

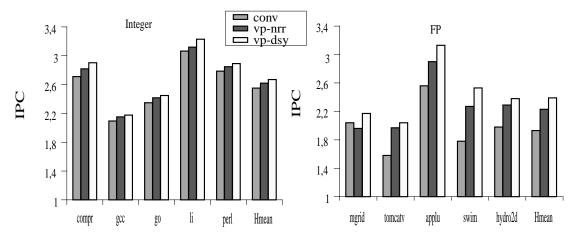


Figure 6. Performance of the virtual-physical renaming versus the conv. scheme for a 64 physical registers. Two different deadlock-avoiding approaches are shown for virtual-physical renaming: the NRR and the DSY.

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 7 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 vp-NRR deadlock-avoiding scheme as well as the DSY one are shown. Virtualphysical registers with DSY is always better than virtual-physical registers with NRR, 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 unbounded 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 (Farkas, Jouppi & Chow, 1996).

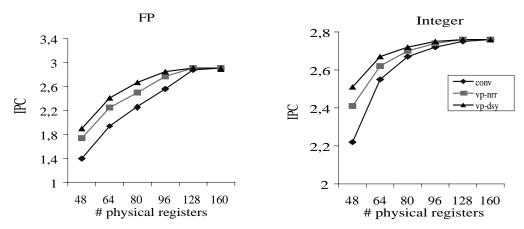


Figure 7. Performance as a function of number of physical registers for the conventional register organization and the virtual-physical scheme with the two deadlock-avoiding approaches.

5.3 Quantifying the Advanced Execution of Instructions

As described above, the advanced execution of instructions have important benefits. In this section, we present statistics about this effect for the DSY deadlock-avoiding approach.

We consider that one instruction is advanced when it is executed in a situation where the conventional renaming scheme would not be able to execute it. In particular, we say that an instruction is executed in advance when at the time it is issued, the processor has a number of older instructions in the window such that in the conventional scheme they would consume all physical registers. In this scenario, the conventional scheme would not be able to execute this instruction.

Assuming a register file with 64 physical registers, we evaluate the percentage of advanced instructions and the number of cycles that they are avanced as follows:

- If an instruction finds more than 32 instructions older than it in the window that require a physical register, we considered that this instruction is advanced and we record this cycle (the *advanced-cycle*).
- In the first cycle that this advanced instruction has less than 32 older instructions in the window that require a physical register (this will always occur before the advanced instruction commits), we considered that this is the earliest cycle that this instruction could be executed in the conventional renaming scheme and thus, we record this cycle (the *conventional-cycle*).
- When an advanced instruction commits, the number of cycles that this instruction is advanced is computed as the difference between the *conventional-cycle* and the *advanced-cycle*.

Figure 8.a shows the percentage of loads that miss in cache and the percentage that miss and are advanced. We can observe a reasonable degree of load-misses that are advanced (20% on average for integer benchmarks and 86% for FP benchmarks). There are some remarkable cases such as *tomcatv*, which also obtains the best speed-up (it advances the 99% of loads that miss in cache). Figure 8.b also depicts the percentage of conditional branches that are mispredicted and the percentage that are mispredicted and advanced. We can see that in average 6% of mispredicted conditional branches are advanced for integer benchmarks and 74% for FP benchmarks. *Tomcatv* is again the most benefited by instruction advancement since 97% of the mispredicted conditional branches are advanced.

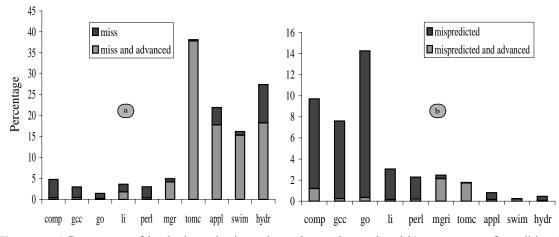


Figure 8. a) Percentage of loads that miss in cache and are advanced and b) percentage of conditional branches that are mispredicted and advanced.

Another interesting statistic is the average number of cycles that an instruction is advanced. We have measured that load instructions are advanced in 9.4 cycles for integer benchmarks and 26.9 cycles for FP benchmarks (these average figures include only advanced instructions). For conditional branches these numbers are 4.8 and 28 respectively. For divide, multiply and simple instructions these numbers are 2.6, 3 and 6.1 for integer benchmarks and 13.3, 13.9 and 17.1 for FP benchmarks, respectively.

We can conclude that the virtual-physical register renaming causes that some instructions can enter the instruction window and be executed earlier than what the conventional scheme can achieve. This benefit is much more important for FP programs than for integer ones, as it is confirmed by the statistics above: FP programs have a larger percentage of advanced instructions and on average they are advanced by more cycles than for integer programs. This is explained by the fact that integer programs have many mispredicted branches, and this limits the number of instructions that can be in-flight at any given time. Virtual-physical registers allow the processor to have more in-flight instructions than the conventional scheme for a given number of physical registers. However, branch mispredictions prevent the processor from taking advantage of this feature in some cases.

6. Related Work

An alternative approach to delaying the allocation of physical registers was proposed by Wallace and Bagherzadeh (1996). 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.

The Power3 implements what they call virtual renaming (Bose & Moreno, 1999; Song, 1997). Like the PowerPC 620 (Levitan, Thomas & Tu, 1995), 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. (1998). 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.

7. 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.

Two different deadlock-avoiding approaches have been proposed. A comparison among them shows that the on-demand allocation with stealing from the younger (DSY) is the most effective since it provides maximum look-ahead when ILP is limited but this look-ahead never penalizes older instructions.

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.

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. We would like to thank Jim Smith for many useful comments on this work.

References

- P. Bose and J. Moreno (1999). Private communication.
- D. Burger and T.M. Austin (1997). The SimpleScalar Tool Set v2.0. Technical report 1342, University of Wisconsin-Madison, CS Department.
- K.I. Farkas, N.P. Jouppi and P. Chow (1996). Register File Considerations in Dynamically Scheduled Processors. In *Proc. of 2nd. Int. Symp. on High-Performance Computer Architecture*, pp. 40-51.
- A. González, J. González and M. Valero (1998). Virtual-Physical Registers. In *Proc. IEEE* 4th. Int. Symp. on High-Performance Computer Architecture, pp. 175-184.
- A. González, M. Valero, J. González and T. Monreal (1997). Virtual Registers. In *Proc. of Int. Conf. on High-Performance Computing*, pp. 364-369.
- L. Gwennap (1996). Digital 21264 Sets New Standard. Microprocessor Report, 10(14).

- L.Gwennap (1995). HAL Reveals Multichip SPARC Processor. Microprocessor Report, 9(3).
- L. Gwennap (1995). Intel's P6 Uses Decoupled Superscalar Design. *Microprocessor Report*, pp. 9-15.
- S. Jourdan, R. Ronen, M. Bekerman, B. Shomar and A. Yoaz (1998). 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.
- R.E. Kessler (1999). The Alpha 21264 Microprocessor. IEEE Micro, 19(2):24-36.
- D. Levitan, T. Thomas and P. Tu (1995). The PowerPC 620 Microprocessor: A High-Performance Superscalar RISC Microprocessor. In *Proc. of 40th. IEEE Computer Society International Conference*, pp. 285-291.
- T. Monreal, A. González, M. Valero, J. González and V. Viñals (1999). Delaying Physical Register Allocation Through Virtual-Physical Registers. In *Proc. of 32st. Int. Symp. on Microarchitecture*, pp. 186-192.
- M. Moudgill, K. Pingali and S. Vassiliadis (1993). Register Renaming and Dynamic Speculation: an Alternative Approach. In *Proc. of Int. Symp. on Microarchitecture*, pp. 202-213.
- A.S. Palacharla, N.P. Jouppi and J.E. Smith (1997). Complexity-Effective Superscalar Processors. In *Proc. of 24th. Int. Symp. on Computer Architecture*, pp. 206-218.
- J.E. Smith and A.R. Pleszkun (1998). Implementing Precise Interrupts in Pipelined Processors. *IEEE Transactions on Computers*, 37(5), pp. 562-573.
- J.E. Smith and G.S. Sohi, "The Microarchitecture of Superscalar Processors", *Proceedings of the IEEE*, 83(12), pp. 1609-1624, Dec. 1995.
- A. Sodani and G.S. Sohi (1997). Dynamic Instruction Reuse. In *Proc. of 24th. Int. Symp. on Computer Architecture*, pp. 194-205.
- G.S. Sohi (1990). Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers. *IEEE Transactions on Computers*, 39(3), pp. 349-359.
- S.P. Song (1997). IBM's Power3 to Replace P2SC. Microprocessor Report, 11(15): 23-27.
- S.P. Song, M. Denman and J. Chang (1994). The PowerPC 604 Microprocessor. *IEEE Micro*, 14(5), pp. 8-17.
- R.M. Tomasulo (1967). An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1), pp. 25-33.
- D.M. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo and R. Stamm (1996). 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.
- D.W. Wall (1993). Limits of Instruction-Level Parallelism. Technical Report WRL 93/6 Digital Western Research Laboratory.

- S. Wallace and N. Bagherzadeh (1996). A Scalable Register File Architecture for Dynamically Scheduled Processors. In *Proc. 1996 Conf. on Parallel Architectures and Compilation Techniques*, pp. 179-184.
- K.C. Yeager (1996). The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2), pp. 28-40.