The CPU IA-64: Key Features to Improve Performance

Ricardo Freitas

Departamento de Informática, Universidade do Minho
4710 - 057 Braga, Portugal
freitas@fam.ulusiada.pt

Abstract. The IA-64 architecture is significantly different from previous IA-32, and it offers to the operating system and applications a set of features that can improve efficiency in code execution by reducing memory accesses. A quick overview on some IA-64 new features is presented, with a focus on the registers set and on the register stack engine, which are the main features that mostly reduce the memory accesses.

1 Introduction

The IA-64 has a complete different architecture of IA-32 processor, including the instruction set. In this communication we try to make an approach to this new architecture, by talking about some of the new features.

This CPU has great improvements on its performance and cache organization. Here we will concentrate on the CPU performance and on a selected set of features.

A brief overview is given on the features that have a significant impact on the CPU performance: speculation, predication, explicit parallel instruction computing (EPIC) and the large registers set related to the register stack engine (RSE). The latter reduces computation time by decreasing the number of memory accesses.

The main goal of this communication is to analyse the registers set and the RSE. The RSE is not a new concept, since it was introduced by Sun in SPARC. So, we will enter into more detail on this register implementation - the RSE - and how it provides a more efficient implementation of high level language features like functions calls.

At the end some conclusions will certainly be taken and some future perspectives will be made.

2 IA-64 Architecture Overview

In this section we will overview some of the new features of IA-64: speculation, predication and explicit parallel instruction computing. The register set and the RSE will be detailed in the following sections.

2.1 Speculation

Speculation is, in common terms, a conclusion or theory reached by conjecture. A unique feature of IA-64 is data and control speculation. The idea is to offer the compiler with a mechanism by which it can securely move load instructions around to

accommodate for memory access latency without the problems such as faults that may occur, null pointer dereferences or page faults.

In the IA-64 architecture we can have data speculation by introducing *advance load* instructions. An *advance load* is just like a regular load in accessing memory. Still, in addition to that, the object register, load address and number of bytes being loaded are inserted in the Advance Load Address Table (ALAT). The ALAT is checked by every store instruction for entries with overlapping addresses. Such entries are not validated. At the original location of the load in the instruction stream, a speculation check `chk.a` instruction is placed. When executed, it analyses the ALAT for the entry inserted by the matching advance load. If the entry is there, speculation has succeeded; if not, it has failed and a branch is taken to fix-up code.

Control speculation makes possible to the compiler to decrease the stall due to load instructions that suffer huge latencies resulting from cache misses. The compiler can speculatively program such loads far ahead of their regular position in the instruction stream, even in advance of intervening branch instructions. This indicates the load must be executed conditionally depending on the result of the branches. For such speculative loads, exceptions such as page faults are deferred until the result of the speculated branches are identified.

2.2 Predication

The idea is to avoid as much as possible branching on conditional statements by simply prefixing every instruction with a predicate. Predication is implemented using 64 predicate registers of 1 bit each. When the predicate evaluates to `true` the instruction is executed, otherwise the instruction is not executed. The architecture provides powerful ways of writing complex `if-then-else` statements using predicates and parallel comparisons. In general, the compare instruction, `cmp`, sets the first predicate to `true` if the test is positive and the second predicate to `false`. With predication, both compares are run in parallel and they target the same predicates. The two results are stored and then a ***or*** operation is made to see witch one will be in fact executed, performing the complex `if-then-else` statement in 3 cycles (including initialization), without incurring any branches at all.

2.3 Explicit Parallel Instruction Computing

EPIC is a new name for the idea that was first implemented by the VLIW processors (Very Long Instruction Word). The idea is to present instruction level parallelism (ILP) to the compiler and use faster and simpler hardware. The compiler is nearer to the source code, which means that it can get a better understanding of what the program is trying to achieve, it has access to more resources in terms of time and space to help make optimization decisions. As VLIW processors, IA-64 groups instructions into bundles. Each bundle contains three instruction slots of 41 bits each and a template field used to encode which functional units are required (M-unit for memory access, I-unit for integer operations, F-unit for floating point, B-unit for branching). Dissimilar to VLIW, IA-64 allows concurrent execution of numerous bundles. Instructions that can be executed in parallel are grouped by and are terminated by a stop bit which is encoded in the template field. Such a stop is necessary when you have dependencies between successive instructions. This information essential for safe parallel execution is encoded

in the instruction stream. This yields better portability across CPUs of the same kind. It must be noted that stop bits can emerge in the middle of a bundle.

The compiler will be the great responsible for determining and clearing the parallelism present in the instructions to be executed. This is a combination of speculation, predication and explicit parallelism.

3 Registers Set

The IA-64 Register set has 128 general-purpose registers, each 64 bits wide. These registers are conceptually similar to the general-purpose registers such as `eax` on the x86. The IA-64 general-purpose registers are named with an `r`, followed by the register number. Thus, `r0` is the first general-purpose register, and `r127` is the last general-purpose register.
The first 32 general-purpose registers (`r0-r31`) are static. That is, any code that refers to one of these registers will be referring to the exact same register in silicon. All the x86 registers behave this way, and therefore they can be considered static.

Some of the static registers have predefined meanings, and are usually referred to some other way than their `r` name. The global pointer and the stack pointer are two of the most important registers that fit this category. The `r12` register is used as the stack pointer and is thus called the `sp` register. The `r1` register is the global pointer.

In addition to the 32 static general-purpose registers, the IA-64 also has 96 dynamic general-purpose registers, which means that they do not always refer to the exact same register in silicon.

The IA-64 also has 128 floating-point registers. They are named f0 through f127. The floating-point registers are 82 bits in length, allowing them to hold up to a C++ long double. Certain floating-point registers have predefined meanings too. For example, the `f0` register is always set to 0.0, while the `f1` register always holds the value 1.0. The last set of registers contain the branch registers. The IA-64 defines eight branch registers, named `b0` through `b7`. These are 64-bit registers that hold the address of a code location that the CPU can assign control to. On the IA-64, all control transfers take the form of a branch. The `br.call` instruction is correspondent to the x86 `call`; the `br.ret` instruction is like the x86 `ret`; and a simple `br` instruction is similar to an x86 `jmp`.

Dynamic General-Purpose Registers

As it was mentioned before dynamic registers does not always refer to the exact same physical register on the CPU. That is, a register such as `r37` in one function is probable to be assigned to a completely different physical register than `r37` in another function.

Registers will be renamed when control goes from one function to another, keeping values in registers and out of memory as possible; with this technique each function can have its own set of up to 96 registers to work with and so all local variables and parameters can be stored in 96 registers, from `r32` through `r127`. This will reduce the memory access, which is one of the main goals of IA-64; therefore register renaming of the dynamic registers is a vital element to achieve that goal.

As we all know if a parameter needs to be passed to the stack, and the stack is, in case, the main memory and not the memory cache, the CPU might waste some clock cycles just to read the parameter, in opposite the time of access to a register can be single clock cycle.

When all 96 dynamic register are full, how the IA-64 resolves this situation? It is now time for the RSE enters in action

4 The Register Stack Engine

The main goal in a processor is always to speed up his performance so that the programs run faster. The main cause to slow a processor is its access to the memory, has was said before.

Sliding Window in SPARC
The idea of a window of registers that can moves from one function to another is not new. The SPARC architecture from Sun introduced this concept: 32 general purpose integer registers are visible to the program, at any given time. Of these, 8 registers are global registers and 24 registers are in a register window. A window consists of three groups of 8 registers, the output, local, and input registers. The objective is to make possible to some function always have his local variables on a selected window of the registers processor, meaning that his access will be very fast. This works very well when we are dealing with the same process/program; however when we move to a different process all the windows registers will have to be saved into the memory, thus most of the gain will be lost because of the time spending on saving those registers. The IA-64 learned with this experience and their designers created the Register Stack Engine (RSE).

RSE
This hardware implementation inside the processor helps a subset (the last 96) of the general-purpose registers implement the register stack by manipulating register overflows. The idea is to provide the CPU with memory stack implemented on the registers. The software will have unified and flexible register structure acting like the conventional memory stack, but much faster.

It is the RSE and register remapping working together that makes possible for determined function to have all the last 96 general-purpose registers for it self. The remapping makes possible for a function to see always registers from `r32` through `r127`, the RSE will be responsible to make a one-to-one correspondence between a given register in a particular function frame and where it will be saved in the Backing Store Memory, making the 96 dynamic registers always available. In example, we can say that the RSE works like a tank tread going up and down a wall. Each tread plate refers to a dynamic register, and the wall will be the backing store – Figure 1.

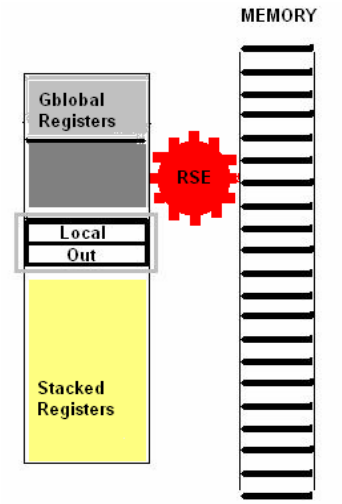

Fig.1 - Register Stack Engine working
Courtesy of Hewlett-Packard

This will make the function call process extremely fast and with almost no overhead. Each time a function/procedure is called, it is allocated a group of registers, called the register stack frame, from these 96 registers that will be local to that function. The function allocate a virtual register stack frame for itself with its size explicitly specified by the `alloc` instruction. Then, when the registers used in a function exceed the number of physical registers, the RSE will automatically save the register stack frame contents on the backing store memory managing the overflows. The work of the compiler in terms explicit spills will be reduced by this virtual register stack frames and RSE.

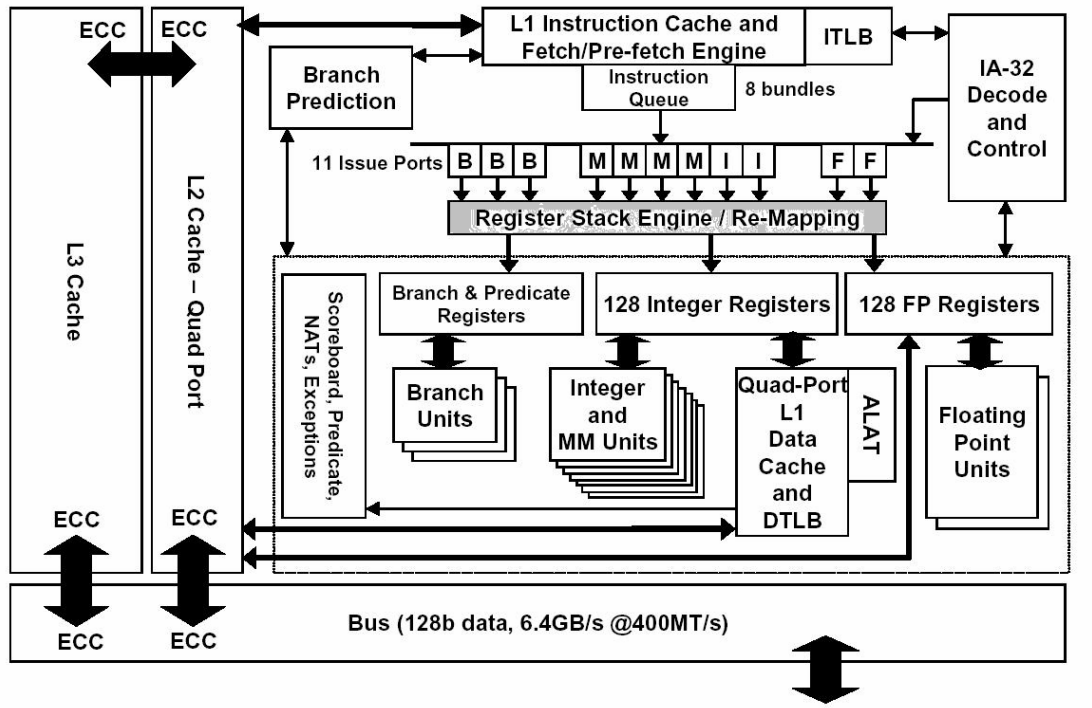

Fig.2 - Block diagram of the IA-64 architecture
Courtesy of Intel

The management of the register stack will be made by the hardware; it also avoids unnecessary register spills and fills on function calls.

We will try to explain with more detail, using the following example, what happens when one function calls another function [1].

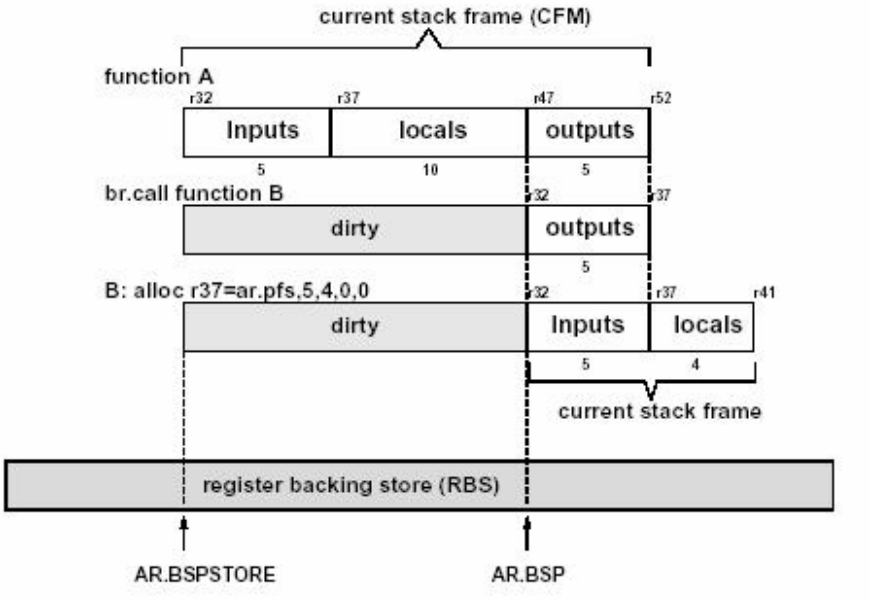

Fig.3 - Register Stack Engine (RSE)
Courtesy of Hewlett-Packard

With the help of figure 3 we will try to explain what happens when function B his called from function A witch has 5 arguments. On top we have the current stack frame of function A. Each frame attributes will be on the current frame marker register (CFM). The size of the frame, `CFM.sof`, in this case the is 20. The `CFM.sol`, is 15 that give us the number of registers that will be 5 input arguments + 10 locals. With this we can infer that the highest number of output registers required by function A to call any other functions can be calculated with `CFM.sof-CFM.sol`=5. With the rXX notation we can observe the logical name of the registers that can be manipulated by the program; the physical registers are represented by the bars. The `br.call` triggers the renaming based on the number of output registers. Now `r32` is the first argument to function B. It is the `alloc` instruction in function B that resizes the frame to the necessitate of the function. Evidently it has 5 arguments more 5 locals. The branch also causes the frame marker of function A to be copied into the previous function state register `ar.pfs`.

The `ar.pfs` register will be conserved and saved (here in r37) and restored by function B in case of modification, such as a successive function call from B.

The function A's locals are now unreachable by the program (automatically preserved) and are on the physical registers which hold old state. The number of physical registers will be full by this renaming mechanism and the register stack engine will spill the old state registers to the register backing store memory.

When execution returns from nested calls, the RSE automatically restores registers from backing store as needed.

Two registers are very important to the process of spills and fills: `ar.bspstore` and `ar.bsp`. The register spills are managed by `ar.bspstore` in a first-in first-out (FIFO) manner; this register points to the memory place where the next register spill will happen.

On the other hand the `ar.bsp` register will point over the location where the last register will be saved, right on the top of the backing store. The `ar.bsp` register will

advance towards the higher addresses every time a new function is called to accommodate the registers that are no more important to this function processing.

The RSE can be configured to run in two different modes. This can be performed by an application register called `ar.rsc`. The two modes are the lazy mode and eager mode. The lazy mode means that the spills will only occur when the file of physical registers is exhausted. The eager mode makes possible to the RSE choose to spill asynchronously from program execution, this means that it can spill even if the file of registers is not exhausted. Currently only the lazy mode is implemented.

We must note that a programmer has the possibility to force a spill and fill operations. This can be performed by explicitly use the `flushrs` and `loadrs` instructions.

In resume the RSE switches the contents of physical registers between general register file and memory, providing a model that looks like unlimited register stack. It operates in the background and the application does not need to know that it exists or how it operates. The RSE is thread implemented on hardware that reads and writes the dynamic registers out to Backing Store Memory.

Register Remapping

In two forms the Instruction Set Architecture requires the remapping of register names; they are the register stacking and the register rotation. Register stacking ensures that the all of the dynamic general-purpose registers can be used by each active frame every time a new function is called; so the hardware control the set of registers by remapping the register number to the correct physical register, and by spilling and restoring the registers between the registers and the backing store memory. On the other hand the register rotation tries to improve instruction level parallelism by allowing overlapped execution of loop iterations. By hardware it will be possible to control counters that will be able to remap the registers numbers to different physical registers on different iterations instructions. Those counters are known as rotating register base.

5 Conclusions

By making the physical register stack size larger than the current 96, almost all the spills and fills can be eliminated, resulting in even greater performance advantage. The IA-64 combines perfectly speculation, predication and explicit parallelism making the EPIC a reality. This Itanium is very different from IA-32 in architecture and instruction set. By learning with previous experiences IA-64 improved the RSE mechanism, which is a great advance for processors performance. We believe that the register stack of the Itanium architecture is a very good feature, which will become more and more important as future workloads become more object oriented.

References

[1] Stephane Eranian and David Mosberger, The Linux/IA-64 Project: Kernel Design. Technical Report HPL-2000-85, Hewlett-Packard Laboratories, June 2000. http://www.hpl.hp.com/techreports/index.html

[2] Matt Pietrek,: Under the Hood, IA-64 Registers. Microsoft, June 2000. http://msdn.microsoft.com/

[3] Hewlett Packard, Inside the Intel Itanium 2 Processor: an Itanium Processor Family member for balanced performance over a wide range of applications, Hewlett Packard Technical White Paper, July 2002.

[4] Scott Townsend, How Itanium's Register Stack Architecture Accelerates Deeply Nested Code, Intel Optimizing Center, August 7, 2002.

[5] Compaq, Dell, Futjitsu, Hewlett-Packard, IBM, Intel, NEC. Developer's Interface Guide for IA-64 Servers (DIG64). http://www.dig64.org

[6] R. David Weldon, Steven S. Chang, Hong Wang, Gerolf Hoflehner, Perry H. Wang, Dan Lavery, John P. Shen , Quantitative Evaluation of the Register Stack Engine and Optimizations for Future Itanium Processors, Intel Labs.