The making of Linux/ia64

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 mid-2000 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 HP Labs 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 HP Labs, 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 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 hundreds 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. We finish by describing how we've setup an IA-64 native user environment (NUE) in which it is easy to port and execute real world applications.

2 IA-64 overview

The first implementation of the HP/Intel co-designed IA-64 architecture, 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 simple hardware. Compared to architectures with out-of-order execution capabilities, IA-64 provides a more flexible approach because the compiler has access to the entire program source code and can use more resources (space, time) to make the right decisions in terms of optimization opportunities.

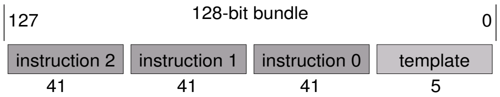

Figure 1: IA-64 Instruction Format

Like VLIW processors, IA-64 groups instructions 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).

In contrast with VLIW, IA-64 allows concurrent execution of multiple bundles. Groups of instructions that can be executed in parallel are terminated by a stop bit. This stop bit is encoded in the template field of every bundle. It also helps portability across CPUs of the same family by not relying on implementation specific information which has always been a barrier to adoption of VLIW processors.

A total of 128 integers registers of 65 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.

The compare instruction will set `p1` to true if `r1` equals zero. Predicate register `p2` will be set to the complement, i.e., `p2=!p1`. We need a stop bit (denoted by `;;`) after the compare instruction to ensure that the processor waits until the predicate registers are set before proceeding onto the additions. If `p1` is `true` then the first addition will be executed and the second, guarded by `p2`, will be ignored without requiring any branches. If `p1` is false then exactly the opposite will happen.

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, due to a `NULL` pointer dereference for example. 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.

```
(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

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 integer 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 recovery can be accomplished either by executing a normal load or by jumping to recovery code. Figure 3 shows how a load can, speculatively, be moved before the branch instruction to avoid the processor stall. 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 (not shown in the figure).

```
// do something else
st8 [r16]=r0
ld8 r17=[r18];;
add r19=r8,r17;;
st8 [r20]=r19
```

Can be transformed into:

```
ld8.a r17=[r18];;
// do something else
st8 [r16]=r0
ld8.c.clr r17=[r18]
add r19=8,r17;;
st8 [r20]=r19
```

Figure 4: Example of data speculation

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. We give an example in Figure 4. At the top, the load in `r17` might stall the processor and delay the addition. Moving the load earlier in the execution stream would help mask the latency, however we don't know whether the store at `r16` might conflict with the load, so the move is risky but if we use an advanced load (`ld8.a`), then we could rewrite the code as shown with no problem. At the original location of the load, we now have a load check instruction (`ld8.c.clr`). This instruction will check in the ALAT if the entry corresponding to `r17` is still marked as valid and, if so, will move on to the addition immediately. Otherwise there was a conflict and the normal load will be re-executed. The `clr` extension simply indicates to remove the entry from ALAT. It should be noted that IA-64 also has the ability to combine data and control speculation with speculative advanced loads.

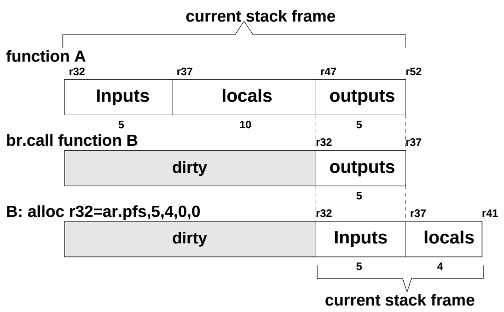

Figure 5: RSE behavior on function call

To avoid unnecessary register spills and fills on function calls, IA-64 provides a dynamic renaming scheme on stacked registers. The idea is to provide a "fresh" set of registers each time a function is entered. This is depicted in Figure 5: `rXX` shows the logical register 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.

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. This means that you can have at least 96 automatically preserved registers on the active execution path without incurring memory spills or fills.

The SPARC [4] architecture also provides registers windows. But there, the windows are fixed in size which, in practice, tends to be either too small or too big. In contrast, IA-64's dynamic approach allows the window to be exactly as big as necessary.

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 creating the illusion of a pipeline. We'll give an example of this feature in Section 5.

More details about the disclosed capabilities of the architecture can be found on HP's IA-64 web site at:

`http://www.hp.com/go/ia64/`

The Application Instruction Set Architecture Guide [2] has been published and is available from HP's and Intel's web sites. Several tutorials on the architecture can also be found on the Internet [3, 5].

3 The tool chain

This is the first time that Linux is being ported to a new mainstream architecture before the underlying hardware is available. This means not only that we have to rely on simulation for most of the project but also that no development tools existed initially.

Linux heavily relies on using the GNU C language for both the kernel as well as for user level code, like the C library. Since a GNU C compiler did not exist for IA-64, we decided to work on it first. The obvious candidate was `egcs`, a very active branch off the gcc development tree[1]. Creating an optimizing compiler for EPIC is not a trivial task and would have required changes to the `egcs` front-end. Such an effort was 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]. As compilers improve, we expect to simply recompile our code 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` and `size` are also required. To this end, we added IA-64 support to the GNU `binutils` package. The tool chain uses the standard LP64 data model (Longs and Pointers are 64 bits) and 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.

4 The simulator

One of the challenges of the project was that no hardware would be available for much of the project and that we would rely on simulation for kernel bring up. While this may sound scary at first, it turned out to be quite a nice experience as we'll describe later on.

Our execution environment uses a simulator, developed by HP, which emulates the full instruction set of the CPU but not all of the platform, e.g., no PCI chips or firmware. It supports two modes of execution: user or system as shown in Figure 6.

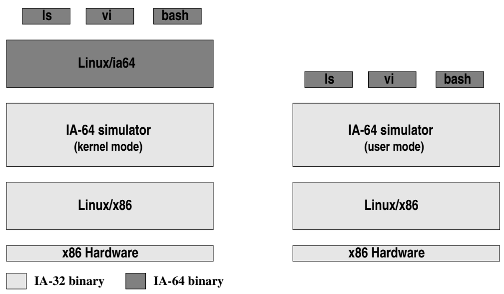

Figure 6: Execution modes

In user mode only the non privileged instructions are available allowing user applications to run directly on top of the simulator. The IA-64 emulation stops at the system call boundary and execution

[1] The egcs team is now reunited with the gcc team, see `http://egcs.cygnus.com`

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

```
struct {
  long long tv_sec;
  long long tv_usec;
} ia64_tv;

struct timeval {/* from <sys/time.h> */
  long tv_sec;
  long tv_usec;
} ia32_tv;

case __NR_gettimeofday:
/* sanity check on args (arg0,arg1) */
...
r = gettimeofday (&ia32_tv, &tzbuf);
if (r != -1) {
 ia64_tv.tv_sec  = ia32_tv.tv_sec;
 ia64_tv.tv_usec = ia32_tv.tv_usec;
 sim_memcpy(arg0,&ia64_tv,sizeof(ia64_tv));
 if (arg1)
   sim_memcpy(arg1,&tzbuf,sizeof(tzbuf));
}
...
```

Figure 7: Example system call emulation

traps into the simulator. At that point, system calls are emulated using the host OS. We have ported this simulator from HP-UX to Linux. Running on top Linux greatly simplifies system call emulation because most calls map more or less directly to their x86 equivalent. Generally, all that is required is 64 to 32 bits parameter translation.

Figure 7 shows the example of how we emulate `gettimeofday(2)`. The sizes of the `timeval` structures differ between IA-32 and IA-64, therefore we first execute the IA-32 system call and, if the operation succeeds, we convert the structure back to 64-bit quantities and copy the result back to the arguments which are both addresses in this case.

In system mode, the full instruction set is available, virtual memory and interrupts are simulated, so OS kernel bring up is possible. Access to I/O devices is achieved by having special device drivers in the kernel which trap into the simulator to get service from the host OS. We give a detailed example in the next section.

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.

5 The kernel

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

- deliver a straight port and minimize 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 by 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 version `2.1.126` and we are currently using the `2.3.X` code base.

5.1 Kernel attributes

The kernel is running in native 64-bit mode and uses little-endian byte ordering for obvious compatibility reasons with IA-32. The current page size is 8KB. Applications see a 64-bit address space, though the current kernel implements only 43-bit (8TB) at this point. Should applications ever grow beyond this limit, the kernel could be changed to support a larger address space. For example, by simply doubling the current page size from 8KB to 16KB, the address space would increase to 47-bit (128TB) with no modifications to existing applications.

5.2 Device drivers

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 the host OS. We built a SCSI driver (simscsi), a serial 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] pairs. 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 file systems. 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.

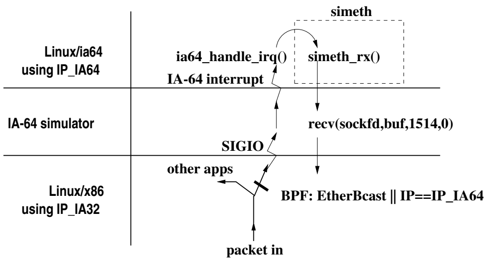

Figure 8: Reception of network packets

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 Berkeley Packet Filter (BPF), we can allocate a specific IP address to the simulated kernel. The standard network commands like, ifconfig or route, are used by the Linux/ia64 kernel to configure its "own" interface.

Figure 8 shows how a packet is received. The simulator opens a raw socket and puts the interface into promiscuous mode, very much like tcpdump, then attaches a packet filter program to it. This filter tells the Linux/x86 kernel to deliver all packets which are either Ethernet broadcasts (like ARP requests) or IP packets with destination address set to the Linux/ia64 kernel. Once such a packet arrives, a SIGIO signal is sent to the simulator which then posts an IA-64 interrupt. This causes the IA-64 kernel to execute the Ethernet receive code (simeth_rx()) which calls back to the simulator to read the Ethernet frame and eventually pushes the packet up the network stack.

5.3 Development time line

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 then, 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 C library which included some string operations, a basic STDIO, a simple malloc and most of the system call stubs. We used it extensively to recreate a comfortable test environment by writing simple test programs like a tiny shell (tsh), ls, cat, mount, halt, etc.

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.

When we look back at the time line, we think that such rapid progress can be attributed to two major factors. The first one comes from the Linux kernel itself and the fact that the same code base has been ported to many other architectures including 64-bit

CPUs like Alpha or Sparc64. The code you have to produce for a new architecture is well identified in the source tree and a clean API exists between machine-independent and machine-dependent parts of the kernel. This is very helpful to get started as you know where to focus your attention and you can also learn from the previous ports. Another reason comes from the fact that we've been using simulation and not real hardware. Even though simulation is slow, it does not really matter when you try to bring up a kernel, accuracy is much more important. The ability to do source level debugging on the kernel, just like you would do with user programs, turned out to be of great help as well.

```
      ...
      init p6 to true
      init r33,r35
      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 // r14==8?
      cmp.eq.and p6,p0=8,r15 // r15==8?
(p6)  br.wtop.dptk.few 1b
      ...
```

Figure 9: core loop of strlen_user()

5.4 Code examples

5.4.1 strlen_user()

As an example of how to combine control speculation with register rotation inside the kernel, we show, in Figure 9, an actual code sequence extracted from `strlen_user.S` usually found in `arch/ia64/lib`. This function computes the length of a string passed from a user program. It differs from the regular `strlen` function by the fact that is has to check for memory access violations. 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 but more efficiently.

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 forward in the string. Each loop iteration loads 16 bytes 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.

Data is inserted in `r32` and `r34` (stage 0) and gets "rotated" each time we go around the loop, it is eventually consumed when it reaches `r34` and `r35` (stage 1) 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` is ignored) forcing the execution out of the loop.

```
      ...
      // 16bytes/iteration
      mov ar.lc=PAGE_SIZE/16-1
      mov ar.ec=2
      mov pr=0xffffffffffd0000
      add src2=8,src1
      add tgt2=8,tgt1;;
1:
(p16) ld8 r32=[src1],16
(p16) ld8 r34=[src2],16
(p17) st8 [tgt1]=r33,16
(p17) st8 [tgt2]=r35,16
      br.ctop.dptk.few 1b
      ...
```

Figure 10: core loop of copy_page()

5.4.2 copy_page()

The next example is the `copy_page()` function usually found in `arch/ia64/lib/copy_page.S`. It is a nice example because the loop is fairly simple and you don't have to worry about alignment problems. Here again we use register rotation to hide memory access latency. This code example shows how a loop can be unrolled without incurring code expansion.

Here again, we copy 16 bytes per iteration to maximize bandwidth usage. A simple way to depict what's going on it to take the analogy of two separate execution streams each one copying 8 bytes and loading the next 8 bytes at each iteration. the `srcX` and `tgtX` symbolic names are used to describe the source and destination base registers for load and copy in each stream. They are initialized 8 bytes apart to avoid collision and get automatically post-incremented by the loads and stores. The loop will be executed `ar.lc`+`ar.ec` times. The `br.ctop` loop is similar to a repeat/until loop, therefore we must set the loop counter `ar.lc` to $n-1$ where n is the number of iterations required.

We also use two pipelines of depth 2 with `[r32-r33]` and `[r34-r35]` just like the previous example. This time however, you can notice that all the instructions in the body of the loop are completely independent of each other, thus no stop bit is required leading to a 1 cycle per iteration loop.

Now the tricky part concerns the initialization and drainage of the pipelines and that's where the predicate registers `p16` and `p17` and the epilogue counter (`ar.ec`) come in handy. The predicate registers `[p16-p62]` rotate similarly to the other registers but predicate `p63` is special. Its value depends on the relative values of both `ar.lc` and `ar.ec`. For this type of loop (`br.ctop`, i.e., a counted loop), as long as `ar.lc` is greater than zero, its values stays at 1 (`true`). The circular rotation causes `p16` to inherit whatever value was in `p63` at the previous iteration. When `ar.lc` reaches 0, `p63` is set to zero and rotation continues. With this mechanism, it becomes easy to embed the initialization (prologue) and drainage (epilogue) of the pipeline inside the loop by simply setting the predicate registers before entering the loop body.

Given the depth of our pipelines, we want the first iteration of the loop to execute only the load part, then enable both loads and stores and terminates by only storing what's left in the two pipelines. The `mov pr=` operation initializes the predicates such that `p16` is true and `[p17-p62]` are false (notice that `p16` and `p17` are one rotation apart). During the first iteration `p16` is `true`, thus only the loads are executed. In the second iteration `p17` receives `true`, i.e., the previous value from `p16`, which itself gets true from `p63` and both loads and stores are executed. When `ar.lc` reaches zero all the loads required have been executed and we simply have two extra stores to do, i.e., one more iteration, that's why `ar.ec` is set to 2 (1 would not cause an extra iteration). At this point `p63` is now zero and the last branch will cause `p16` to get 0 effectively disabling the loads. Finally `ar.ec` is decremented and reaches 1 at which point the loop ends.

Another powerful feature to notice from those examples is that whenever the memory access latency of the machine changes, the structure of the code stays the same and only a few places need to change to account for deeper/shorter pipeline.

6 First steps in 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 hundreds of commands, tools and extra libraries need to be recompiled and sometimes fixed.

While we were doing kernel work at HP Labs, CERN[3] decided to join our effort and started working on those libraries. The first goal was to deliver a generic port and then to look at doing EPIC op-

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

timizations for performance critical routines. The GNU libc version 2.1 is used on the major platforms and was, thus, the obvious choice.

After just three weeks of intense work, they managed to run the `"Hello World!"` program. With the first code drop from CERN we were able to compile real world applications. We quickly recompiled a complete login sequence with `init`, `mingetty`, `login` directly using RPMs[4] from standard distributions. Soon, we had the other basic packages like `util-linux`, `sh-utils`, `fileutils` and even `netkit-base`.

We managed to get shells like `pdksh`, `tcsh` and `bash`. We also got our first full screen editor with `vim`, a vi-clone.

Porting existing packages can be a bit tedious as code quality varies a lot but it turns out to be an excellent validation test for the kernel and libraries. Most of the problems we've encountered so far with applications revolve around non 64-bit clean code. For example, until very recently `ping` was not 64-bit clean because it was relying on `struct timeval` being of size 8 which is not true on IA-64 where it is 16 bytes.

The problem is somewhat alleviated by the fact that most of the key packages have been fixed to run on Linux/Alpha. However many programs still need some cleaning because relying on unaligned access trap handler is clearly not a long term solution.

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. These days a system wouldn't be complete without a web browser and thus Mozilla must also be worked on just like all the languages like Java, Perl, Tcl/Tk, GNU Fortran, Python, etc.

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

7 The NUE environment

As we were porting applications, we realized that our development methodology was not very practical. Running big applications on top of the kernel is inherently slow as our simulator had never been designed for speed. Recompiling existing packages directly from RPMs is quite a challenge as all Makefiles need to be tweaked to change the compiler to use the egcs for IA-64. Sometimes it's even worse, as some packages use helper programs during the build process and most Makefiles assumes host and target environment are identical.

To circumvent those problems we had to come closer to what you would get on the target system in terms of development tools, locations of files, name of commands and ability to execute binaries. To achieve this, we developed what we call the Native User Environment (NUE). Specifically, NUE has to:

- provide an easy to use porting environment hosted on Linux/x86
- minimize modifications to packages (especially Makefiles)
- allow execution of IA-64 binaries without kernel and directly from the shell prompt in a transparent fashion.

The key point is that most applications don't actually require to run on top of the Linux/ia64 kernel to execute successfully, oftentimes user-level simulation is good enough.

Our simulator already offered user-mode simulation and could run in batch mode, i.e., command line invocation with no output. The next piece of the puzzle was to get transparent execution at the shell prompt. We used a mechanism similar to what you get for a shell script. Linux has a module called binfmt_misc which is used to dynamically bind binary types to specific interpreters which is generally used with Java programs. So we simply had to tell the kernel that whenever it is trying to execute an ELF64 binary it should launch the simulator. This is achieved very simply as shown in Figure 11 where ia64sim is the name of the simulator.

```
# echo ":ia64:M::\x7fELF...:\
:/bin/ia64sim:" \
  >/proc/sys/fs/binfmt_misc/register
```

Figure 11: binfmt_misc with IA-64 binaries

The next step was to build an environment in which all cross compilation tools would have native names like cc, ld, as. To do this safely, we decided to build a self-contained environment which you would enter via chroot. So we recreated a standard Linux file system tree keeping only the non binary commands. The next step was to put the tool chain in the right place and install all required headers files and libraries in /usr/include and /usr/lib respectively. We also needed to import quite a few IA-32 binaries like the dynamic loader, IA-32 shared libraries such that host binaries would still run. We also copied IA-32 tools like editors, make, cp actually creating an hybrid system where you could mix and match binary types. For the illusion to survive the build process (especially the configure phase) we have had to fake a few commands like arch and uname and force them to return what they would normally print on a real IA-64 system. Figure 12 shows what the output of a few commands looks like.

With those pieces in place, it became very easy to take an existing source RPM package and recompile it directly using a simple command as is shown in Figure 13.

With this comfortable environment we started tackling more seriously the hundreds of packages that you typically find on Linux distributions. So far we have successfully recompiled and executed editors like vim and emacs, shells like bash, news readers like tin, web browsers like lynx, network

```
# cd /nue
# chroot .   <- in NUE now
# /bin/arch
ia64
# uname -m
ia64
# ld -v
GNU ld version 2.9.4 (with BFD 990404)
Supported emulations:
   elf64_ia64
# file /usr/bin/ld
ELF 32-bit LSB executable, Intel 80386,
version 1, dynamically linked, stripped
# file /usr/bin/rpcgen
/usr/bin/rpcgen: ELF 64-bit LSB exe-
cutable, IA-64 version 1, stripped
```

Figure 12: output of commands in /nue

```
# rpm --rebuild --target ia64 \
      mingetty-0.9.4-10.src.rpm
Installing mingetty-0.9.4-10.src.rpm
Building target platforms: ia64
Building for target ia64
Executing: %prep
...
```

Figure 13: Rebuilding RPMs

commands like ftp and telnet and many others. This environment is fairly easy to replicate onto another machine and can be of great help when porting applications to IA-64.

8 Next steps

As of today, we have a complete IA-64 tool chain hosted on Linux/x86 and based on egcs-1.1.2, gas-990404 and GNU libc v2.1. It 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 inside our Native User Environment.

We are actively collaborating with other industry partners like Cygnus, IBM, Intel, SGI and VA Linux Systems as part of the Trillian project. In the near term and in the framework of this project, we're planning on working on the kernel to fill in the missing pieces like SMP support, the platform specific code, the boot loader and also the IA-32 emulation. Our partners at CERN 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 the information about IA-64 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 hard coded data structure sizes and other 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.

9 Conclusion

After giving a brief 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, an easy-to-use porting environment, 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 other partners. By doing so, we've made significant progress and think we are on time to deliver a complete and optimized distribution to the open source community sometime next year when machines become available. We also hope that our effort will help jump start 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.

[4] SPARC International. *The SPARC Architecture Manual, Version 9*. Prentice-Hall, 1993.

[5] Sverre Jarp. *IA-64 Architecture: A detailed tutorial*. CERN. http://nicewww.cern.ch/˜sverre/.