Out-of-order Predicated Execution with Translation Register Buffer

Amaury Darsch, André Seznec

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Out-of-order Predicated Execution with Translation Register Buffer

Amaury Darsch and André Seznec

N°5011

Novembre 2003

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--5011--FR+ENG

Out-of-order Predicated Execution with Translation Register Buffer

Amaury Darsch * and André Seznec †

Thème 1 — Réseaux et systèmes
Projet CAPS

Rapport de recherche n°5011 — Novembre 2003 — 26 pages

Abstract: New generation superscalar processors combine predication with large resources. A typical example is the EPIC architecture as defined by the canonical IA64 ISA. Unlike traditional ISAs, these new instruction sets are resistant to an out-of-order execution engine, because of the resource size as well as the complexity of executing predicated instructions. In this paper, we present a novel register management policy that facilitates the out-of-order execution of a fully predicated ISA. For this purpose, a new mechanism, called Translation Register Buffer (TRB) is introduced. A translation register acts as an intermediate register that associates a logical register with a physical register. By providing an indirect access to a logical register, the translation register supports the side effects induced by the cancellation of instructions by predicates. In order to demonstrate the TRB validity, a complete simulation framework that fully supports the IA64 ISA has been designed. This original implementation features an emulator and a cycle accurate 10-stage out-of-order core simulator. Our simulation results indicate that, on average, a 10% performance increase can be achieved, as compared with the equivalent in-order EPIC architecture.

Key-words: CAPS, IA64, IPF, IAOO, ITANIUM, Simulator, Emulator, Out-of-order execution, predication

(Résumé : tsvp)

CAPS IA64

* amaury.darsch@irisa.fr
† andre.seznec@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Exécution dans le désordre avec les Registres de Translation

Résumé : Les processeurs de nouvelle génération combinent la prédication avec de larges ressources. L'exemple le plus typique est le processeur EPIC de la famille IA64. Par rapport aux approches traditionnelles, les processeurs à jeu d'instructions prédiquées, sont résistants aux mécanismes d'exécution dans le désordre. Dans ce papier, nous présentons un nouveau mécanisme de gestion de registres qui facilite l'exécution dans le désordre des jeux d'instructions prediquées. Ce nouveau mécanisme dénomé *Registre de Translation* fonctionne comme un register intermédiaire qui transforme un registre logique en un registre physique. L'introduction d'un niveau intermédiaire permet de resoudre les problèmes du à l'annulation des instructions par les prédicats.

Mots-clé : CAPS, IA64, IPF, IAOO, ITANIUM, Simulateur, Émulateur, Exécution dans le désordre, Prédication

1 Introduction

For the past decade, out-of-order superscalar processor architectures have been extensively studied and several successful products have demonstrated the benefits of such architectures [7]. Recently, ISAs featuring large register file and predicated instructions [2] have been considered for general purpose computing. The EPIC IA64 typically illustrates this trend. Large register file gives more room to compiler optimizations while predicated instructions permit to perform *if-conversion* [9, 8, 4, 14].

Unfortunately, an out-of-order microarchitecture is extremely sensitive to the number of ISA registers, whereas executing efficiently predicated instructions with an out-of-order microarchitecture is still an open issue. Therefore, the Itanium I and II processors [6] that implement the EPIC IA64 ISA have been released with an *in-order* execution microarchitecture.

There has been several proposals to address the problem of out-of-order execution of predicated instructions [11, 5], but they do not consider the interaction that exists between the register file size and the number of source or destination operands. Furthermore, while many discussions around the execution of predicated ISAs have focused on physical register renaming, most of them have neglected the difficulties associated with the instruction scheduling as well as the data propagation in the bypass network.

In this paper, we attack these difficulties (predicated ISA, large register file and complex instructions) by introducing a novel register management policy based on *translation register buffer* or TRB. As Gonzáles et. al. virtual-physical registers [3, 16], the translation registers use an indirection on registers. Virtual-physical registers delay the allocation of physical registers while translation registers are used to manage the side effects induced by the execution of predicated instructions. The use of TRB also permits to design a system that implements a dual register file system (logical and physical). The logical register file is used to store committed instruction results while the physical register file is used to store the in-flight instruction results. At the same time, the use of TRB permits to redirect a particular operand register access, when the register write has been canceled by a predicated instruction. In that respect, the TRB strategy is well adapted to support a fully predicated ISA. To illustrate the use of TRB, the IA64 ISA is taken as our reference ISA, though the TRB concept can be adapted to any other predicated ISA.

Overview

This paper is organized as follow. Since we use the IA64 ISA as our main illustration platform, Section 2 provides an overview of the programming model and presents some specificities of the IA64 ISA. Section 3 discusses register renaming with predicated instruction as it plays an important role within an out-of-order microarchitecture. We also clarify the different meanings of register renaming in the context of the IA64 and an out-of-order microarchitecture. Section 4 introduces the Translation Register Buffer (TRB) concept. Section 5 combines the concept of TRB with a classical out-of-order architecture and illustrates the execution of predicated instructions. Section 6 describes

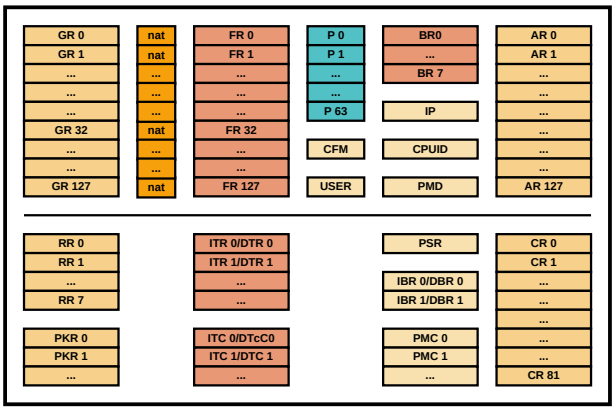
how the TRB is used with predicated instructions. Section 7 presents some experimental results. Section 8 discusses related works while Section 9 summarizes this study and provides direction for future works.

2 Brief Overview of the IA64 ISA

The INTEL/HP EPIC IA64 ISA is the typical example of an ISA defining large ISA register files and predicated instructions. Almost all instructions are predicated and features zero, one or two sources operands and one destination. But some instructions can have four sources operands (`br.call`) or 5 destinations (`br.call` or `br.ctop`).

Figure 1 shows the programming model as defined by the IA64 ISA. The massive amount of ISA registers comprises 128 *general registers* (GR), 128 *floating-point registers* (FR), 64 *predicated registers* (PR), 128 *application registers* (AR) and other book-keeping registers. The IA64 ISA defines two modes of operations for certain classes of registers. Some are said to be *stacked* (i.e. window based) and/or other are said to be *rotating* (within a pipeline loop). The GR are stacked and rotating registers (from 32 to 127). The FR are rotating registers (from 32 to 127) as well as the PR (from 16 to 63). Both stacked and rotating registers are under the control of the *register stack engine* (RSE) whose state is contained in the *current frame marker* (CFM) register. Other registers are mostly control registers. During the execution, stacked and rotating registers are logically renamed by the RSE. This concept of ISA and logical register numbers is discussed in Section 3.

Figure 1 IA64 logical register file.



The IA64 ISA also defines special registers that are built by combining several ISA registers. For instance, the *all predicate register* is a 64 bits register that is built by coalescing all 64 predicate

registers. This kind of register is extremely difficult to handle with an out-of-order architecture built around a single register file, since it requires to read in parallel 64 independent registers.

3 Register renaming with the IA64 and out-of-order architecture

Register renaming has two distinct meanings, depending whether or not it is used in the context of the IA64 ISA or in the context of an out-order architecture. Due to the existence of stacked and rotating registers, a *logical register renaming* (i.e the computation of the effective register number) occurs even with an IA64 in-order architecture. With an out-of-order architecture, *physical register renaming* removes the false dependencies (WAR and WAW) that the use of register based ISA introduces in the program. Therefore, an out-of-order execution with the IA64 ISA will require both logical and physical renaming.

3.1 IA64 logical register renaming

The *logical register renaming* is specific to the IA64 ISA since it defines stacked and rotating registers[1]. Stack registers are used to manage register frames across procedure calls. The size of the frame is controlled through the `alloc` instruction. During the execution, the ISA registers that are within the frame are logically renamed. *Logical renaming* is done by the *Register Stack Engine* (RSE) and operates within the GR, FR and PR spaces. Conceptually, upon entry of a procedure, the `alloc` instruction creates a new register frame by specifying the frame size, the number of inputs, outputs and local registers. This information is stored in the CFM register. During the execution, an ISA register number is transformed into a logical register number.

Although this renaming operation is not that complex, it triggers some side effects with an out-of-order architecture. The most noticeable one is with the `alloc` operation. The `alloc` instruction needs to be executed prior the physical renaming stage in order to become visible for the instruction stream. In fact, all RSE related instructions have a similar behavior and therefore complicate the overall renaming circuitry. However, *logical renaming* in an out-of-order execution core is out of the scope of this paper.

3.2 Physical register renaming and predicated ISA

The *physical register renaming* [13, 7, 15] has been introduced with an out-of-order architecture as a mean to remove *Read after Write* (RAW) and *Write after Write* (WAW) register dependencies between instructions. The complexity of the physical renaming circuitry is directly driven by the issue width as well as the number of source and destination operands. In the case of the IA64 ISA, the physical renaming logic must be prepared to operate with 4 source operands, five destinations and

[1] The SPARC ISA also requires a simple logical register renaming

one predicate per instruction.

Unfortunately, predicated instructions do not coexist very well with renaming techniques. Since an instruction might be canceled by a predicate, the destination register will not be written and further instructions that use such register as a source operand will never see the right instruction result. Example 1 illustrates the disastrous effect of such behavior. If the predicate `p1` is false, the physical register R2 will never be written and the next instruction will not use the right renamed register (or will never execute).

Example 1 Physical renaming problem with predicate.

```
     mov r1 = 1;;     mov R1 = 1
(p1) add r2 = r1,1;;  add R2 = R1,1
     sub r2 = r2,1    sub R3 = R2,1
```

```
     mov r1 = 1;;     mov R1 = 1
(p1) add r2 = r1,1;;  canceled
     sub r2 = r2,1    sub R3 = R2,1
```

The trivial solution to deal with this situation is to consider the predicate as a source operand and have the instruction always execute. In such case, the number of source operands is increased by one (the predicate) plus the number of destination operands. For instance, a two sources and one destination instruction becomes a four sources instruction. In the case of the IA64 ISA, one of the worst instruction is the `br.call` that has five destination operands and three source operands. Using the predicate as a source operand will result in an instruction with five destination operands and nine source operands (3+1+5).

4 Translation register buffer

Our motivation is to design a microarchitecture that supports the execution of predicated instructions with large register file. For this purpose, we introduce a register management policy based on *Translation Register Buffer*. Through the rest of this paper, we denote by *Translation Register Buffer*, the set of *translation registers*. With a *translation register*, the logical and physical registers are decoupled and the mapping (i.e translation) is performed dynamically during the execution.

4.1 Taxonomy of the renaming process

Through the rest of this paper, we denote by *ISA register*, a register that is defined by the ISA and that is part of the opcode. A *logical register* is a logically renamed ISA register. The logical register number is computed during the execution by the RSE. A *physical register* is an in-flight register that is allocated and released by the out-of-order machinery. Furthermore, we denote by *Logical Register*

File (LRF), the register file defined by the ISA[2]. Finally, we denote by *Physical Register File* (PRF), the register file that is used to store the in-flight instruction results prior their commitment in the LRF.

Table 1 maps the various register types with the standard stages and resources found with an out-of-order engine. Given an *ISA register number* (IRN), the *logical renaming* stage computes a *logical register number* (LRN), with the help of the RSE. In our proposal, the *logical register* is renamed into a *translation register* with the help of a translation register allocator, a Register Alias Table (RAT) and a dependency resolver. After the instruction execution, a physical register allocation is performed during the write-back stage. Finally, during the commit stage, the instruction results are copied into the LRF.

Table 1 Register types and names.

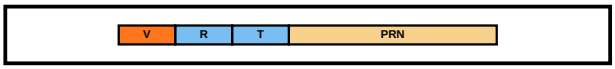
Stage	Register type	name
decode	ISA	IRN
logical renaming	logical	LRN
translation renaming	translation	TRN
write-back	physical	PRN
commit	logical	LRN

The key idea demonstrated in this paper is that a *translation register* introduces an indirection level that permits to map (i.e translate) a *logical register* into a *physical register* or an *ISA register*. The exact translation is determined by the execution flow and the marking in the *translation register*.

4.2 Translation register operation

Figure 2 illustrates the *translation register* format. A *valid bit* (V) indicates that an entry in the TRB is valid. The *ready bit* (R) indicates that the operand is available. The *translation bit* (T) indicates whether the register value must be obtained from a physical register or the logical register. The PRN is the register number used during the translation. It refers to a physical register that is allocated when the translation register is updated.

Figure 2 Translation register format.



The operand read must wait for the translation *ready* bit (R) to be set. Then, the effective source of the operand depends on the *translation* bit (T). If the *translation* bit (T) is set, then the operand is

[2]The term architectural register file, ISA register file or logical register file are all equivalent. However, the index used for the effective access is the LRN.

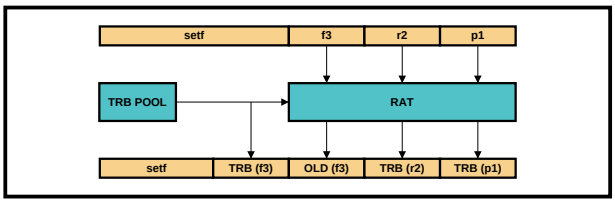
read on the physical register file by using the PRN as an index. If the *translation* bit (T) is unset, the operand is read through in logical register file. This case reflects an access to a previously committed instruction result. Note that in such case, the operand is not available in the physical register file, but must be read in the logical register file (by using the LRN).

The register write policy affects the *ready* and *translation* bits. When a translation register is allocated from the pool, the *ready* bit (R) and the translation bit (T) are unset. During the write-back stage, if the instruction is not canceled, a new physical register is allocated and the translation register is updated as well in order to reflect that a translation can occur. The *translation* bit, the PRN and the ready bit (R) are all set. This mode of operation is the normal mode. The complete discussion about the translation register update is further developed in Section 6.

4.3 Translation renaming with TRB

The translation renaming maps the decoded instruction LRNs into TRNs. The translation stage operates like a physical renaming stage found within classical out-of-order architecture. The instructions in the issue group are renamed in parallel. The predicate and the source registers are renamed by using the corresponding entry in the RAT while the destination registers are renamed by allocating new entries in the TRB. When the instruction commits, the old translation register is released. Such allocation scheme built around a free list is well described in the literature[15]. Figure 3 illustrates the translation renaming circuitry with a *map table* or *register alias table* (RAT).

Figure 3 Translation renaming with a RAT and TRB.



A fully renamed instruction contains (for the predicate, each source and target registers) the L-RN, the new and previous TRNs. The previous TRN is used (under certain conditions discussed in Section 6) in the write-back stage when an instruction is canceled and in the commit stage for TRB resource release (in a similar way found within conventional out-of-order architecture).

4.4 TRB initialization

The initial state of the TRB engine is set to permit the access for all ISA registers. For each ISA register, a translation register is allocated. Each translation register has its *ready* bit set. Since the *translation bit* is unset, the operand read will proceed in the logical register file.

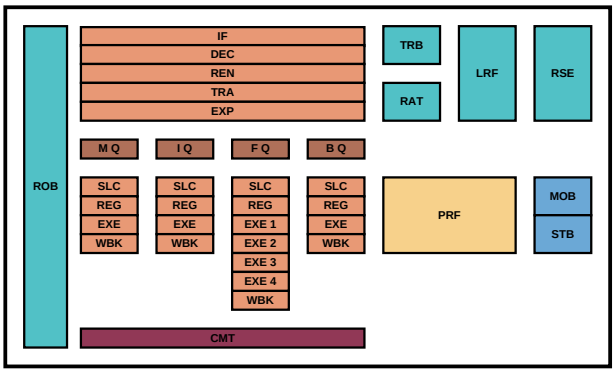
5 Putting it all together

In order to demonstrate the use of TRB, an out-of-order architecture is described hereafter. The main pipeline has 10 functional stages and is designed to operate with the TRB subsystem. After a brief description of the pipeline operation, the TRB engine is described in details.

5.1 Pipeline description

Figure 4 illustrates our base validation model that implements the canonical IA64 ISA. The model mimics a classical out-of-order superscalar processor. Some logic pieces have been adapted to fit the IA64 requirements as well as the TRB microarchitecture.

Figure 4 IA64 out-of-order model.



The microarchitecture components are distributed around a 10-stage pipeline. The simulation model implements four pipeline types, each corresponding to the IA64 instruction type (M, I, F and B). Associated with the pipeline is a set of resources that interacts with the 4 pipelines. Except for the TRB part, most of these resources are found in classical out-of-order microarchitecture. The first noticeable difference is the presence of two renaming stages. The REN stage is the logical renaming stage while the TRA stage is the translation renaming stage. The second difference with usual out-of-order pipeline, is the absence of a specific memory read stage since the IA64 ISA does not combine an address computation and a memory access in a single instruction. Our implementation operates

with a dual register file. The LRF is used to store the committed instruction results and the PRF is used to store the in-flight instructions results.

Table 2 shows the various pipeline stages that are used within the out-of-order engine. The first five stages operate in-order on all issued instructions. With each unit type, an instruction queue separate the in-order part with the out-of-order part. Instructions are fetched by the *Instruction Fetch* (IF) stage and transfered to the *Decode* (DEC) stage. The *Logical Renaming* (REN) stage performs the logical renaming operation with the help of the Register Stack Engine (RSE). The *Translation Renaming* (TRA) stage, physically renames the predicate, source and target registers with the help of the TRB engine (cf. 5.2). Once renamed, the instructions are expanded to their respective unit queue (M/I/F/B units) in the *Expand* (EXP) stage. At this point the instructions are ready to be executed in an out-of-order mode.

Table 2 Pipeline stages summary.

Stage	**Description**
IF	Instruction fetch
DEC	Decode
REN	Logical renaming
TRA	translation renaming
EXP	Instruction expansion
SLC	Instruction selection
REG	Register read
EXE	Instruction execution
WBK	Register write-back
CMT	Instruction commit

Each pipeline unit features a reservation station that stores instructions waiting for their operands to become ready. The out-of-order wakeup/select machinery is implemented in the *select* (SLC) stage (cf. 5.3). An instruction becomes selectable as soon as the availability of its operands is anticipated for the execution stage. We speculatively assume that the predicate will be ready by the end of the execution stage. Once selected, the instruction acquires its operands in the *Register Read* (REG) stage (cf. 5.4). This operation requires the full power of the TRB engine. Once the operands are read, the instruction is ready for execution. The *Execute* (EXE) stage executes the instruction (cf. 5.5). At this point, all instructions have completed their execution, and enter the write-back (WBK) stage (cf. 5.6). In this stage, the results are written into the PRF for temporary storage. The instruction is also marked executed in the ROB. Finally, the instruction is committed in the *Commit* (CMT) stage (cf. 5.7). During this phase, the temporary entries in the PRF are released. The old TRN is also released and the instruction result is written into the logical register file. Table 3 provides a summary of the resources found in the out-of-order engine.

Table 3 IA64 out-of-order components summary.

Component	Description
RSE	Register Stack Engine
ROB	Reorder buffer
RAT	Register alias table
TRB	Translation register buffer
LRF	Logical register file
PRF	Physical register file
MOB	Memory ordering buffer
STB	Store buffer

5.2 Translation renaming stage operation

The *Translation Renaming* (TRA) stage renames an instruction by allocating a translation register for each destination register. The instruction has been previously logically renamed by the REN stage. This translation renaming process operates in parallel on an instruction group and removes RAW and WAW dependencies on registers. When this process is completed, the instructions are placed into their respective queue. The simulation model uses one instruction queue by unit type (i.e M, I, F and B).

5.3 Instruction selection stage

The *Select* (SLC) stage is the instruction selection stage. Instructions are waiting in a reservation station. There is one reservation per execution unit[3]. The selection logic is responsible to find the oldest instruction that is ready for execution. In this stage, the predicate might be unknown (and ignored) but its value will be available in the execution stage (cf. 5.5). Instructions are selected speculatively by assuming that they always execute. However, instructions waiting in the reservation station can be canceled when the predicate is broadcasted by the *Execute* (EXE) stage.

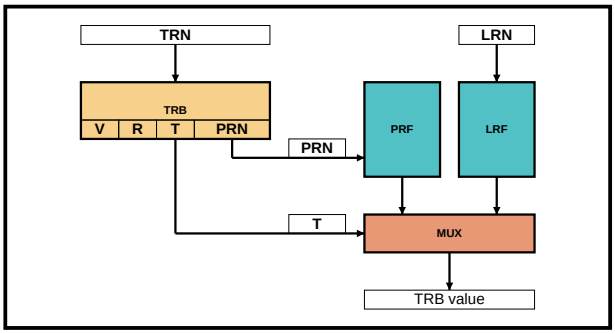
5.4 Register read stage operation

The *Register Read* (REG) stage reads the source operands prior to the instruction execution. The operands can be read from three possible sources: namely the bypass network, the physical register file (PRF) and the logical register file (LRF). If the source operand is not available from the bypass network, the associated translation register is used. As mentioned in Section 4.5, there are several scenarios that can take place. First, if the *ready* bit (R) is unset, the operand is not ready and the instruction must be re-scheduled. With a ready translation register, the *translation* bit (T) indicates whether a translation takes place. When a translation occurs, the operand must be read from the PRF.

[3]It is possible to have multiple units of the same type. For instance, the Itanium II has 4M, 2I, 2F and 3B units

If a translation does not occur, the operand is read from the LRF. Figure 5 illustrates the functional read operation with a translation register.

Figure 5 Register value translation.



As a summary, we list the possible scenarios for reading operands from one source or another.

- **Bypass network**
  The producer instruction has completed its execution, but the write-back did not occur yet.
- **TRB entry not ready**
  The instruction was scheduled speculatively but the source operand is not available. Possible reasons include a cache miss from a load that was supposed to provide the operand or a producer instruction re-scheduling (cf. 5.5).
- **TRB entry with translation**
  The producer instruction has completed its write-back phase but is not yet committed. The source operand is available from the physical register file.
- **TRB entry without translation**
  The producer instruction has been committed. The source operand is available from the logical register file.

5.5 Instruction execution and re-scheduling

The *Execute* (EXE) stage executes the instruction and broadcasts the results in the bypass network. In this stage, all operands, including the predicate, must be available. If one operand is not available, the instruction is re-scheduled for later execution. An operand might not be available in time, because the latency of the instruction producing one operand has not been properly anticipated (e.g. cache miss or instruction cancellation, cf. 6.1). The predicate must be available in order to avoid the

broadcast of incorrect values in the bypass network when an instruction is canceled. If the predicate is not available, the instruction is re-scheduled. Note that scheduling the oldest instruction first guarantees the absence of deadlocks.

5.6 Write-back stage operation

The *write-back* (WBK) stage writes the instruction result into the physical register file (PRF). The physical registers are allocated in that stage, and the associated translation registers are updated to reflect their new state. The complete discussion about the TRB operations with predicate is developed in the next section.

5.7 Commit stage operation

The *Commit* (CMT) stage writes the instruction results into the LRF and frees the instruction resources. The instructions are committed in-order. When the instruction results are written into the LRF, the physical registers that were allocated during the write-back stage are released. The translation registers are updated to reflect that there is no longer a translation, but that the data are ready (i.e *translation* (T) cleared)). The old TRB entries are also released during this stage. If the instruction has been canceled, the TRB entries are marked ready and some cleanup might occur as well.

6 TRB operations with predicate

The TRB mechanism has been primarily designed to operate with predicated instructions. In the presence of canceled instructions, special care must be taken to update the translation register.

6.1 TRB update without cancellation

Without cancellation, the translation register is updated during the write-back (WBK) and commit (CMT) stages. The instruction is scheduled for execution independently of its predicate value. The predicate value is checked in the execute (EXE) stage. This is important since the instruction result must not be propagated in the bypass network if the instruction is canceled. If the operand or the predicate is not ready, the instruction is re-scheduled. Without cancellation, the instructions can operate in back-to-back mode without any penalty.

6.2 TRB update with cancellation

By default, when an instruction is canceled, the destination translation registers are only updated during the commit stage by setting the ready bit. Since the translation register will be marked ready without any translation, the dependent instructions are guaranteed to obtain the correct value from the logical register file. Unfortunately, this default mode of operation can significantly delay the operand read by forcing too many instruction re-scheduling in the execute (EXE) stage. Fortunately,

this problem can be solved in the most frequent case.

In fact, it is possible to update the translation register during the write-back stage by simply forcing the translation (if any) with the previous translation register value. As mentioned in section 4.3, an instruction carries, for each destination register, the old translation register number (old TRN). This index is used by the commit stage to free the old translation register. This old TRN can also be used to access the previous translation in the case of instruction cancellation. Such mechanism is effective at only one condition; **updating a translation register with a old one is a valid operation if and only if the old translation register is ready**. If this rule is not enforced, a deadlock could occur as a result of recursion among the TRB accesses. For this reason, if the old translation register is not ready, the update is not done during the write-back stage, but rather during the commit stage.

6.3 TRB update summary

In summary, we distinguish the following rules that permit to update a translation register:

- **Not canceled instruction**

  Write-back stage: Allocate a physical register for each target registers and update the translation registers accordingly.

  Commit stage: Update the LRF, clean the translation bit, clear the physical register and clear the old translation register.

- **Instruction canceled, old TR not ready**

  Write-back stage: No physical register allocation occurs (i.e translation register not updated). Broadcast false predicate to the reservation station for instruction cancellation.

  Commit stage: Marks the translation register as ready and clear the old translation register.

- **Instruction canceled, old TR ready**

  Write-back stage: Copy-allocate a physical register for each target registers and update the translation registers accordingly. Broadcast false predicate to the reservation station for instruction cancellation.

  Commit stage: Update the LRF, clean the translation bit, clear the physical register and clear the old translation register.

6.4 Example of TRB update with predicate

In order to illustrate the last point, Example 1, from Section 3.2 is analyzed in details. The instructions are numbered from I1 to I3. In this example, we do not consider the logical renaming stage.

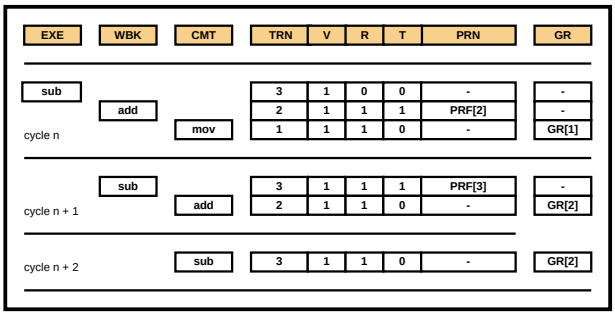
```
      mov r1 = 1;;      (I1)
(p1)  add r2 = r1,1;;   (I2)
      sub r2 = r2,1     (I3)
```

```
      mov TR1 = 1;;     (I1)
(TR0) add TR2 = TR1,1;; (I2)
      sub TR3 = TR2,1   (I3)
```

After the translation renaming stage, we assume that instruction I1 to I3 are renamed with the translation registers TR1 to TR3. we assume also that the predicate is renamed with the translation register TR0.

In the case the predicate p1 (translation register TR0) is true, thus not canceling the instruction I2, the pipeline will process the instructions as shown on Figure 6. The left side of the diagram shows the pipeline state at various cycles while the right side shows the translation registers states.

Figure 6 Example 1 without cancellation.



At cycle n, the instruction I1 commits. The translation register TR1 is updated by cleaning the *translation* bit (T). Since the translation register TR1 is ready, the register translates to the ISA register GR[1]. In the same cycle, the instruction I2 enters the write-back stage. Since the instruction is not canceled, a physical register PRF[2] is allocated and the translation register is updated as well. Note that the instruction I3 enters the execute stage (EXE) and that the translation register TR3 is not ready. In the next cycle, the instruction I2 commits and the instruction I3 enters the write-back stage. The instruction I2 and I3 operate in back-to-back mode and the preceding scenario can be applied.

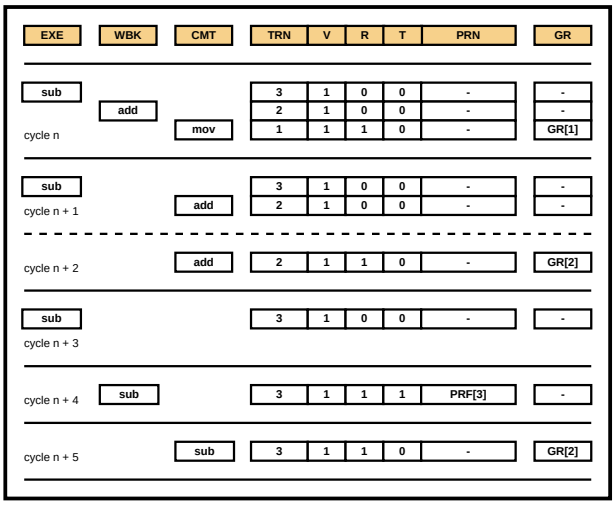
For illustrating the case where the predicate p1 is false, let's assume that the old register GR[2] is the target register of a `getf` instruction as indicated by example 3. With this example, the old GR[2] translation register for instruction I2 corresponds to the one affected to instruction I0.

Example 2 Example 1 with cancellation and latency

```
      getf r2 = f2       (I0)
      mov  r1 = 1;;      (I1)
(p1)  add  r2 = r1,1;;   (I2)
      sub  r2 = r2,1     (I3)
```

Due to the I0 instruction latency and the machine resources constraints, the GR[2] target register of instruction I0 might or might not be ready in time for instruction I2. This is where the rules developed in Section 6.2 are applied. When the instruction I2 enters the write-back stage and the predicate p1 is false, there are two possibilities. If the old GR[2] translation register (the one associated with instruction I0) is ready, the instruction flow continues as shown on Figure 6. In this case, the translation register TR2 is updated with a copy-allocate operation. On the other end, the old translation register is not ready, and therefore the instruction I2 updates the TRB at the commit stage, as indicated on Figure 7. However, in this case, the instruction I3 needs to be re-scheduled, since the GR[2] register is only available one cycle later.

Figure 7 Example 1 with cancellation and latency.



6.5 Managing IA64 specificities

6.5.1 Invariant target register

As a side effect, the TRB mechanism is also capable to manage a special difficulty that we call the *invariant target register* problem. This IA64 specific problem is is illustrated with Example 2 and generally occurs with compare-like instructions. In this example, the instruction semantic specifies that the predicates p2 and p3 are left unchanged if the registers r4 and r5 are equal.

Example 3 Invariant target register problem.

```
cmp.eq.and p2,p3 = r4,r5
```

```
if (r4 == r5) {
  p2,p3 untouched
} else {
  p2 = false
  p3 = false
}
```

In light of Example 2, an *invariant target register* is a register that is conditionally written, depending on the source operands value. The *invariant target register* behavior is a generalization of the instruction execution with predicate. With an out-of-order engine, the invariant target register problem is similar to a canceled instruction.and the trivial solution to deal with this kind of situation would be to add the target registers as source registers and decide whether this value needs to be copied back. However, in the presence of a TRB, an invariant target register is a register that is treated like a predicated one, except that the cancellation condition is computed dynamically.

6.5.2 Special predicate registers

As mentioned in Section 2, the IA64 ISA defines special registers like the (all) predicate register. In our model, this register is accessed from the logical register file by making sure that any former instruction that affects a predicate register has been committed.

7 Experimental evaluation

A complete simulation framework, called **iaoo** (Intel Architecture Out-of-Order), has been designed to emulate or simulate the canonical IA64 ISA. For this purpose, a set of libraries and application programs have been implemented as listed below:

- **iaka**: An IA64 emulator. The **iaka** application program emulates an IA64 binary program and produces an execution trace.

- **iaoo**: An IA64 simulator The **iaoo** application program simulates an IA64 binary program and produces an execution trace.
- **iaos**: An IA64 static analyzer. The **iaos** application program produces static analysis for an IA64 binary program.
- **iata**: An IA64 trace analyzer. The **iata** application program produces reports and statistics from an emulated or simulated IA64 program execution trace.

The **iaoo** simulation environment operates with the full IA64 ELF binary that conforms to the Intel ABI specification [1]. As of today, the Linux environment is one implementation of it. For a practical point of view, the **iaoo** environment operates with statically linked binaries. In fact, there is no real restriction to operate with dynamically linked binary executables, except that it would complicate the overall system execution and add some overhead that is not really needed for the analysis. The execution model is designed to be identical with a real one. For this reason, the process image that is built in memory and the data layout that is emulated are almost identical to the one used during the program execution on a IA64 hardware platform.

The **iaka** emulator and **iaoo** simulator can produce a result "trace". The trace includes the executed instructions, the register read and write operations, the memory access as well as the canceled instructions. With an emulator, the trace is produced by the emulator itself. With a simulator, the trace can be produced by any pipeline stage.

The traces are stored in a binary file that can become huge when executing large simulations. The trace file can be further processed to produce dynamic runtime statistics. Example 4 illustrates the typical structure of a trace produced by an emulator or a simulator.

Example 4 Simulation trace dump.

```
trace 30 {
 CMT:RINSTR @0x40000000000189a0 [M34] [M] [T] alloc r41=ar.pfs,10,0,3,0
 CMT:RINSTR @0x40000000000189a0 [A05] [I] [T] addl r15=1144,r1
 CMT:RINSTR @0x40000000000189a0 [I29] [I] [T] sxt4 r14=r33
 CMT:RINSTR @0x40000000000189b0 [A05] [M] [T] addl r18=88,r1;;
}
```

7.1 Static analysis

In order to assess the TRB's behavior, a suite of SPEC 2000 benchmarks has been used. Table 4 summarizes the static analysis for five integer and three floating-point benchmarks. Each program has been compiled with GCC 3.2 using a medium optimization level (O2). All programs are statically linked and the presented data include the base libraries (i.e libc and lm). All statistics were produced by the **iaos** application program which features a binary code disassembler and analyzer.

Table 4 shows the number on instructions (instr), the percentage of `nop`, the percentage of predicated instruction (pred) and the percentage of predicated instructions that are not branches (nbrp).

Table 4 SPEC 2000 integer, GCC O2 static analysis.

name	instr	nop	pred	nbrp
gzip	149454	30%	13%	6.1%
bzip2	140590	30%	13%	6.3%
mcf	126392	30%	13%	6.4%
parser	158853	30%	13%	6.4%
twolf	217217	29%	11%	5.4%

All percentage are absolute. The program sizes vary between 100K and 300K IA64 instructions. Each application exhibits a 30% average of `nop` instructions and a 10% average of predicated instructions. Among these predicated instruction, a 6% average of instructions corresponds to predicated instruction that are not branches. Finally, the average stop bit distance, (i.e the number of instruction between two stop bits) is 2.8.

Table 5 SPEC 2000 integer, ECC O2 static analysis.

name	instr	nop	pred	nbrp
gzip	193460	30%	11%	5.0%
bzip2	189876	30%	11%	5.0%
mcf	175180	30%	11%	5.1%
parser	209504	30%	11%	5.2%
twolf	330543	30%	11%	5.0%

A similar analysis was performed with another compiler and exhibits similar trends. Table 5 shows the results that were obtained with the Intel compiler ECC. The average of *no operation* instructions remains around 30% with a slight decrease with the predication rate.

Finally, it is worth to note that the same analysis with floating-points benchmarks have exhibited the same results. Table 6 shows the results obtained with a SPEC 2000 suite compiled with GCC version 3, O2 optimization level.

7.2 Dynamic analysis

The five SPEC 2000 integer and three floating-point programs mentioned previously have been used to perform a dynamic analysis. All programs were emulated with the **iaka** application program.

Similar with the static analysis, Table 7 reports the dynamic results for both integer and floating-point benchmarks. The percentage of nop instructions (nop), the predication rate (pred) and the non-branch predication rate (nbrp) are in-line with the static ones. We also reports the percentage

Table 6 SPEC 2000 floating, GCC O2 static analysis.

name	instr	nop	pred	nbrp
swim	187933	30%	12%	5.9%
ammp	244564	32%	8%	4.2%
mgrid	196461	29%	12%	5.8%

Table 7 Baseline GCC dynamic analysis.

name	nop	pred	nbrp	cancel	nbrc
gzip	28%	10%	3.4%	5.1%	0.9%
bzip2	25%	9.2%	3.1%	3.8%	1.3%
mcf	31%	20%	13%	8.9%	6.1%
parser	29%	16%	6.4%	10%	3.1%
twolf	31%	10%	7.4%	4.4%	2.7%
swim	36%	10%	5.8%	4.5%	2.3%
ammp	30%	17%	9.5%	9.0%	5.0%
mgrid	29%	4.4%	2.2%	1.8%	0.7%

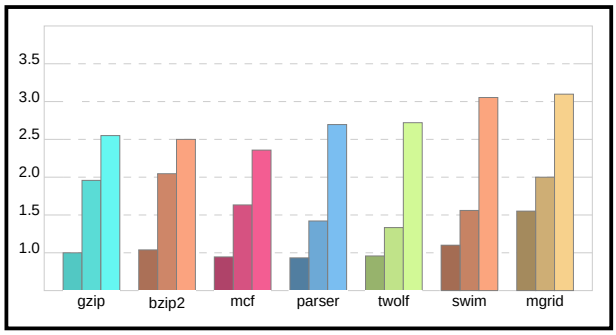
of canceled instructions (cancel) as well as the non-branch canceled instruction rate (nbrc). They illustrates the program behavior for a given compiler. Except for mcf and ammp,the rate of non-branch predicated instructions is low, suggesting that the GCC compiler does not fully exploit the potential of predication. However, similar distributions of non-branch predicated instructions were encountered using the ECC compiler.

7.3 Baseline in-order results

In order to be able to compare the TRB potential, we emulated a very optimistic in-order IA64 architecture using the **iaka** application program. The emulation is performed by using a simple scheduler that uses the base IA64 ISA dispersal rules and the stop bits. A two bundles window (i.e. 6 instructions) was used to perform the emulation. We did not take into account the limited number of resources, nor the floating-point and memory latency, and therefore give best-case results for in-order implementations.

Figure 8 shows the worst (no branch prediction), medium (10 bits gshare branch predictor) and best (perfect branch predictor) IPC that were obtained the five integer and three floating-point SPEC 2000 benchmarks. The emulator was configured with a 10-cycles pipeline refill penalty. This penalty is ignored in best-case mode.

Figure 8 IPC in-order with no/gshare/perfect predictor.



7.4 Out-of-order results

The SPEC 2000 benchmarks analyzed previously have been simulated with the **iaoo** application program. The simulator implements the out-of-order core as described in Section 5. The 10-stages pipeline operates with a dual register file and a complete TRB engine. The simulator was configured with the Itanium II parameters as reported by the Table 9.

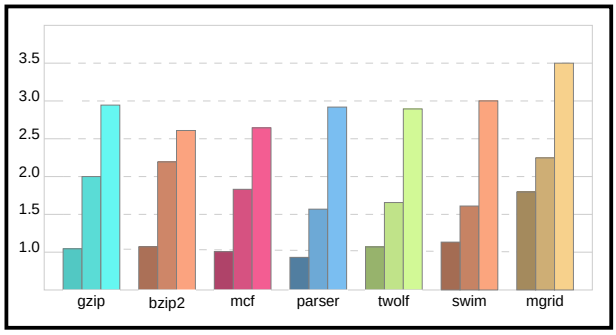
Table 8 Simulator model parameters.

Component	Type
Issue width	6 instructions
Unit queue depth	16
Reservation station	16 entries
M/I/F/B units	4/2/2/3
ROB size	256
Memory	perfect
Branch predictor	none/1K gshare/perfect

The simulator is much precise than the emulator. It takes into account the limited number of resources, as well as the floating-point operation latency. Branch misprediction are resolved at the commit stage and incur a 10-cycle mispenalty. A perfect memory model has been used (i.e. always hit in L1 cache) and the misprediction penalty in the emulator was identical with the simulator. This model has been selected as a mean to characterize the impact of the out-of-order execution with register only effects. If the memory hierarchy had been simulated, the results would have been affected by the memory latency. Compared with the in-order emulation of the previous section, the simulator is by far, more conservative but still demonstrate some IPC improvements as shown with

Figure 9. On average a 10% improvement is obtained with an out-order architecture that implements the TRB mechanism.

Figure 9 IPC out-of-order with no/gshare/perfect predictor.



The simulation results indicates that the 10% gain is independent of the branch predictor nature and therefore reflects solely the gain with register access. Note that this result is limited by the perfect memory model that was used. In the presence of a hierarchical memory, the benefits of an out-of-order architecture will be more noticeable. Other benefits for out-of-order execution might also come from selective predicate prediction and aggressive branch misprediction resolution immediately after the execution stage.

8 Related work

Although, predication can be dated back to at least 1948 with the IBM 604, the association of a fully predicated ISA with and out-of-order microarchitecture was studied by Chang et. al. [12]. They recognized that the execution of predicated instructions requires special register renaming techniques. Recently, with the arrival of the IA64 architecture, the out-of-order execution of fully predicated ISA is again gaining attention by the research community. Work by Wang et. al. [11] recognized that the problem of executing predicated instructions is a by-product of the register renaming. In their paper, they propose to solve the multiple reaching path problem by using a *select* μop instruction. Such instruction is used to select a particular register in the presence of multiple writers. To do so, their mechanism is built by extending the Register Alias Table (RAT) with several (four in the presented example) pairs of physical registers and dependent predicate. Another approach was taken by Chuang et. al [5] with the use of systematic predicate prediction. Conceptually, predicate prediction reuses all existing techniques found within a classical out-of-order machinery. However, it heavily relies on selective replay, that can incur substantial performance loss if not carefully designed. It is interesting to note that the TRB engine could be associated with a selective predicate prediction.

As already mentioned, the TRB architecture is related to González's pioneering work with virtual registers [3] and later extended by Monreal et. al [16]. The use of virtual registers as a mean to delay

the allocation of physical register proves to be effective in reducing the register pressure on the pipeline. Furthermore, Postiff et. al. [10] recognize the benefits of using virtual registers in the presence of large register file by combining them with two different register files. Our strategy is the logical next step that adds the support for predicated instructions. Conceptually, the *select* μop and the TRB architectures are both built around a selection mechanism. With the *select* μop, the selection is performed by a microinstruction and with the TRB, the selection is done through an indirect access. When the predicate is false, the *select* μop approach incurs a one cycle latency (due to the select function) after the instruction evaluation. On the other hand, a TRB based system incurs a one cycle latency only if the physical register cannot be copy-allocated in the write-back stage. Note also that the *select* μop will require extremely large RAT and that the select function is a 8-source operands instruction. By comparison, the TRB engine relies on regular RAT but do need additional hardware for the buffer itself. Finally, both *select* μop and predicate prediction do not work with invariant target registers.

9 Conclusion

In the past ten years, out-of-order execution superscalar processors have shown their ability to efficiently execute a wide spectrum of general purpose applications. In parallel, predicated instruction sets have been popularized by the introduction of a few architectures for embedded applications such as the Philips Trimedia, the HP-ST Lx architecture as well as the EPIC IA64 ISA for general-purpose applications. Even if implementing an efficient out-of-order execution processor for such predicated ISAs still appears to be very difficult, convergence of these two trends is very likely to occur. As a contribution towards this convergence, we have presented a novel register mechanism for implementing an out-of-order execution processor that operates with predicated instructions.

Although, the paper has focused on the IA64 ISA, the concept of *Translation Register Buffer* is sufficiently open to be adapted for other predicated architecture. In future works, we plan to combine TRBs with predicate prediction and thus, reducing the critical dependency path. The TRB will also allow us to support *selective predicate prediction* (i.e high confidence predicate prediction).

References

[1] *Intel Itanium Architecture. Application Binary Interface*, 2000.

[2] *Intel Itanium Architecture. Software Developer's Manual*, 2000.

[3] Antonio González, Mateo Valero, José González, Teresa Monreal. Virtual Registers. In *International Conference on High Performance Computing*, 1997.

[4] B. Rau and J. Fisher. Instruction Level Parallel Processing. *Journal of Supercomputing*, pages 9–50, 1993.

[5] Weihaw Chuang and Brad Calder. Predicate prediction for efficient out-of-order execution. In *International conference on Supercomputing*, 2003.

[6] Harsh Sharangpani, Ken Arora. Itanium Processor Microarchitecture. *IEEE Micro*, pages 24–43, September 2000.

[7] G. Sohi J. Smith. The microarchitecture of superscalar processors. *IEEE proceedings*, pages 1609–1624, December 1995.

[8] Richard Johnson and Michael Schlansker. Analysis techniques for predicated code. In *29th international symposium on Microarchitecture*, 1996.

[9] Joseph C. H. Park and Mike Schlansker. On predicated execution. Technical Report HPL-91-58, Hewlett-Packard Laboratories, May 1991.

[10] Matthew Postiff, David Greene, Steven Raasch and Trevor Mudge. Integrating superscalar processor components to implement register caching. In *15th international conference on Supercomputing*, 2001.

[11] Perry H. Wang, Hong Wang, Ralph M. Kling, Kalpana Ramakrishnan, John P. Shen. Register Renaming and Scheduling for Dynamic Execution of Predicated Code. In *International Conference on High Performance Computer Architecture*, 2001.

[12] P.Y. Chang, E. Hao, Y. Pat and PP. Chang. Using predicated execution to improve the performances of a dynamically scheduled machine with speculative execution. In *International Conference on Paralle Architectures and Compilation Techniques*, June 1995.

[13] R.M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, pages 25–33, January 1967.

[14] S.A. Mahlke, R.E. Hank, J.E. McCormick, D.I. August and W.W. Hwu. A comparison of full and partial predicated execution support for ILP processors. In *International Symposium on Computer Architecture*, 1995.

[15] Dezsõ Sima. The Design Space of Register Renaming. *IEEE Micro*, pages 70–83, September 2000.

[16] Teresa Monreal, Victor Vinals, Antonio González, Mateo Valero, José González. Delaying Physical Register Allocation through Virtual-Physical Registers . In *32nd Annual International Symposium on Microarchitecture*, 1999.

Contents

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399