Porting Linux to IA-64

Stéphane Eranian   David Mosberger

Hewlett-Packard Laboratories
1501, Page Mill Road Palo Alto CA 94303 USA

{eranian,davidm}@hpl.hp.com

Abstract

The IA-64 architecture, co-developed by HP and Intel, is going to reach market some time next year with Merced as its first implementation. Major industry players have endorsed this new architecture and technical details are gradually becoming publicly available. However, the complete architecture will not be fully disclosed until machines become available. To provide for early availability of Linux on IA-64, in February 1998 HPLabs began a project to bring Linux to this new architecture with the eventual goal of releasing it to the open source community. This paper gives an overview of the IA-64 architecture and describes our effort so far.

1 Introduction

At HPLabs, we have been working on porting Linux to the IA-64 architecture since February 1998, an activity which is now part of a broader industry effort. The initial goal of our project was to produce a self hosting system that would be available when the first IA-64-based products would appear. Given the progress made so far, we are now looking into producing a fully optimized, complete Linux distribution with most standard packages available. We intend to release all the code to the open source community for eventual integration into the official code base when machines become generally available sometime next year.

Bringing Linux to a new architecture is more than just porting the kernel. To become really usable a system must include a development environment i.e., a complete tool chain, a kernel, the C and math libraries and thousands of tools and commands.

This paper gives an overview of IA-64 architecture with code samples to illustrate some key features. We describe how we brought the various pieces together by first producing a complete tool chain and creating a comfortable simulation environment, then working on the kernel and the C library until we reached the point where building real applications became possible.

2 IA-64 overview

The first implementation of the HP/Intel co-designed IA-64 architecture i.e., the Merced CPU, will reach market sometime next year and will be quickly followed by the faster McKinley[1] in 2001, Madison and Deerfield in 2002. This new architecture builds upon lessons learned from RISC, CISC and VLIW processors. It introduces a new computing paradigm called EPIC (Explicitly Parallel Instruction-set Computing). The basic idea is to expose instruction level parallelism (ILP) to the compiler and use faster and relatively simpler hardware. The compiler operates at a much higher level and has potentially a broader understanding of what the program is trying to accomplish leading to more optimization opportunities.

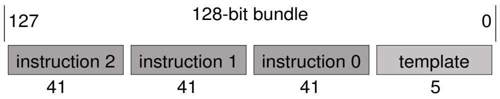

Figure 1: IA-64 Instruction Format

As for VLIW processors, IA-64 instructions are grouped into bundles as shown in Figure 1. Each bundle contains three instruction slots of 41 bits each and a 5-bit template field that encodes which execution unit types are needed by the instructions (M-unit for memory access, I-unit for integer operations, F-unit for floating point or B-unit for branching) and where a stop bit is needed. This synchronization point (barrier) may be necessary between instructions to avoid conflicting operations, like a register read after write dependency, for instance. Parallelism can span more than one bundle and the stop bit can be introduced in the middle of a bundle.

A total of 128 integers registers of 64 bits each and 128 floating point registers of 82 bits each are available. Integer registers between 32 and 127 are called "stacked registers" and are used with the stack engine during function calls. The architecture uses a simple load/store model like RISC and introduces some new features along with the more traditional capabilities expected from a modern CPU, such as multimedia instructions.

The following C code:

```
r2 = r1 == 0 ? r4+r5: r3+r6+1;
```

gets translated into:

```
     cmp.eq p1,p2=0,r1;;
(p1) add r2=r4,r5
(p2) add r2=r3,r6,1
```

Figure 2: example of predication

The concept of predication is implemented using 64 predicate registers of 1 bit each (`true` or `false`). Most instructions can be predicated; if the predicate evaluates to `false`, the instruction is simply not executed. This mechanism can avoid costly branches as is demonstrated in figure 2 with a classic `if-then-else` statement. Predicate `p1` will be set to `true` if `r1` equals zero, p2 will take the opposite value (`p2=!p1`) and only one of the two `add` instructions will be executed, without requiring any branches.

Another feature of IA-64 is control and data speculations, which provide ways to safely move loads off the critical execution path without having to worry about exceptions that could occur, like NULL pointer dereference. A compiler can take advantage of this mechanism to hide memory access latency. Speculation is available for both integer and floating point loads.

Control speculation is the execution of an operation before the branch which guards it. Data speculation is the execution of a load instruction before a potentially conflicting store (aliased address) and is also called advanced load.

The safety of the operation is ensured by the fact that failed speculative loads don't generate faults but instead mark their target register as invalid using a NaT (Not a Thing) bit i.e., the 65th bit of each register. In the case of floating point registers, a special value called NatVal is used instead of an extra bit. Several check instructions can be used to determine whether the load succeeded or not. In case of failure, a normal load can simply be executed or a jump to a recovery code is possible when more operations are required. Advanced loads rely on an internal table called ALAT (Advanced Load Address Table) which is used to check whether or not the target register of the advanced load contains stale information with regards to stores which might have happened after it. Figure 3 shows how a load can, speculatively, be moved before a branch. If the load (`ld8.s`) fails then the NaT bit will be set on `r1` and the check (`chk.s`) will jump to the recovery code.

```
(p1) br.cond label
     ld8 r1=[r5];;
     add r2=r1,r3
```

Can be transformed into:

```
     ld8.s r1=[r5]
     // do something else
(p1) br.cond label
     chk.s r1, recovery_label
     add r2=r1,r3
```

Figure 3: Example of control speculation

To avoid unnecessary registers spills and fills operations on function calls, IA-64 provides a dynamic register renaming scheme which is depicted in figure 4: `rXX` shows the logical numbers whereas the bars represent the physical register file. Registers `r47-r51` are holding parameters to pass to function `B`. The branch instruction (`br.call`) causes the stack frame to "virtually" move forward by renaming `r47` to `r32`. The `alloc` instruction simply resizes the current frame to accommodate local variables. So each time you enter a new function a "fresh" set of registers is available and the compiler does not have to worry about spilling/restoring callee/caller save registers. Registers outside of the current stack frame are considered "dirty" and if no more physical registers are available to satisfy the `alloc` instruction, the register stack engine (RSE) will spill the "dirties" onto a designated backing store location in memory. When returning from a function call, the saved registers are automatically restored from memory.

Finally, IA-64 provides a powerful register rotation mechanism to do software pipelining and unroll loops without incurring code expansion. Integer, floating point and predicate registers can be rotated during a loop cre-

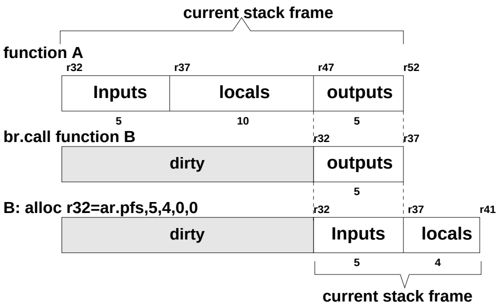

Figure 4: RSE behavior on function call

ating the illusion of a pipeline. We'll give an example of this feature in section 4.

More details about the disclosed capabilities of the architecture can be found on HP's IA-64 web site[1] and in the Application Instruction Set Architecture Guide[2]. Intel's web site also provides online architecture overview tutorials[3].

3 First steps

This is possibly the first time that Linux is being ported to a new platform before any hardware is available. This means not only that we have to rely on simulation for most of the project but also that very few development tools exist.

Linux heavily relies on using a GNU C compiler for both the kernel as well as for user level code, like the C library. Such a tool did not exist for IA-64, therefore we decided to work on it first. The obvious candidate was `egcs`, a very active branch off the gcc development tree. Creating an optimizing compiler for EPIC is not a trivial task and would have required changes to the `egcs` front-end. Such effort was clearly out of the scope of this project. Instead, we focused our attention on producing a functional back-end which generates correct code but doesn't try to use EPIC features like speculation or register rotation, for instance. Cygnus maintains the GNU C compiler and has officially announced last April that they will produce an optimized version of the GNUPro toolkit for IA-64[2]. So as compilers improve, we expect to simply recompile our code in order to get better performance.

A compiler by itself is not enough; the GNU assembler, GNU linker, binary object manipulation library (BFD) and tools like `nm`, `objdump`, `ar` are also required. So we ported the GNU `binutils` package. The tool chain uses `LP64` as the programming model (Longs and Pointers are 64 bits) and the binary format is the official `ELF64` as defined for IA-64. By June 1998, the tool chain was able to pass the `gcc` test suite and the `"Hello World!"` program was generated correctly.

For the execution environment we use a simulator, developed by HP, which emulates the full instruction set of the CPU but not the whole platform i.e., no PCI nor BIOS emulation. It supports two modes of execution: user or system. The former allows us to run user-level applications and execution traps into the simulator for system call emulation. In system mode, kernel bring up is possible as the full VM and interrupts are simulated. Access to I/O devices is achieved by having special device drivers in the kernel which trap into the simulator and get service from the host OS. To simplify user-mode emulation, we have ported this simulator from HPUX to Linux/x86 which allowed a one to one mapping for system calls and parameter marshaling resorted, most of the time, only in 64 to 32 bits conversions.

Once the tool chain and the simulator were in place, the whole development environment was running on Linux/x86 and work on the kernel could really begin.

4 The kernel

We started working on the kernel in late October 1998. Our goals for the kernel were as follows:

- deliver a straight port minimizing the changes to the machine independent part of the code
- follow very closely the development of the official kernel as this would make the final integration phase much smoother.

We have kept our modifications very localized creating new files in two machine dependent directories, namely `arch/ia64` and `include/asm-ia64`, which made it quite easy to follow the latest official kernel developments. In October 1998, we started with `2.1.126` and we are currently running `2.3.X`.

The kernel is running in native 64-bit mode and uses little-endian byte ordering for obvious compatibility reasons with x86. The page size is currently 8KB and the

[1] See `http://www.hp.com/go/ia64`

[2] See `http://www.cygnus.com/news/ia-64.html`

virtual address space of a user process is 43 bits (8TB).

In order to get access to I/O, we developed a series of interrupt driven device drivers which trap into the simulator to get service from host OS. We built a SCSI driver (`simscsi`), a console driver (`simserial`) and later an Ethernet driver (`simeth`). The SCSI driver is very simple and calls the simulator for read/write requests using `[offset,size]` tuples. The disk is emulated using a file on the host as a disk image. Using the `loop` device, we can easily transfer files back and forth between the host and target. This driver also allows us to exercise the complete SCSI code. The serial console driver traps into the simulator for get/put character and an `xterm` is used as the front-end. The Ethernet driver manages raw Ethernet frames which are obtained, via the simulator, from the real interface on the host using raw sockets. The interface is put in promiscuous mode and using a block packet filter, we were able to allocate a specific IP address to the simulated kernel.

We took the incremental approach of bringing up subsystems one by one. We began in late October with an almost completely commented out `start_kernel()` function. At that point we had only the kernel banner working. Since, we have been enabling components like VM and interrupts which allowed us to get through the famous `BogoMips` loop. Shortly after we added context switches and by Christmas we were able to create and run kernel only threads.

Then, we added support for system calls and we landed in user mode in January and were able to execute the `"Hello world!"` program produced a few months ago. At that time we did not have a complete C library, therefore it was impossible to recompile standard applications and run them on the kernel. Instead, we had a μlibc i.e., an extremely small subset of a classic which we used to rewrite simple test programs like a tiny shell (`tsh`), `ls`, `cat`, `mount`, `halt`, etc. Those tools helped us recreate a familiar and comfortable test environment.

In early March, the network stack was up and running. The system had its own IP address and you could ping in and out as well as login from any remote machine. Here again, we rewrote simple versions of `ping`, `rlogin`, `inetd` and `ifconfig`. By Easter, we had signal support and a few weeks later `ptrace` was in place (including system call tracing, single stepping and peek/poke) and it became possible to use the `strace` program for debugging purposes.

As an example of how to combine control speculation with register rotation inside the kernel, we show, in Figure 5, an actual code sequence extracted from `strlen_user.S`[3]. Normally, this function uses an optimistic exception scheme to avoid systematic bounds checking. When a fault is detected, execution goes through an exception table and branches back slightly later in the code with some registers holding special error codes. This example demonstrates how one can use control speculation to achieve exactly the same goal[4]. We use two pipelines of depth 2, `[r32-r33]` and `[r34-r35]`. We load 8 characters at a time (`ld8.s`) speculatively which is handy to safely look way forward in the string. Each iteration loads 16 bytes[5] taking advantage of the memory bandwidth (2 memory operations allowed per bundle) and looks for the zero byte in the previous 16 bytes. Registers `r16` and `r17` are initialized 8 bytes apart and used as base pointers on the string. They are automatically incremented (by `16`) by the load. The `czx` instruction returns the position of the zero byte or `8` if not found.

```
      ...
      init p6 to true
      add r17=8,r16
1:
      ld8.s r32=[r16],16
      ld8.s r34=[r17],16
      czx1.r r14=r33
      czx1.r r15=r35
      ;;
      cmp.eq.and p6,p0=8,r14
      cmp.eq.and p6,p0=8,r15
(p6)  br.wtop.dptk.few 1b
      ...
```

Figure 5: core loop of strlen_user()

Data is inserted in `r32` and `r34` and gets "rotated" each time we go around the loop, it is eventually consumed when it reaches `r33` and `r35` at the next iteration. In reality registers are simply renamed (no data copied) by group of eight (`[r32-r39]`). The illusion of smaller pipelines is created by always entering data at fixed "stages", like we do for `r34`. In case we go too far ahead and hit a page that's not mapped, when the register gets used in stage 1, its NaT bit will be set and the parallel compare instructions will result in `p6` set to 0 (`p0` acts as a sink in this case) forcing the execution out of the loop.

[3]Usually found in `arch/ia64/lib`

[4]The whole function is not shown for space reasons

[5]`r33,r35` are properly initialized before the loop

5 The User land

Once you have a kernel, the work is far from being done as most of the code lives at the user level. First, the C and math libraries need to be ported, then thousands of commands, tools and extra libraries need to be recompiled and sometimes fixed.

While we were doing kernel work at HPLabs, CERN[6] decided to join our effort and started working on those libraries. The first goal was to deliver a generic port and then, look at doing EPIC optimizations for performance critical routines. Linux is using the GNU libc version 2.1 on the major platforms and was, thus, the obvious choice.

After just three weeks of intense work, they were able to run the `"Hello World!"`. With the first code drop from CERN we were able to compile real world applications. We quickly managed to recompile a complete login sequence with `init`, `mingetty`, `login` directly using RPMs[7] from standard distributions. Soon, we had the other basic packages like `util-linux`, `sh-utils`, `fileutils` and even `netkit-base`.

Because we were still missing some libraries, noticeably `curses` some packages were still incomplete but we managed to get shells like `pdksh` and `bash`. We also got our first full screen editor with `vim`, a vi-clone.

Porting existing packages can be very tedious as code quality varies a lot. Most of the problems we've encountered so far revolve around non 64-bit clean code. As an example, the basic `ping` command is still not clean even though one can argue that it runs on Linux/Alpha but it's simply because it relies on the unaligned access trap handler. This is clearly not a long term solution and programs must be gradually fixed.

It is our goal to get a complete distribution, so the basic libraries and utilities are just the first step and work is needed to port other, possibly larger, packages. Clearly a GUI is needed and X11 i.e., XFree86, its associated applications, toolkits and desktop environment like GNOME and KDE will need to be ported. A decent debugger, namely `gdb`, must be also be available for any serious development to become possible. Nowdays a system wouldn't be complete without a web browser thus Mozilla must also be worked on. Languages like Java, Perl, Tcl/Tk, GNU Fortran, Python must also be ported.

[6]Centre Européen de la Recherche Nucléaire – Geneva, Switzerland, see `http://www.cern.ch`

[7]Redhat Package Manager, see `http://www.rpm.org`

6 Next steps

As of today, we have a complete IA-64 development environment hosted on Linux/x86 and based on `egcs-1.1.2`, `gas-990404` and GNU libc v2.1. Our tool chain produces functional code and has proven to be quite robust to get us that far. We have a working kernel with major subsystems enabled. Many real world applications are running on our kernel and also directly on Linux/x86 using the user-mode simulation.

We are actively collaborating with other industry partners like Cygnus, Intel, SGI and VA Linux Systems (Trillian project) and expect to see a major development speed up from this broader effort.

In the near term, we're planning on working on the kernel to fill out the missing pieces like SMP support, the platform specific code, the boot loader and also the IA-32 emulation. Our CERN partners will continue to work on the libraries and noticeably on the dynamic linker and various optimizations. As Cygnus moves forward with their compiler work, we expect to see major improvement to our code. We also intend to tackle the large application space.

While this project still needs to be protected by strict non-disclosure agreements (NDAs), we are keeping Linux developers abreast of our progress and intend to share as much as we can as more information about the architecture is disclosed to the public. Even though it may be hard to join this project, you can still help significantly by making sure than any code you write or read is 64-bit clean. This not only means looking at all abusive casts but also at things like hardcoded data structure sizes and others bad coding habits. We discovered that many of the packages used with Linux don't have good validation tests, so another way of helping would be to develop good test suites. While no tests can be perfect, it would help catch some errors very early on.

7 Conclusion

After giving a rapid overview of the major features of IA-64 we have described what it really takes to port a complete Linux system to this new architecture. At this point in time we have brought forward a complete GNU-based tool chain, a simulator, most of the kernel and the beginning of a real Linux distribution.

While the non-disclosure restrictions make it hard to work completely in the open, we are trying to stay as close as possible to the spirit of open source projects by working with outside partners whenever and wherever appropriate. By doing so, we've made significant progress and think we are on time to deliver a complete distribution to the open source community sometime next year when machines become available. We also hope that our effort will jumpstart a Linux community around this new exciting architecture.

References

[1] Linley Gwennap. Intel outlines high-end roadmap. *Microprocessor Report*, pages 16–19, October 1998.

[2] Hewlett-Packard Company/Intel Corporation. *IA-64 Application Instruction Set Architecture Guide*. http://www.hp.com/go/ia64.

[3] Intel Corporation. *Merced Processor & IA-64 Architecture*. http://developer.intel.com/design/IA64.