Reducing Register Pressure Through LAER Algorithm

Gao Song

National ASIC Design Engineering Center
Institute of Automation, Chinese Academy of Sciences
PO Box 2728, Beijing, 100080, P.R. China

brandon_198102@yahoo.com.cn

Abstract

When modern processors keep increasing the instruction window size and the issue width to exploit more instruction-level parallelism (ILP), the demand of larger physical register file is also on the increase. As a result, register file access time represents one of the critical delays and can easily become a bottleneck. In this paper, we first discuss the possibilities of reducing register pressure by shortening the lifetime of physical registers, and evaluate several possible register renaming approaches. We then propose an efficient dynamic register renaming algorithm named LAER (Late Allocation and Early Release), which can be implemented through a two-level register file organization. In LAER renaming scheme, physical register allocations are delayed until the instructions are ready to be executed, and the physical registers in the first level are released once they become non-active, with the values backupped in the second level. We show that LAER algorithm can significantly reduce the register pressure with minimal cost of space and logic complexity, which means the same amount of ILP exploited with smaller physical register file, thus shorter register file access time and higher clock speed, or the same size of physical register file to achieve much higher performance.

Keywords: dynamic register renaming, register pressure, out-of-order, superscalar processors.

1 Introduction

Modern dynamically scheduled superscalar processors achieve high performance by aggressively exploiting available instruction-level parallelism (ILP) from applications. They examine a large instruction window to find multiple ready and independent in-flight instructions that can be executed in parallel, and issue them out-of-order (Smith and Sohi, 1995). The size of the instruction window of wide issue processors has a deterministic effect on the amount of ILP that can be exploited, since a larger instruction window contains more instructions to be checked for parallel execution. However, supporting a large instruction window also increase the requirement for the structure size of other parts of the microprocessors, such as issue queue, reorder buffer (ROB), and physical register file (Farkas, Jouppi and Chow, 1996). In this work we are mainly concerned with the problems caused by a large physical register file.

In a RISC-like instruction set architecture (ISA), nearly all of the instruction operands reside in the register file. The size of the register file directly determines the number of in-flight instructions and hence the instruction window size, since every dispatched instruction with a destination register has to be assigned a new physical register file. The dispatch stage will be stalled when free registers run out, which would degrade IPC by forcing the processor to look for ILP within a smaller window until oldest instructions commit and release their registers.

However, the access time of a register file is significantly affected by its size, as well as its number of ports (Farkas et al. 1996). Since wider issue width inevitably leads to more read and write ports, a larger register file will have a longer access time, which tends to be significantly longer than the latency of other common operations, such as integer additions (Cruz, Gonzalez, Valero and Topham, 2000). Therefore, it's very likely that the register file access time will become one of the longest delays of forthcoming processors. Under this scenario, it will severely compromise the processor clock cycle time and has a detrimental impact on performance, unless the register file is pipelined. However, pipelining a register file has serious side effects of its own. First of all, a register file with multi-cycle access time would increase branch misprediction penalty since a misprediction would be discovered more cycles later. It would also increase register file pressure by prolonging register lifetime. What's more, pipelined register file would require multi-level bypassing logic among the function units, thereby increasing the bypassing logic complexity and the bypassing delay, another cycle time critical path (Balasubramonian, Dwarkadas and Albonesi, 2001). All these issues are critical for processor performance. Finally, pipelining the register file, a heavily ported RAM structure, is not a trivial task itself.

Due to these constraints, the physical register files in modern superscalar processors have been very modestly sized (64 in MIPS R10000, and 80 in Alpha 21264), although larger size obviously results in improved IPC, even beyond 128 entries (Farkas et al.1996).

However, it is not difficult to find that current dynamic superscalar processors are using much more physical registers than strictly needed to store temporary values of programs. In another word, the physical registers are being wasted. This is a logical result of current register renaming scheme, where a physical register is allocated

too early at the decode stage, and released too late when next instruction with the same logical destination register commits. An illustration example and a detailed statistics of this waste are presented in Section 3.1.

In this work, we first give a discussion of possible solutions to the problem of the physical register waste in current dynamic processor, and then present a novel dynamic register renaming scheme with two-level hierarchical register file organization, named LAER (Late Allocation and Early Release) algorithm, which can effectively shorten the life time of physical registers and hence, reduces the register file pressure. In the LAER algorithm, physical registers in the first level (L1) are not allocated until the instructions that write them are ready to be issued for execution, and are released back to free list when they are no longer needed except in the event of a recovery from a mispredicted branch or an exception. The values they contain are backupped in the second level (L2).

We show in this paper that LAER algorithm can provide significant savings in number of physical registers in L1, which implies an increase in instruction throughput due to shorter cycle time, resulting from shorter register file access time, or more ILP to exploit without an increase in register file size. What's more, it doesn't require much extra space and logic, and has little impact on other parts of the microarchitecture.

The organization of this paper is as follows. Section 2 reviews the conventional register renaming scheme. Section 3 gives the analysis and statistics of physical register waste in current scheme and discusses possible solutions to the problem. LAER renaming scheme is presented in detail in Section 4. In section 5 we evaluate the performance of our proposal. Finally, Section 6 outlines some related work, and concluding remarks of this work are made in Section 7. The main references are listed in Section 8.

2 Register Renaming

First implemented in IBM 360/91 (Tomasulo, 1967), register renaming was introduced to remove name dependences among register operands (WAR and WAW), thus allowing more ILP to be extracted within the given instruction window. It is a key issue for the performance and has become a common technique employed by modern dynamic superscalar processors.

There are mainly 2 methods to implement dynamic register renaming (Smith and Sohi, 1995), differing in the locations used as rename storage.

In the first, there is a physical register file called architectural register file, with the same size as the logical register file (the register set defined in the ISA), and the customary one-to-one relationship is maintained. The renamed register values are stored in entries of a buffer typically referred to as a reorder buffer (ROB), which provides one entry for every in-flight instruction. The ROB is a structure used to maintain proper instruction ordering for precise exceptions by keeping instructions committing in program order (Smith and Pleszkun, 1988). When an instruction with a destination register commits, the result value is written from ROB entry to architectural register file. The source operands are read from architectural register file or from a ROB entry when an instruction is issued to function units. This is the method used in the Intel Pentium Pro (Gwennap, 1995).

The second method of renaming is using physical registers to store uncommitted register values. The physical register file can be separated from the architectural register file, and values are copied to architectural register file when instructions commit (Levitan, 1995), similar to the ROB approach. Another more popular approach is using a larger merged physical register file, which contains both committed and uncommitted values. In this scheme, the operands are always read from and written to the merged physical register file. By means of a mapping table, the value of each logical register is associated with a physical register. When an instruction is decoded, its source logical registers are translated to their current physical mapping using the mapping table. If this instruction has a logical destination register, a new physical register will be obtained from the free list (the physical registers are not currently mapped to any logical register) and assigned to this logical register, and the corresponding entry of mapping table is updated to reflect the new mapping. The previous physical register mapped to this logical destination register is recorded in the ROB entry of this instruction and can not be released back to free list until this instruction commits from the ROB, in case that a branch misprediction or an exception occurs before this instruction commits, in which case this previous physical register is restored to the mapping table and becomes the current mapping again.

The merged physical register file scheme simplifies the operand fetch task and eliminates value transfer between register files or between ROB and register file, and has become the most popular method widely used in latest high performance processors, such as MIPS R10000 (Yeager, 1996) and Alpha 21264 (Kessler, 1999). In this paper, we focus on this renaming scheme and refer to it as the conventional renaming scheme.

3 Motivation and Solution Discussion

3.1 Motivation

A severe physical register waste can be observed in the conventional renaming scheme described in the previous section, because the physical registers are allocated much before the values they need to store become available, and released much after all their consumers have already read the values from them. This can be illustrated by a simple example. Suppose the following FP code:

Original code	**Renamed code**
1) Lr2 <- mem	**Pr1 <- mem**
2) Lr8 <- Lr2 * Lr4	**Pr2 <- Pr1 * Pr12**
3) Lr2 <- Lr8 / Lr6	**Pr3 <- Pr2 / Pr16**
4) Lr8 <- Lr2 + Lr6	**Pr4 <- Pr3 + Pr16**

In a four-way or even wider width superclaclar processor, these four instructions can be fetched and decoded in the same cycle. When they are decoded, say in cycle 0, four free physical registers Pr1, Pr2, Pr3 and Pr4 are assigned to logical registers Lr2 and Lr8, and the logical registers of source operands are translated to the their physical mappings using the mapping table, as shown above. Assume that Pr12 and Pr16 are the current physical mappings of Lr4 and Lr6 respectively, and both contain available values.

Let's assume that instruction 1 (inst1) is a load that can start its execution in cycle 1 but produces a cache miss. Assume also that the latencies of cache miss, FP multiplication, FP division and FP addition are 20, 10, 20 and 5 cycles respectively.

In the conventional renaming scheme, Pr1 can't be released until inst3 commits. That will be in cycle 51 if these four instructions can be issued once their operands are available and can commit in the next cycle as soon as they complete execution. Hence Pr1 will be used for 52 cycles, and similarly Pr2 will be used for 57 cycles. Notice that all the physical registers are allocated in cycle 0, but Pr1 and Pr2 are not written until cycle 20 and cycle 30, and become useless in cycle 21 and 31 when the consumers have read the values. That's to say, they are actually needed in 2 cycles but are amazingly wasted for the rest 50 and 55 cycles. What's more, unlike our assumption, instruction commit is usually many cycles after the execution completes, due to uncompleted older instructions, especially long latency loads from memory. Therefore, for the code given above, the waste of physical registers could be even more severe than as described here.

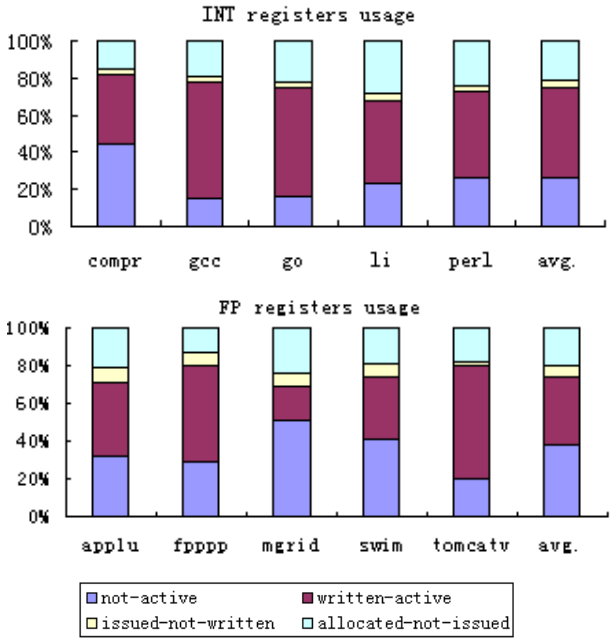

Figure 1: the usage of physical registers in the conventional renaming scheme.

To be more accurate and general, we made a statistics on the physical registers usage of the conventional renaming scheme on 10 selected Spec95 benchmarks, provided that the maximum physical register requirement is meet, in our experiments that is 160 physical registers for both integer and FP. Figure 1 shows the percentages of physical registers that are wasted because of early allocation (allocated-not-issued + issued-not-written) and late release (not-active). In this paper we say a physical register in use is not active when it has been written, all the instructions that need the value it contains (we call these instructions consumers) have already read from it, and the logical register it's assigned to has been remapped. For more details about the evaluation framework refer to Section 5. Obviously, only the physical registers in the written-active state are strictly needed by the processors to store temporary values. On average, the early allocation and late release of physical registers increase the register requirements by 54% and 53.5% respectively for integer, and by 72.6% and 104.5% for FP. For some programs such as mgrid these figures can be as high as 173% and 290%.

As shown above, early allocation and late release both have severe effects on physical register requirements. While there has been previous work addressing each issue, an ideal solution to this problem needs to reduce the register waste caused by both, so that the register pressure can be reduced to the minimum.

3.2 Analysis and Solution Discussion

To summarize, in the conventional renaming scheme, the physical registers perform 3 functions:

1) Track data dependences. Logical registers are renamed at decode stage to physical registers to remove name dependences in the instruction window. Then true data dependences are identified and maintained through the physical registers

2) Store temporary values. Once the values are written into the physical registers, they stay there waiting for their consumers to read.

3) Maintain proper states. The physical registers are still occupied when they become non-active because they are needed to hold the old values of the logical registers, which are part of the processor state that has to be well maintained in case of branch mispredictions or exceptions. They are freed when the state they maintain is sure to be changed, that is, when the next instruction with the same destination logical register commits from the ROB.

Among these 3 functions, the first and the third are exactly the reasons for early allocation and late release respectively. So what is needed to reduce register pressure is to shift these 2 responsibilities from physical registers.

Shifting the third one can be easily handled through a two-level hierarchical register file organization, which has already been proposed and evaluated in previous work (Balasubramonian et al. 2001). The first level (L1) only contains active values, while non-active values are backupped in the second level (L2) in case that they will be needed in the event of a branch misprediction or an

exception, in which case they would be copied back to L1 and become active again.

A counter can be associated with each physical register and keeps track of its pending consumers. The counter is incremented each time a source register is renamed to the physical register and decremented each time a consumer issues and reads the value (Smith and Sohi, 1995). The condition to do the backup safely is that the physical register has been written, the counter has been decremented to zero, and the corresponding logical register has been remapped, which ensures that there won't be new consumers to come.

As for the task of tracking data dependences, what's actually needed by the processor is just an identifier for register operands. We can assign a tag or a virtual identifier for each new decoded instruction that has a destination register. The virtual identifiers can be used to keep track of data flow and do not require any storage location. The physical registers then can be allocated at some time later.

The early allocation waste would be decreased to zero if a physical register were not allocated until the instruction completes execution and the value to be stored becomes available, and this is the approach used by Monreal, Gonzalez and Valero (1999). However, with a two-level organization, this could be troublesome. When an instruction finishes its execution but there is no free physical register available, it can be squashed and send back to issue queue to be re-issued, or steal a physical register from a younger instructions that has completed execution and been assigned a physical register (Monreal et al. 1999). In the latter method, the instruction from which the physical register is stolen also has to be squashed for re-issue. However, if the squashed instruction has triggered one or more backups when it decremented corresponding counters at issue, not only the counters have to be incremented back, but also the backupped values need to be copied back to L1. Besides extra logic to implement physical register stealing, this will add to the control logic complexity of backupping and may delay the re-execution of the squashed instruction because of the latency of copying value back to L1. In addition, frequent re-execution will increase function unit contention and extra logic is needed to reset consumers of the re-issued instructions in the issue queue as not ready.

To avoid re-execution, the value of the stolen can be stored in a reuse buffer, which can be implemented by part of the entries in L2, and the value is copied back to L1 when a free L1 is available. We simulated this scheme in our early experiments but found it not feasible, for it adds to the complexity of value moving logic and the port requirements between the 2 levels. In addition, when an instruction reaches the head of ROB but with its result still in reuse buffer, and physical registers in L1 are used up, a deadlock would occur since the commit stage would be permanently stalled, unless it steals a physical register from younger instructions. However, this would result in a more complex stealing logic, and the commit stage would be frequently delayed.

Given all these considerations, we decide to allocate physical register when the instruction has all its operands ready and the required function unit is free, that is to say, to allocate at issue. This approach simplifies the logic of backupping and operands broadcasting, and will not cause any increase in function unit contention and commit delay. The only drawback is that the early allocation waste cannot be reduced as much as the scheme based on allocating at write-back stage. However, as can be seen from Figure 1, the loss will be very small since the number of physical registers occupied by instructions in execution (issued-not-written) doesn't contribute a lot to the total waste, only about 6.7% for integer and 10% for FP. This is because long latency operation instructions usually represent a small fraction of executed instructions, while long chains of dependent instructions are far more common in the code, and this is exactly what makes instructions waiting for cycles in the issue queue.

Having decided to perform allocation at issue, we need to handle two more problems that do not exist in the conventional renaming scheme or a two-level register file organization that only eliminates late release waste. In these schemes, physical registers are allocated at the decode stage and the instructions are assigned physical registers in program order, that is, older instructions have higher priority to get physical registers. When free physical registers run out, the dispatch stage will be stalled until new physical registers are freed. However, in current dynamic superscalar processors the instructions in the issue queue can be issued out-of-order. In this case, it may happen that when the instruction at the head of ROB eventually has all of its operands ready and the required function unit is free, the physical registers have been used up by younger instructions. Under this circumstance, no instructions would be able to commit and therefore no physical register would be released, which would result in a deadlock. This situation is avoided in LAER scheme by using two counters that respectively keep the track of the number of free L1 and how many in-flight instructions in the issue queue require a L1 (we call them L1 needers). The issue of an instruction would be stalled if it has more older L1 needers than free L1. Though the processor's ability of out-of-order issue will be weaken when they fall short of free L1, this stall wouldn't be long since new L1 will be freed when old instructions commit. What's more, as will be shown in Section 5.2, there would be much less issue stalls than the dispatch stalls there would be when L1 physical registers are allocated at the decode stage, since the L1 requirements are reduced by great amount. In addition, different from dispatch stall, it is not the whole issue stage that is stalled, but only part of the young instructions in the issue queue.

Also generated from the out-of-order characteristics of issue, the second problem has to be addressed is that, in a two-level register file organization that doesn't delay the allocation (Balasubramonian et al. 2001), when the backupped value is to be reinstated to L1 in a recovery event, it can be safely copied back to the L1 physical register where it previously stayed, but this can not be ensured in a scheme like LAER, as illustrated here.

Assume instruction A writes its result to L1 physical register X, a subsequent instruction B writes to the same logical register, and then the value in X is backupped to L2 and X is released. If a branch misprediction makes it necessary to reinstate the value produced by A, it would have to come between A and B. In this case, B and all its successors would be squashed and the L1 they use, if any, would be released. If the allocation of L1 happens at decode stage, X can only be assigned to the instructions younger than B. Hence X would always be free after a recovery event, whether it is re-allocated after the backup or not, so it can still be used to contain the reinstated value, and the checkpointed register mapping table (Yeager, 1996) need not be modified. However, if L1 is allocated at issue stage as in LAER, it's likely that X is mapped to an instruction older than the mispredicted branch, or even older than A. Therefore, a new L1 needs to be allocated to reinstate the value, and it isn't necessary that a free one can always be found.

If the allocation failed, in the LAER algorithm, the value remains in L2 and the pending copy would be recorded in a structure named waiting list (we call it WL for short). The fetch and dispatch from the right target resume as normal, and the older instructions keep going ahead in the ROB, except that new dispatched instructions that need the values of the pending copy would have to wait for their operands. The value would be reinstated to L1 once free L1 is available. We will show in Section 5.2 that the reinstate of backupped values wouldn't be delayed often, less than 2% on a base model.

While the discussion above has already given an outline of the LAER renaming scheme, its implementation is detailed in the next section.

4 LAER Algorithm

4.1 Dynamic LAER Register Renaming

In this section we describe the LAER implementation in the order as an instruction goes down the pipeline. The register file organization is shown in Figure 2, which can be used for both integer and FP renaming. The function units read all their operands from and write all their results to L1.

1) Fetch.

As in the conventional renaming scheme, instructions are fetched from I-cache to the fetch queue at this stage, and branches are predicted.

2) Decode & Dispatch.

At this stage, what is allocated is a tag named virtual register instead of a physical register, which is assigned to the virtual register at instruction issue. A virtual register is not related to any storage location and can be occupied until the time it can be safely freed. The register renaming is performed through two mapping tables. The first one is called the logical-to-virtual mapping table (LVMT), which is indexed by the logical register number and each entry contains the current virtual register to which the corresponding logical register is mapped. The second one is the virtual-to-physical mapping table (VPMT). It has

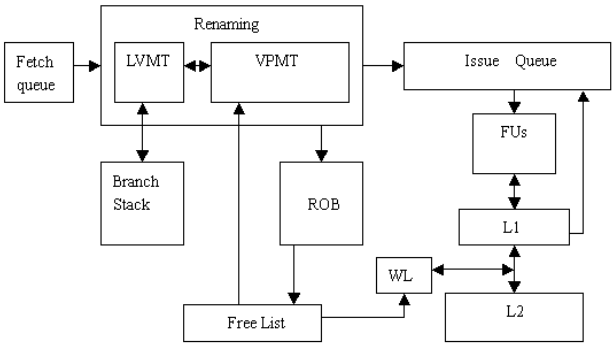

Figure 2: the physical register file organization of LAER renaming scheme.

an entry for very virtual register, and contains the following five fields, as shown in Figure 3:

- V bit: indicates whether the L1 physical register mapped to this virtual register contains a valid value.
- B bit: indicated whether the value is backupped in L2.
- Preg: the current physical register to which the virtual register is mapped, if any. This field specifies a L1 physical register if the V bit is set, and a L2 physical register if the B bit is set.
- C bit: indicates whether the virtual register is a current mapping, that is, whether the logical register it is assigned to has been remapped to another virtual register.
- Cnt: the counter that indicates the number of pending consumers in the window.

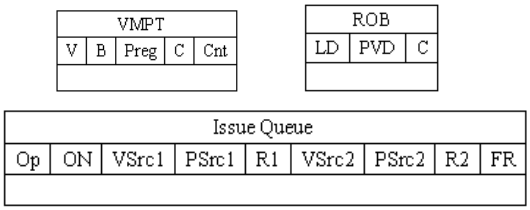

Figure 3: The VPMT, the issue queue and the ROB.

When an instruction is decoded, if it has a destination register, it will be assigned a new virtual register, and the corresponding entry in the VPMT resets all the fields except the C bit. Since they don't require any storage location, the number of virtual registers can be set to meet the maximum requirement of a given instruction window size, namely the number of logical registers plus the window size (Monreal et al. 1999). The corresponding entry of the LVMT is then updated to indicate the new logical-to-virtual mapping. The previous virtual register is kept in ROB, and the C bit of its entry in VPMT is reset.

For each source register operand, the LVMT is looked up to get the current virtual mapping. For each virtual source register, the counter in the corresponding entry of the VPMT is incremented to indicate the new consumer. If the V bit in the entry is set, the logical source register is

renamed to the L1 physical register specified in the Preg field; otherwise it is renamed to the virtual register.

If this instruction is a branch, the LVMT is checkpointed and the current mappings are saved in a branch stack to enable a quick recovery in the case of a misprediction, like in the MIPS R10000 design (Yeager, 1996).

Besides, there is a counter named L1-needers, which keeps the track of the number of the un-issued in-flight instructions that need L1. When an instruction with a destination register is decoded, it will record the current value of L1-needers, which indicates how many older instructions need L1 since instructions are decoded in order. Then, the counter is incremented to indicate this new L1 needer.

When all these have been done, the instruction is dispatched to the issue queue, where is waits until it is issued, and the ROB, where it remains until it is committed.

3) Issue & Execution.

An entry of the issue queue contains the following fields, as shown in Figure 3:

- Op code: the operation code.
- VD: the virtual destination register identifier, if any.
- ON (older needers): number of un-issued older in-flight instructions that need L1.
- VSrc1 and VSrc2: the virtual register identifiers of the two source operands (assume they are all registers).
- PSrc1 and PSrc2: the source operands in the form of L1 physical registers.
- R1 and R2: each one indicates whether a source operand is ready.
- FR: indicates whether the required function unit is free.

An entry of the ROB has the following fields:

- LD: the logical destination register.
- PVD: the previous virtual mapping of the logical destination register.
- C bit: indicates whether the instruction has completed its execution.

When an instruction has its R1, R2 and FR fields all set, it attempts to get a free L1. The value of in the ON field is compared with another counter named free-L1, which keeps the track of the number of free L1 physical registers. If the latter one is larger, a free L1 will be allocated and the instruction is then issued to the corresponding function unit, with the assigned L1 physical register. The Preg field in the corresponding entry of the VPMT is also updated to reflect the new virtual-to-physical mapping. For the issued instruction, the VPMT is looked up to decrement the consumer counters of the operand producers, using the virtual registers specified in the VSrc fields.

After issued, the instruction is executed in the function unit until completion, unless a branch misprediction or an exception occurs.

4) Write-Back.

When the instruction completes its execution, it sets the V bit in the corresponding entry of VPMT, and the C bit in the corresponding entry of the ROB. Then the virtual destination register is broadcasted to the issue queue with its L1 mapping. If there is a match in a VSrc field whose R bit is not set, the corresponding PSrc field is updated with the L1 physical register and the R bit is set.

5) Commit.

When an instruction commits from the ROB, the previous virtual register mapped to the same logical destination register, specified in the PVD field of the ROB entry, is released. Its physical mapping specified in the Preg filed of the corresponding VPMT entry, if any, is also released. The physical register to be freed is in L1 if the V bit is set, and in L2 if the B bit is set.

In the whole process, when an entry of VPMT has its V bit set, C bit reset, and its counter has been decremented to zero, a L2 physical register is allocated to backup the value in the L1 physical register specified by the Preg field. Then the L1 is released and the Preg field is modified to indicate the new mapping in L2. If no free L2 is found, the value remains in L1 and no backup happens.

In the case of a branch misprediction, a quick recovery can be achieved by restoring the LVMT with the checkpointed mappings stored in the branch stack. While the logical-to-virtual mappings are not affected by the backups and needn't be changed, the VPMT has to be scanned for every virtual register in the restored LVMT. If the B bit of an entry is set, a free L1 has to be allocated to copy the backupped value back. If the value is reinstated successfully, the V bit of the entry is set, and the Preg field is updated with the new L1 mapping. If there is no free L1 available, the virtual register is added to the WL (waiting list), where it waits until a new L1 is freed. The new fetched instructions that need this value as source operand also have to wait.

The popping of the WL is triggered by a free L1 flag, which is set when the free list of L1 physical register is not empty. When the value is copied back to a L1 physical register, the virtual register and the L1 physical register are broadcasted to the issue queue to set the R bits of waiting consumers, just like the result broadcasting at the write-back stage.

Since the value with more pending consumers should have higher priority to be reinstated, the counter of each virtual register can be also recorded in WL, and be incremented synchronously with the one in the VPMT. When a new released L1 triggered a WL popping, it is assigned to the virtual register with the largest counter value. However, this needs more space and logic to implement synchronous counters and priority arbitration. We employ a simple stack structured WL in our scheme since the experiments show that the reinstate failure is a rare happening.

Besides restoring the LVMT, all instructions in the ROB following the mispredicted branch are squashed. The virtual registers assigned to them are released back to the virtual register free list by restore their read pointer, like the mechanism used in MIPS R1000 (Yeager, 1999). Also released are their physical mappings, if any.

A recovery would also be required in the event of an exception. Unlike recovering from an incorrect branch, the register mappings are usually gradually undone by traversing the ROB in reverse order from the youngest instruction to the exception one. For each entry, its virtual destination register is obtained by looking up the LVMT using the logical destination register specified in the LD field. The virtual register and its physical mapping, specified in the corresponding entry of the VPMT, if any, are all released. Then the LVMT entry is restored with the previous virtual mapping specified in the PVD field of the ROB entry. Most processors handle exception in such a simple but slow way of ROB traversing, since exceptions are not as frequent as branch misprediction. In the LAER renaming scheme, there would be backupped values that need to be reinstated to L1 during this process, and this can potentially add to the latency of the recovery. What's more, there may be many surplus copies during the traversing. We reduce the delay and unnecessary copies by utilizing the branches in the ROB, similar to the measure taken by Balasubramonian et al. (2001).

Since branches come far more frequently than exceptions (usually fewer than 10 instructions apart (Wall, 1993), especially in integer programs), when an exception occurs there may be some branch instructions following it. We first perform a branch recovery using the checkpointed mappings of the branch immediately following the exception instruction. Then the ROB entries between this branch and the exception instruction are squashed and the mappings are undone gradually as described above.

Due to the value reinstate and the ROB traverse, the recovery process may be slow and would take several cycles. However, this process can be overlapped with the new instruction fetching and decoding, and it would be several cycles before the new fetched consumers need the backupped value, especially when instructions need to be fetched from the new target of a mispredicted branch. So it is not expected to be a large overhead.

4.2 Complexity of LAER algorithm

Since the access time is determined by the size of L1 when its ports number is set by the issue width and the number of the function units, the L2 can contain more entries than L1 to ensure the minimal waste in L1, and total number of the physical registers in 2 levels has no need to exceed the sum of the number of the logical registers and the size of ROB.

The inter-level bandwidth can be set to the minimum since the backupping can be in parallel with other work, and the reinstates are only required in a recovery event.

A number of extra structures are introduced in the proposed scheme, namely a virtual-to-physical mapping table, a waiting list, and two counters.

The VPMT has NVR row of ($\lceil \log_2 (\text{NPR}) \rceil + \lceil \log_2 (\text{ROB\_size}) \rceil + 3$) bits each, where NVR is the number of virtual registers, equal to the number of logical registers plus ROB_size, and NPR is the maximum of the number of L1 physical registers and the number of L2 physical registers.

To store the virtual register identifiers of pending copies, the WL needs much less entries than the L2 since the reinstate failure is a rare happening, and $\lceil \log_2 (\text{NVR}) \rceil$ bits are needed for each entry. As for the value broadcasting when popping WL, it can share the broadcasting logic used at write-back stage.

As for the two counters, the L1-needers and the free-L1 require $\lceil \log_2 (\text{IQ\_size}) \rceil$ and $\lceil \log_2 (\text{NL1PR}) \rceil$ bits respectively, where IQ stands for the issue queue and NL1PR is the number of physical registers in L1.

The LAER renaming scheme also requires the issue queue to contain 3 more fields, namely the ON and the 2 VSrc fields, meaning $\lceil \log_2 (\text{IQ\_size}) \rceil + 2 \lceil \log_2 (\text{NVR}) \rceil$ extra bits for each entry.

Besides the extra structures and fields, there is a slight bit overhead in the entries of the LVMT, the issue queue and the ROB, since there are more virtual registers than physical registers.

Finally, it should be noted that the issue process might be slower than that in the conventional scheme, since it includes more tasks. Besides monitoring the R bits and the FR bit, the value of the ON field is compared with the free-L1 counter, and a L1 physical register needs to be allocated if the counter is larger.

5 Performance Evaluation

5.1 Experimental Framework

The LAER renaming scheme has been evaluated in detail using the MASE (Micro Architecture Simulation Environment) simulator (Larson, Chatterjee and Austin, 2001) of SimpleScalar tool set. The simulator has been modified to include split integer and FP physical register files where both of the renamed values and the committed values are stored, instead of storing them in the ROB entries and then committing them to an architectural register file. We also modelled a split integer and FP issue queue, a mapping table, a free list, a busy bits table, and a branch stack, similar to the MIPS R1000 design (Yeager, 1996), and the load/store operation handling has been modified in more detail and accuracy. We refer to this modified simulator as a conventional processor model, called Conv. for short. The main simulation parameters are summarized in Table 1. Then, the Conv. simulator was further extended to model our LAER proposal, with the same simulation parameters. We refer to it as LAER.

In order to insure high comparability of the simulation results of Conv. and LAER, we modified and debugged them in great efforts to eliminate all the checker errors, and finally the two simulators we used for experiments ran all the given test programs in the tool set and all the selected benchmarks with zero checker error.

Parameter	Value
Fetch width	8 (up to 2 branches)
Issue width	8 (out-of-order issue)
ROB size	128
Issue Queue Size	64 int/64 FP
Load/Store Queue	64 entries with store forwarding, no blind speculation
Branch Predictor	2K BTB with 2-bit counters
Branch Stack	Infinite entries (always enough)
L1 I/D-cache	32KB, 2-way set-assoc, 32/64 byte lines, 1 cycle hit time, LRU
L2 unified cache	1MB, 2-way set-assoc, 64 byte lines, 12 cycle hit time, LRU
I/DTLB	30 cycles miss latency
Memory	8 bytes width, 50 cycles access time
Function units (latency)	8 ialu (1); 4 imult (7); 8 fpalu (4); 4 fpmult/fpdiv (10/18); 4 memport (1)
Physical registers	48 / 64 / 80 / 96 / 128 / 160

Table 1: Simulation parameters.

As benchmarks, we selected 10 programs from the Spec95 suite, five integer programs (compress, gcc, go, li and perl) and five FP programs (applu, fpppp, mgrid, swim and tomcatv). To reduce simulation time, they were not simulated to completion. For every benchmark, we select a representative segment of 50 million instructions. The simulations were first fast-forwarded from the beginning until 1 million instructions before the selected segments, and then these 1 million instructions were simulated in detail to prime all structures. All statistics started from the first cycle of the execution of the selected segments. Details of the benchmarks are listed in Table 2. We used the precompiled Spec95 codes provided in the SimpleScalar tool set.

5.2 Performance Results

We evaluated the LAER scheme with three register file configurations. The first one has 48 entries in L1 and 64 in L2, named LAER-base. The second one is called LAER-medium and the numbers of the physical registers in L1 and L2 are 64 and 80. The last one, LAER-max, has the most aggressive configuration, that is, 80 entries for both L1 and L2. All the three models have the minimal bandwidth between L1 and L2, namely one read port and one write port.

Figure 4 shows the IPC for each benchmark, assuming 48 physical registers in Conv. and in the L1 of LAER (LAER-base) for both the integer and FP register files.

As can be seen, the LAER-base significantly outperforms the Conv. with 48 physical registers, providing an average speed-up of 7.9% and 61.4% for integer and FP programs

Benchmarks	Input (ref)	Simulation Window	IPB
compress	Bigtest.in	250M~300M	5.09
Gcc	toplev.i	50M~100M	4.81
Go	9stone21.in	250M~300M	6.84
Li	*.lsp	250M~300M	4.17
Perl	scrabbl.in	200M~250M	5.25
applu	Applu.in	250M~300M	25.2
fpppp	natoms.in	250M~300M	50.3
mgrid	mgrid.in	250M~300M	62.6
Swim	swim.in	250M~300M	9.7
tomcatv	tomcatv.in	300M~350M	5

Table 2: Benchmarks. IPB stands for Instructions Per Branch. A high IPB indicates a low branch rate.

respectively. As expected, FP codes benefit much more from the LAER renaming scheme than integer codes, since FP programs in general impose much higher pressure on physical registers. This is due to the different behaviours of these applications. As can be seen in Table 2, FP programs usually have a lower branch rate, which in turn means lower branch miss rate, and thus the instruction window is usually filled up. In addition, there are more long latency operations in FP codes, and therefore instructions spend more time waiting for the operands and waiting to commit. All these contribute to a higher register requirement.

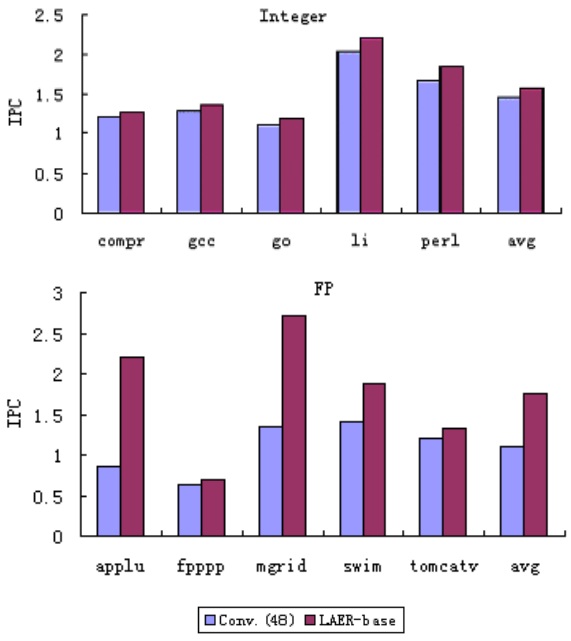

Figure 4: Performance of Conv. (48) versus LAER-base, which has also 48 entries in L1.

As a more general comparison, Figure 5 illustrates how the performance varies as a function of the number of physical registers in Conv. and in the L1 of LAER.

As the figure shows, with only 48 physical registers in L1,

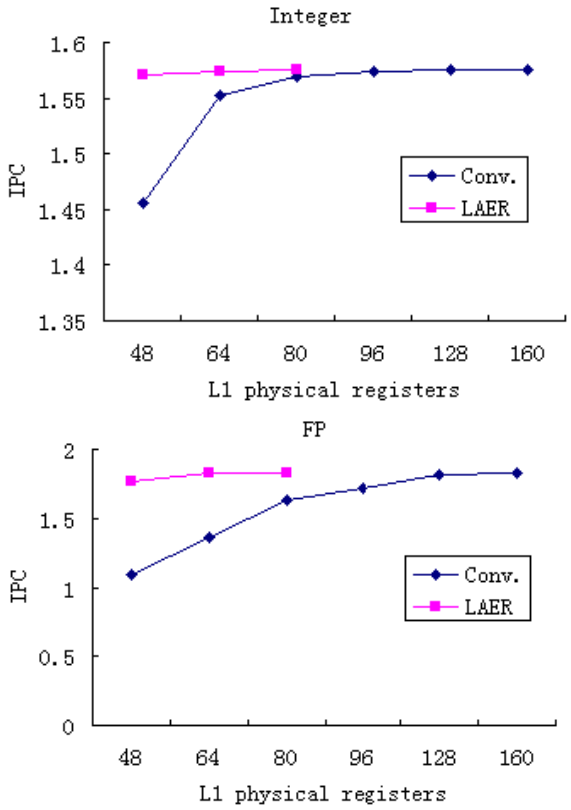

Figure 5: Performance as a function of the number of L1 physical registers for Conv. and LAER.

LAER-base has already achieved significant high performance. With respect to the maximum IPC achieved by an unbounded size register file (160 for Conv. here), it has a slight IPC degradation of only 0.3% for integer and 3.4% for FP. With more adequate physical registers, LAER-medium has a negligible IPC degradation of 0.05% and 0.01%. As expected, the performance achieved by LAER-max is equal to or even higher than the maximum value.

As a good candidate for a processor design that attempts to find the best trade-off between the number of physical registers and performance, LAER-medium can achieve the performance that would require 118 integer and 158 FP physical registers in the conventional renaming scheme, providing a saving in L1 of 46% and 60% respectively. This directly translates to significant savings in the register file access time (Farkas et al. 1996).

As for the issue stalls and the reinstate failures of the backupped values, we show the statistics of various stalls of each benchmark on LAER-medium in Table 3. As the table shows, with 64 and 80 physical registers in L1 and L2 respectively, LASE-medium experiences few issue stalls. Actually, given the same number of physical registers in L1, LAER has much less issue stalls than the dispatch stalls in the Conv., as shown in Figure 6. What's more, an issue stall in LAER means an instruction is stalled for issue; while a dispatch stall in Conv. implies the stall of the whole decode stage.

Also can be seen in Table 3, the value transfer between the two levels are rarely blocked. On average, less than 1% of the integer backups and 3% of the FP backups fail due to full L2, since we provide plenty entries in L2.

Benchmarks	Issue stall rate	Backup failure rate	Reinstate failure rate
compress	0%	0.72%	0%
gcc	7.49%	0.04%	0%
go	9.3%	0.07%	0.01%
li	1.85%	0%	0%
perl	1.03%	0.18%	0%
applu	4.48%	7.56%	0%
fpppp	6.2%	1.42%	0%
mgrid	1.77%	0.42%	0%
swim	6.51%	0%	0%
tomcatv	10.36%	0.46%	0%

Table 3: Statistics of the LAER-medium, the benchmarks are simulated in the selected segments.

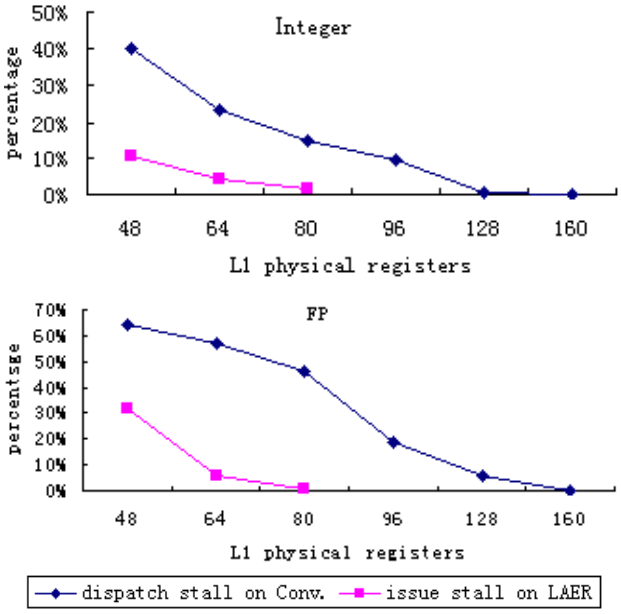

Figure 6: The issue stalls in LAER vs. the dispatch stalls in the Conv.

When the values need to be copied back to L1, LAER-base experiences a reinstate failure rate of 0.44% and 1.64% for integer and FP respectively, while LAER-medium and LAER-max almost have no trouble for this, implying a low requirement on the WL size.

6 Related Work

The organization of register file has been extensively researched in the previous work.

Monreal et al.(1999) proposed a renaming scheme that delays the physical register allocation until the write-back stage. A stealing from younger instructions is performed when the allocation fails, and the instruction whose physical register has been stolen is sent back to the issue queue to be re-executed.

A two-level register file that attempts to reduce the waste caused by late release was proposed by Balasubramonian et al. (2001). The physical registers are released in a manner similar to that in LAER scheme, but the value transfer between the 2 levels is simpler and easier because of the in-order physical register allocation at the decode stage. This work also address the problem of register file ports pressure by banking the register file.

Alternative approaches to delay allocation and bank register file have been proposed by Wallace and Bagherzadeh (1996), and Cruz et al. (2000).

A number of work has also focused on the hierarchical or banked register file organization in the context of VLIW processors (Capitanio, Dutt and Nicolau 1992, Llosa, Valero and Ayguad´e 1994 1995, Zalamea, Llosa, Ayguade and Valero 2000).

7 Conclusion

This paper tackles the problem of the increasing impact on performance of the register file access time. We present a novel dynamic register renaming scheme with a two-level register file organization, named LAER algorithm, where the register pressure is reduced by eliminating the physical register waste caused by both early allocation and late release.

We show that LAER algorithm can provide significant savings in the number of physical registers directly accessed by the function units, reducing the register pressure by 46% and 60% for integer and FP respectively. This translates into shorter register file access time, more ILP and hence higher performance.

In future work, we would like to improve the scheme in terms of complexity and its impact on other parts of the microarchitecture, and have a more detailed evaluation of the area and power consumption of the proposed register file organization. We also hope to have a quantified comparison between our LAER renaming scheme with other related proposals.

8 References

Balasubramonian, R., Dwarkadas, S. and Albonesi, D.H. (2001): Reducing the Complexity of the Register File in Dynamic Superscalar Processors. *Proc. MICRO-34.*

Capitanio, A., Dutt, N. and Nicolau, A. (1992): Partitioned register files for VLIWs: A preliminary analysis of tradeoffs. *Proc. MICRO25*, pp. 292–300.

Cruz, J., Gonzalez, A., Valero, M. and Topham, N. (2000): Multiple-banked register file architectures. *Proc. 27th Annual International Symposium on Computer Architecture.*

Farkas, K.I., Jouppi, N.P. and Chow, P. (1996): Register File Considerations in Dynamically Scheduled Processors. *Proc. International Symposium on High-Performance Computer Architecture*, pp. 40-51.

Gwennap, L. (1995): Intel's P6 Uses Decoupled Superscalar Design. *Microprocessor Report*, pp. 9-15.

Kessler, R. (1999): The Alpha 21264 microprocessor. In *IEEE Micro*,19(2), pp.24–36.

Llosa, J., Valero, M., Fortes, J. and Ayguad´e, E. (1994): Using Sacks to organize register files in VLIW machines. *CONPAR 94- VAPP VI.*

Llosa, J., Valero, M. and Ayguad´e, E. (1995): Non-consistent dual register files to reduce register pressure. *Proc. 1st International Symposium on High Performance Computer Architecture,* pp. 22–31.

Levitan, D. et al. (1995): The PowerPC 620 microprocessor: a high performance superscalar RISC microprocessor. *Proc. COMPCON*, pp. 285-291.

Larson, E., Chatterjee, S. and Austin, T.M. (2001): MASE: A Novel Infrastructure for Detailed Microarchitectural Modeling. *Proc. ISPASS-2001.*

Monreal, T., Gonzalez, A., Valero, M., Gonzalez, J. and Vinals, V. (1999): Delaying Physical Register Allocation through Virtual-Physical Registers. *Proc. MICRO-32*, pp. 186-192.

Smith, J.E. and Pleszkun, A.R. (1988): Implementing Precise Interrupts in Pipelined Processors. *IEEE Transactions on Computers*, vol. 37, pp. 562-573.

Smith, J.E. and Sohi, G.S. (1995): The Microarchitecture of Superscalar Processors. *Proc. the IEEE,* 83(12), pp. 1609-1624.

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

Wall, D.W. (1993): Limits of Instruction-Level Parallelism. Technical Report. WRL 93/6 Digital Western Research Laboratory.

Wallace, S. and Bagherzadeh, N. (1996): A Scalable Register File Architecture for Dynamically Scheduled Processors. *Proc. 1996 Conf. On Parallel Architectures and Compilation Techniques*, pp. 179-184.

Yeager, K.C. (1996): The MIPS R10000 Superscalar Microprocessor. In *IEEE Micro*, 16(2), pp. 28-40.

Yung, R. and Wilhelm, N.C. (1995): Caching Processor General Registers. *Proc. International Conference on Circuits Design*, pp. 307-312.

Zalamea, J., Llosa, J., Ayguade, E. and Valero, M. (2000): Two-Level Hierarchical Register File Organization for VLIW Processors. *Proc. MICRO-33*.