Strategies to Improve Performance in the IA64 Architecture

Guilherme António Teixeira

Departamento de Informática, Universidade do Minho
4710 - 057 Braga, Portugal
guilherme.teixeira@neoplastica.com

Abstract: The Intel Itanium® 64 bit architecture is designed to reach high levels of performance using two main goals. The use of explicit parallelism (EPIC), where the compiler is able to express inherent parallelism in the instruction sequence, and the improvement of cache and register file use, to decrease the data miss penalties. However, to take real advantage of the model, compilers must be radically changed and optimized to use predication, speculative execution, and a set of novel features. This communication explores how these main goals of the IA64 can be feasible, and how they were architectural implemented, turning them a potential solution to improve performance.

1 Introduction

High levels of performance can benefit from improvements in microelectronics or semiconductor technologies, increasing circuit speed and density, with enhancements and good decisions in processor architecture implementation. This communication will focus the second issue, mainly in two main issues: minimizing the overhead of memory latencies, and improving the instruction execution by maximizing instruction level parallelism (ILP) and/or minimizing the branches overhead.

These main questions are also relevant when referring to compilers, especially in the case of Itanium. The Itanium architecture implements the Explicit Parallel Instruction Computing (EPIC) model architecture characterized by three main policies [11]:

the compiler should play the key role in designing the plan of execution, and the architecture should provide the requisite support for it to do so successfully;

the architecture should provide features that assist the compiler in exploiting statistical ILP; and

the architecture should provide mechanisms to communicate the compiler's plan of execution to the hardware.

As pointed out above, the compiler may play a major role in performance optimisation, depending on its implementation. To effectively know what can be done to improve performance, it is essential to understand, from the performance point of view, the main aspects of this novel architecture, since it takes some computing concepts from the past - such as the VLIW approach, which was not so successful - and from RISC based instruction sets, as a superscalar 6 way processor.

To properly address all these subjects, this communications is organized in three parts: the first deals with EPIC architecture and memory utilization, the second presents a brief overview of a Itanium compiler, and concludes by exposing some of the author remarks on the advantages, the challenges and the future of this technology.

2 The EPIC Architecture

The IA-64 instruction set is based on a set of architectural concepts known as EPIC, for Explicit Parallel Instruction Computing. EPIC is based on the premise that the compiler has much better visibility into program execution than does the hardware, and that it must play an important role in optimizing the code execution, such as increasing the Instruction Level Parallelism (ILP). In the next paragraphs, we will analyse the main EPIC features that can improve performance, and some notes on new behaviours in program and compiler programming which may also improve performance.

2.1 The Instruction Format and Operation – a VLIW Model

In Very Long Instruction Word (VLIW) processors, the compiler controls the parallelism in a static way. It identifies which instructions are independent from each other, and communicates it to the hardware in wide words with more than one operation per instruction. In opposition, the superscalar processors dynamically allocate resources and implement parallelism in runtime.

The VLIW architecture had not the expected success in the past, mostly because it wasted memory space, it had big penalties in cache misses, and even by the static decisions defined at compile time. In this environment, the bus bandwidth is a critical issue, and it must be wide enough. Over the last years, new developments occurred, and now the EPIC architecture can be seen as a positive evolution of VLIW with new features to improve performance and to eliminate those disadvantages. The figures below show the instruction format in IA64 architecture.

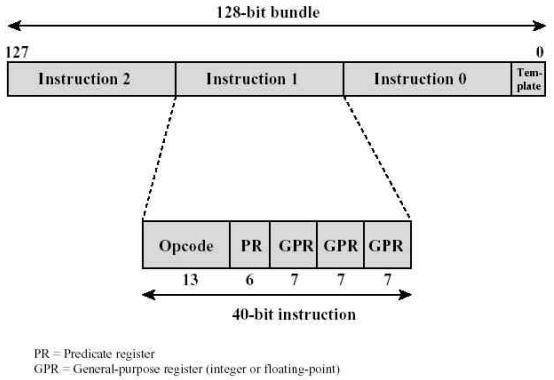

Fig. 1 – The IA64 instruction format. The Bundle [1].

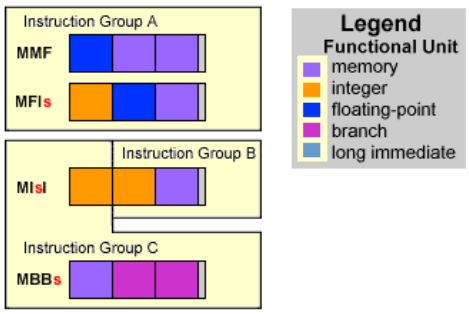

Fig. 2 –Instruction groups A and C end at the end of the bundle. Instruction group B ends in the middle of a bundle (Courtesy of Intel Corp.)

Figure 1 shows a word in the IA64 architecture. Each instruction has a fixed length of 40 bits to accommodate the operation code, to indicate source and destination registers, as predicates for predication execution. This word is called a *bundle* and is a set of 3 "instructions", which do not have read-after-write or write-after-write dependencies among them and may execute in parallel. The bundle also includes an 8-bit template, which includes the grouping information and the type of units will be used for each instruction.

To grant the bundle fulfilling, there are many features that contribute to it. One of them is the instruction groups and the template role. The instruction group can end either at the

end of the bundle, or in the middle of another one. So the bundles can be always fulfilled, and do not need to be filled with null-operations (NOP) instructions. This mechanism also indicates where the parallelism ends. The figure 2 shows an example.

2.2 Predicated Execution

Predication is a technique where the compiler determines which instructions may execute in parallel via a compiler technique known as *if-conversion.*

A typical example in a high level language is a *if-then-else* instruction. By the traditional way, a compiler inserts a conditional branch at *if* point and the program runs by the branch or not, depending of the logical outcome. At the end of this path, it encounters a new unconditional branch for the *else* path, and the two instruction streams join together after the end of the *else* block.

In IA64, at the *if* point, a compare instruction is inserted that creates two predicates, which acquire a logical value, true or false depending on the compare result. After that, the compiler augments both *then* and *else* path instructions with a reference to a register that holds the value for the corresponding predicate of that path. The processor executes both paths of instructions in parallel, because they have no dependencies, and when it knows the compare result, discards one path and commits the correct one.

To optimise and improve performance additionally, the compiler must group instructions not sequentially, but joining the instructions of the both paths in their parallel execution order, in bundles of 3.

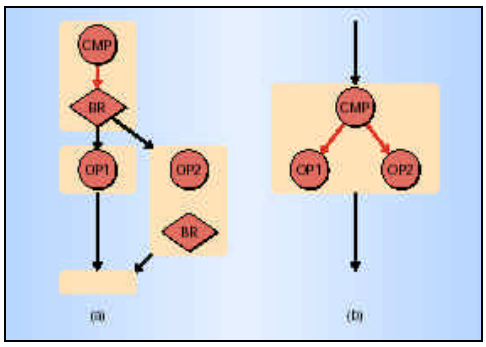

Fig. 3 – Use of predication to eliminate the branch. (a) Basic if-then-else block (b) After the compare (CMP) point, the processor executes both paths of the branch (OP1 and OP2) in parallel, via predicates. [11]

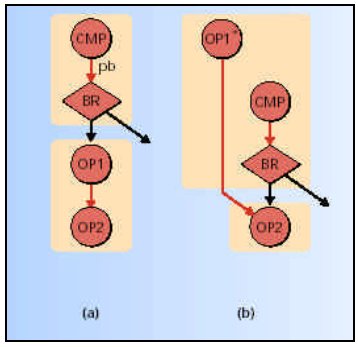

Fig. 4 – The OP1 in (b) was speculative moved into the first block. [11]

This method requires more registers to accommodate the execution of the two instruction paths, and by this reason this technique is especially indicated when branches are not so excessive in directions and when conditional clauses contain only few operations. In the predicated code example below, a data dependency exists between the *cmp* and the two predicated instructions, which execute in parallel.

Unpredicated Code

```
    cmp a,b
    jump EQ
    y=3
    jump END
EQ: y=4
END:
```

Predicated Code

```
    cmp.eq p1,p2=a,b
p1  y=4
p2  y=3
```

Two of the main performance benefits from predication are the reduced number of branches and the effective use of VLIW, avoiding NOP's.

2.3 Speculation – Loads and Control

Another important improvement in IA64 is speculative loading and controlling. This enables the processor to load data from memory before it needs it, to avoid memory latency delays. To hold exceptions, the processor postpones the reporting of them until it becomes necessary to report it. In figure 4 (b) the load instruction for OP1, is replaced by a speculative load in OP1* (ld.s), which executes the memory fetch and performs exception detect.

A checking instruction (check.s) remains in the place of the original OP1 and delivers the exception if it occurs. This check.s instruction can be predicated so that it will only run if the predicate is true. If the ld.s detects an exception, it sets a token bit associated with the target register, called NaT (Not a Thing). Finally if this check.s instruction is executed, branches to an exception-handling routine.

This check feature permits also loads and other instructions that depend on them to be moved over branches. This feature is known as Control Speculation and enables the Out-of-Order operations to be well schedule returning the program result in the correct order at the processing end.

2.4 Memory Hierarchy

To deal with predication and speculative operations, as with all the architecture based in the VLIW, it is critical to hold a very low latency memory, registers, as cache levels and some mechanisms for decrease latencies and improve cache hit ratios. By this reason the IA64 have three cache levels, and a large registry set, as illustrated in figure 5.

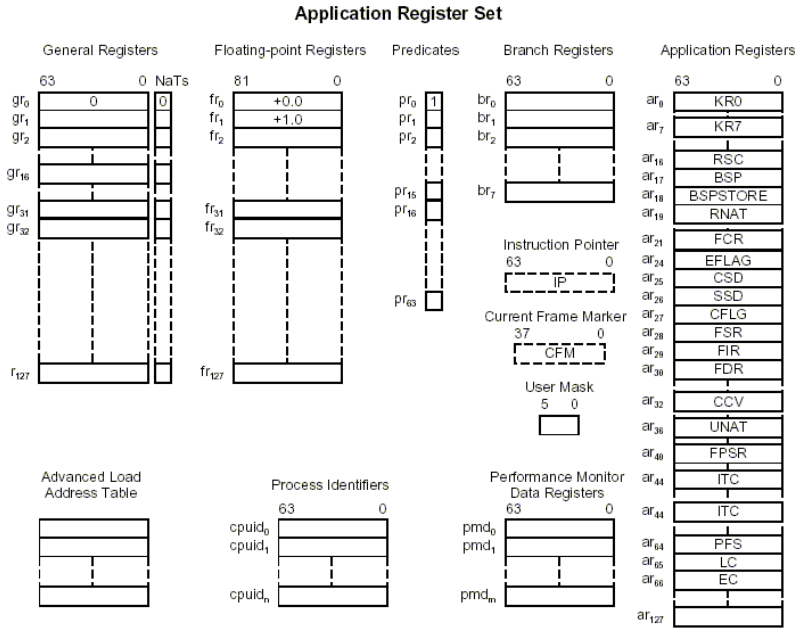

Fig. 5 – Register organization in the IA64 [2].

The IA64 registry file, have a wide range of registers. Among others, 128 for integers; 128 for floating point; 8 branch registers and 64 for predicate execution.

Some of general and floating-point registers are stackable making register stacks implementing register windows, as in the old SUN SPARC® solution. In that case, the procedure switch operations, created a big problem, because all the procedure state needed to be switched too, creating a large overhead in moving the register window.

The IA64 solve this, with an `alloc` operation code, which controls the window size from the compiler, enabling a more efficient resource usage in context switching. The Current Frame Marker (CFM) register assumes this control. When a procedure tries to use more physical registers than remain on the stack, a register stack overflow can occur. To resolve this, the IA64 uses the Register Stack Engine (RSE), which operates in background swapping register state to main memory. Using a register stack reduces the need to perform memory saves.

There are also, some special branch registers within the window to implement a software pipeline, mainly to deal with loops, turning the core of the loop in a set of stages executed as a pipeline. Memory access instructions assume spatial locality, but can be additionally controlled by temporal locality by means of mnemonics used. The cache levels and their characteristics are listed in table 1.

Table 1 – Major attributes of the Itanium processor caches. [7]

Cache	Size (Bytes)	Associativity	Line size (Bytes)	Write Policy
L1I	16K	4 way	32	Read Only
L1D	16K	4 way	32	Write through
L2	96K	6 way	64	Unified: write back
L3	2-4M	4 way	64	Unified: write back

To support the data speculation, there is the ALAT – Advanced Load Address Table containing 32 entries and organized as a 2-way set-associative cache. This table prevents collisions between advanced loads and stores acting as a cache to the register and physical address to be loaded in advance.

2.5 IA64 Pipeline

The pipeline is the glue for the main features we saw over the last pages. In IA64, it has 10 stages as figure 6 shows. Above is a succinct description of each stage.

IPG	FET	ROT	EXP	REN	WLD	REG	EXE	DET	WRB
INST POINTER GENERATION	FETCH	ROTATE	EXPAND	RENAME	WORD-LINE DECODE	REGISTER READ	EXECUTE	EXCEPTION DETECT	WRITE-BACK
FRONT END			INST.DELIVERY		OP. DELIVERY		EXECUTION		

Fig. 6 – Stages of the IA64 Pipeline.

IPG – Instruction Pointer Generation

FET – Load of instruction cache (IL1 cache)

ROT/EXP – The instructions are send to a special 8 bundle buffer. A “Loop Exit Corrector” rotates these instructions in advance, to fulfill all the available ports to the units. In IA64, there are 9 ports (2 – memory, 2 – integer, 2 – float, 3 branch) to address 6 instructions in parallel (6-way superscalar processor). When all the ports are used, it generates a stop bit, to control the allocation of the next instructions. In parallel with the first 3 stages, there are some branch predictor structures that help the IP generator to allocate the reading addresses.

REN – In this stage, the registry renaming features are used.

WLD – Decode instructions.

REG – Operands read from registers. If a stall arises, it is delayed until the execution stage, avoiding some pipeline flushes.

EXE – Instruction execution. If s Stall was computed in the REG stage, now is executed.

DET – Exception detect. The predicates addressed in the EXE stage, are delivered here to retirement, ranch execution or dependency detection.

WRB – Retire Instructions and check of predicates.

The IA64, has we can see now, has a long pipeline. The main problem is that as more stages the pipeline have, the more penalties can be expected from the branch prediction misses. Even in a regular run, without braches, there is a complexity overhead to support the predication, the speculation and other features like the registers renaming. These probably triggered the last generation of IA64 architecture, known as The McKinley, with a shorter 8-stage pipeline.

3 Performance Through the Compiler

3.1 The Compilation Process

On Itanium, as in other architectures, the compiler plays a significant role: it allows applications to gain real advantage by using the available performance tools, here analysed. The profile guided optimisation is an example of improvement in the compilation process. By this technique, a pre-compilation process is done, some running statistics are collected, and after that, the full compilation comes, and is configured with all the information obtained during the pre-compilation process, named the profile. This compilation process is known as the profile guided optimization.

3.2 The Code Tuning

There are also a behaviour we can adopt in order to improve performance. Turning the code “Itanium enabled” can act and improve efficiently our applications. Some examples are the use of `inline` functions for often used small functions in C++, the correct use of static and local variables instead of global ones, and other code improvements which are very important and must be inserted in a well analyzed optimization plan for Itanium application projects.

4 Conclusions

The EPIC model, even just in performance optimisation, is much more then this communication could analyse in few pages. This communication took the key features of EPIC regarding performance, and complemented it with some assumptions and comments about what can be done to improve performance in the Itanium architecture. In the last lines of each of the parts along this communication, some opinions were already discussed. The next paragraphs will pose some additional questions about this technology.

A main challenge of this architecture is the rapid deployment of a new programming philosophy, as well as mature and optimized compilers capable of extracting real advantages of all novel features available in this technology, since the first implementation of IA-64 architecture made a critical broke with the past in this plan. The compilers must be enabled to use the speculative memory accesses, predication, the registers and cache, as well as advances in branch prediction and detailed program analysis. Unfortunately, this is not enough to deplete all possible improvements from the architecture.

To improve the cache hit ratio, for instance, due to the higher VLIW miss penalties, the "best programming techniques" must change to target this architecture, which is not a simple decision, for the reason that it can address performance problems and decreases for other systems.

In spite of this, other superscalar RISC systems - such as the Alpha 21264, a superscalar 6-way with out-of-order execution and register renaming capabilities - are presented as a very aggressive alternative for all the wide range of applications compiled under dynamic mechanisms for high levels of ILP; however, with higher complexity cost!

Let us wait and see if the market deploys solutions for Itanium architecture, and especially if they will extract good performance rates from this architecture and in the following implementations.

Another issue that is not so synergic to the market deployment is the backward compatibility to IA32 systems, which in several opinions and analytical tests did not reach the same performance as native IA32 systems for the same applications. This is critical, since the most possible environments for Itanium, will certainly include old applications that must be supported.

One of the main market slogans of Intel tells us about the simplicity behind some of these key improvements. In fact, the EPIC architecture tells us about the turn over of complexity, since the most complexity goes to the compiler, instead of the microarchitecture, or the processor. In the opinion of some authors [8], the solutions implemented by Intel have a relative extended complexity behind all the registers management and cache levels to turn fully functional the RSE, ALAT, the other available internal tools, like predication and speculation. This problem may create an overhead in processing, as we saw in the pipeline example.

As mentioned before, The McKinley evolution brings a shorter pipeline, as well as some new features, some of them based in microelectronics improvements, and other at the architecture level. Regarding the performance in this chip, Intel announced that McKinley could run the same already compiled programs, from 1,5 to 2x faster, than the current Itanium first generation processors.

Finally, it is well understood that the static compile-time design of the program plan of execution is essential for high levels of ILP even for a superscalar processor, but is equally clear that there are ambiguities at compile time that can only be solved at runtime with dynamic mechanisms.

EPIC subscribes both philosophies and exposes these mechanisms at the architectural level so that the compiler can control these mechanisms, using them selectively when appropriate.

References

[1] Stallings, W., Computer Organization and Architecture, Prentice Hall, 5th Ed., (2000)

[2] Intel IA-64 Architecture – Software Developer's Manual, Vol. I, II, III, IV, Intel, Santa Clara, California, Documents n. º 245317/18/19/20-002, (June-July 2000)

[3] Itanium Processor Microarchitecture Reference for Software Optimization, Intel, Santa Clara, California, Document nº 245473-002, (August 2000).

[4] Greer, B., Harrison, G., Henry G., Li W., Tang P., Scientific Computing on the Itanium Processor, Proc. SC'2001, Denver, (November 2001)

[5] Bharadwaj, J., Chen, W., Chuang, W., Hoflehner, G., Menezes, K., Muthukumar, K., Pierce, P., The Intel IA-64 Compiler Code Generator, IEEE Micro, Volume: 20 Issue: 5, pages 61 – 69, (Set-Oct 2000)

[6] Diefendorff, K., HP, Intel Complete IA-64 rollout – Virtual memory, Interrupts more conventional then ISA, Microdesign Resources, Microprocessor Report, (April 2000)

[7] Quach, N., High Availability and Reliability in the Itanium Processor, IEEE Micro, Volume: 20 Issue: 5, pages 61 – 69, (Set-Oct 2000)

[8] Hopkins, M., IBM Research, A Critical look at the IA64, Microdesign Resources, Microprocessor report, (February 2000)

[9] Dulong, C., Krishneiyer, R., Kulkarni, D., Lavery, D., Li, W., Ng, J., Sehr, D., An Overview of the Intel IA-64 Compiler, Microcomputer Software Laboratory, Intel Corporation

[10] Worley, W., Huck, J., IA-64 Architecture Lab,: Is Out-of-Order Out of Date?, Microdesign Resources, Microprocessor Report, (February 2000)

[11] Schlansker, M., Rau, B. R.; EPIC: Explicitly Parallel Instruction Computing, Computer, Volume: 33 Issue: 2, (February 2000), pages 37 – 45.

[12] Sharangpani, H., Intel Itanium Processor Microarchitecture Overview, Intel Corporation, (October 1999).