The Linux/ia64 Project: Kernel Design and Status Update

Stephane Eranian, David Mosberger
Internet and Mobile Systems Laboratory
HP Laboratories Palo Alto
HPL-2000-85
June 30th, 2000*

E-mail: {eranian,davidm}@hpl.hp.com

Linux, kernel, IA64, Itanium

The IA-64 architecture co-developed by HP and Intel is going to reach market in the second half of 2000 with Itanium as its first implementation. All the major industry players have endorsed this architecture and nowadays most of the specifications are public. To provide for early availability of Linux on this platform, the port started over two years ago at HP Labs and grew to become an industry wide effort. A major milestone was reached earlier this year, when the entire source code produced to support this new platform was released to the Open Source community. In this paper, we describe some of the key system architecture features of IA-64 and the major machine dependent kernel subsystems. We also give a brief update on the application level developments including the software development kit for Linux/ia32 recently released.

* Internal Accession Date Only
Approved for External Publication

The Linux/ia64 project: kernel design and status update

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 in the second half of 2000 with Itanium as its first implementation. All the major industry players have endorsed this architecture and nowadays most of the specifications are public. To provide for early availability of Linux on this platform, the port started over two years ago at HP Labs and grew to become an industry wide effort. A major milestone was reached earlier this year, when the entire source code produced to support this new platform was released to the Open Source comunity. In this paper, we describe some of the key system architecture features of IA-64 and the major machine dependent kernel subsystems. We also give a brief update on the application level developments including the software development kit for Linux/ia32 recently released.

1 Introduction

The Linux/ia64 project started over two years ago at HP Labs with the initial toolchain and kernel work. This activity later became part of a broader industry effort known as the IA-64 Linux Project[1] (formerly Trillian project).

The goal of this project is to produce a single, fully functional and optimized port of the entire Linux operating system to the IA-64 architecture by the time the first systems reach market. Itanium-based products are scheduled to appear in the second half of 2000. Since last February, we have released to the Open Source community all source code, making Linux/ia64 the first publicly available operating system for this platform. As of today, prototype hardware is available and most of the specifications of the architecture have been made public. This allows us to give more technical details about our work.

In previous publications [3, 2], we gave a general overview of how the project got started and how the first developments were made using the HP IA-64 instruction set simulator. In this paper, we quickly review some of the key system-level features of IA-64. Then, we describe in detail most of the major machine dependent subsystems of the kernel like the virtual memory, interruptions and signal handling. We also explain how the IA-32 emulation is implemented. In the second part, we give a status update on the developments at the application level. We cover the libraries, development tools, graphical environments and also Linux distributions. In the last part, we describe the recently released IA-64 software development kit which allows people with no IA-64 hardware to develop applications and do kernel hacking on any Linux/ia32 systems.

2 IA-64 architecture overview

The first implementation of the HP and Intel co-designed IA-64 architecture, the Itanium processor, has now been produced and will reach market later this year. It will be quickly followed by the faster McKinley in 2001. This new architecture builds upon lessons learned from VLIW and RISC. It introduces a new paradigm called EPIC (Explicitely Parallel Instruction-set Computing). The idea is to expose instruction level parallelism (ILP) to the compiler and use faster simpler hardware. The compiler is closer to the source code, i.e., 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.

Like VLIW processors, IA-64 groups instructions into

[1] Check out http://www.linuxia64.org

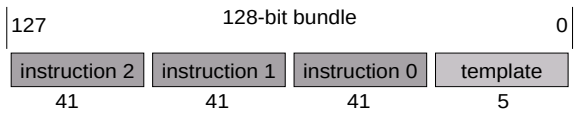

Figure 1: IA-64 Instruction Format

bundles as shown in Figure 1. 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).

Unlike VLIW, IA-64 allows concurrent execution of multiple bundles. Groups of instructions that can be executed in parallel are terminated by a stop bit which is encoded in the template field. Such a stop is required when you have dependencies between consecutive instructions, like in a producer-consumer relationship. The information required for safe parallel execution is encoded in the instruction stream. This yields better portability across CPUs of the same family. It must be noted that stop bits can appear in the middle of a bundle.

It has been over a year since the first public specifications of the architecture [5] have been made public. Earlier this year, the system architecture [7] has been released and just a couple of weeks ago the microarchitecture specifications [8] for the Itanium processor have been available. Today, you can get a lot of information on the architecture from Intel[2] and HP[3] web sites.

In past publications [3, 2], we have already described in detail some of the unique features of the architecture. So here we will only focus on a small subset to help the reader understand some of the kernel design choices.

2.1 Register sets

This architecture provides a large set of hardware resources to the programmer. A total of 128 integer registers, also called general registers, of 65 bits each are available. The 65th bit is used during speculative operations. Integers registers `[r32-r127]` are called "stacked registers" and are used with the stack engine during function calls. You also have 128 floating point registers of 82 bits each. Because of the large number of floating point registers, IA-64 splits them in two partitions of 32 and 64 registers each called low and high. Using the processor status register (PSR) `dfh` and `dfl` bits it is possible to enable or disable the use of either partition. A fault would be generated if access occurred to a disabled partition. This mechanism can be used to speed up the context switch code as we'll see later on.

IA-64 also defines a large set of application registers (AR) which are used to hold state information for the stack engine, IA-32 emulation, atomic operations and loop operations. An interesting AR is the `ar.itc` which is the cycle counter. When `ar.itc` is equal to `ar.itm`, the interval timer matching register, a "timer" interrupt is generated. This can be used as a very low cost interval timer.

There is also a set of eight ARs, called Kernel Registers (KR) which are writable only at the most privileged level but readable at any level, they can be used to safely hold some non sensitive kernel state. Some of the control registers (CR) are used to change the behavior of the CPU with regards to fault handling. Others point to system wide tables, like `cr.iva` which points to the OS-defined interrupt vector table (IVT). The CRs also hold the state of the machine when an interruption occurs.

IA-64 provides a classical load/store model like RISC along with up-to-date features like multimedia instructions. It also introduces some unique features which we'll review quickly now.

2.2 Predication

The concept of predication is implemented using 64 predicate registers of 1 bit each. The idea is to avoid as much as possible branching on conditional statements by simply prefixing every instruction with a predicate. When the predicate evaluates to `true` the instruction is executed, otherwise it simply behaves like a `nop` (no operation). The architecture provides powerful ways of writing complex `if-then-else` statement using predicates and parallel comparisons like shown in Figure 2. In general the compare instruction, `cmp`, sets the first predicate to `true` if the test is positive and the second predicate to the opposite value, i.e., `false`. In the code shown in Figure 2 both compares are run in parallel and they target the same predicates. At first, this may seem like a conflict but in this case it isn't because of the type of test performed. The `or.andcm` comparison will only write both predicates with `p1=false` and `p2=true` when the result of the test is negative: negates hypothesis of `p1=true`. So if you initialize the predicates correctly, you will get the desired effect of a logical `or` operation. If the test succeeds `p1` will be `true`, `p2` will be `false` and we'll execute the first addition. Oth-

[2] Go to http://developer.intel.com/design/ia-64/
[3] Go to http://www.hp.com/go/ia64

erwise we'll run the second addition, indeed performing the complex `if-then-else` statement in 3 cycles (including initialization), without incurring any branches at all.

The following C code:

```
if (r2 == 0 && r3 == 1)
    r32 = r33+r35;
else
    r32 = r36+r33+1;
```

gets translated into:

```
     // r0=0 constant zero
     // sets p1=true, p2=false
     cmp.eq p1,p2=r0,r0;;
     cmp.eq.or.andcm p1,p2=r2,r0
     cmp.eq.or.andcm p1,p2=r3,1;;
(p1) add r32=r33,r35
(p2) add r32=r36,r33,1
```

Figure 2: Predication and parallel comparisons

2.3 Speculation

Another unique feature of IA-64 is control and data speculation. The idea is to provide the compiler with a mechanism by which it can safely move load instructions around to accommodate for memory access latency without having to worry about faults that may occur, like NULL pointer dereferences or page faults.

Speculative loads are available for both integer and floating point registers. If the load fails, instead of "taking" the fault, the error is recorded in the NaT bit of integer registers, i.e., the famous 65th bit. For floating point registers a special value (`NatVal`) is used instead. When the program actually needs the result of the load, it checks the NaT bit of the register used as the load target. If it is not set then the speculation was successful and execution continues. Otherwise, the load can be retried or some recovery code can be invoked.

Control speculation allows the compiler to move a load before a branch that guards it in a safe manner. Similarly data speculation allows to move a load, then called advanced load, before a store that might conflict because of aliased addresses.

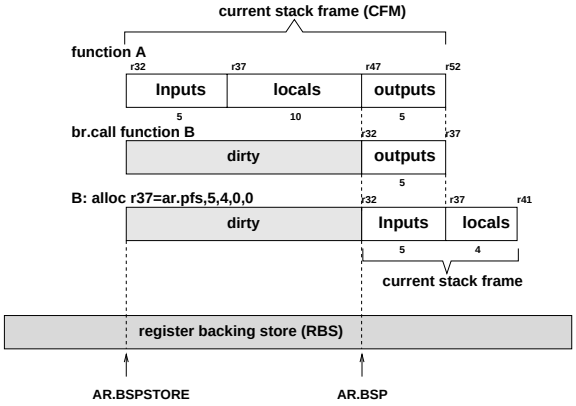

Figure 3: Register Stack Engine (RSE)

2.4 The stack engine

To avoid unnecessary register spills and fills on function calls, IA-64 provides a dynamic renaming scheme for the stacked registers. Each time you enter a new function you get a "new" set of registers for local and output variables.

Figure 3 describes what happens when you call from function A to function B which has 5 arguments. The top bar shows the current stack frame of function A. The attributes of the frame are part of the current frame marker register, `CFM`. The size of the frame, `CFM.sof`, is 20. The number of "local" registers, `CFM.sol`, is 15 = 5 input arguments + 10 locals. From this information, we can deduce that the maximum number of output registers needed by function A to call any other functions it uses is `CFM.sof-CFM.sol=5`. The `rXX` notation shows the logical name of registers that the program manipulates while the bars represent the physical registers. You can see that the `br.call` triggers the renaming based on the number of output registers, in this case 5. At this point `r32` is the first argument to function B. The `alloc` instruction in function B resizes the frame to the needs of the function. Obviously it has 5 arguments and here, we added 5 locals. The branch also causes the frame marker of function A to be copied into the previous function state register `ar.pfs`, This is a preserved register and it must be saved (here in `r37`) and restored by function B in case it gets modified, like during a subsequent function call from B. The registers that were part of function A's locals are now inaccessible by the program (actually been automatically preserved) and are part of the set of physical registers which contain old state, i.e., they are "dirty". When the number of physical registers is exhausted by the renaming mechanism, the register stack engine (RSE) does spill the "dirties" to

a designated memory region, called the register backing store (RBS). When execution returns from nested calls, the RSE automatically restores registers from backing store as needed.

Two registers describe where the stack engine is with regards to spills and fills. Register spills are managed in a first-in first-out (FIFO) manner. The `ar.bspstore` register points to the memory location where the next register spill will occur (for our example, it is where the `r32` of function A will be saved). The `ar.bsp` register points just above the location where the last register will be saved (above where `r36` of function A will be saved), i.e., the top of the backing store. At each function call, `ar.bsp` is advanced towards higher addresses (grows up) to accommodate the new dirty registers. For optimization purposes, the RSE can be configured to run in two major modes via an application register called `ar.rsc`. In lazy mode, spills will only occur when the pool of physical registers is exhausted. In eager mode, the RSE may choose to spill asynchronously from program execution. In the current Itanium chip, only the lazy mode is implemented. It should be noted that the programmer has the possibility to force a spill operation explicitely using the `flushrs` instruction. Similarly, the `loadrs` instruction can be used to force a RSE reload, i.e fill operation.

Although this mechanism is useful for passing parameters, it does not obviate the need for a real memory stack. The software calling convention [4] dictates that at most eight integers registers can be passed by this mechanism, any extra arguments use the memory stack. Up to eight floating points can be passed in registers (`[f8-f15]`). Everything beyond that uses the memory stack. Also any local memory storage (large auto variables, `alloca()`), non stacked registers spills/fills require a real memory stack. Therefore IA-64 applications and OSes have to deal with two stacks: the memory stack and the register backing store (RBS).

2.5 System architecture

Last February, the system architecture [7] of IA-64 was unveiled during the Intel Developers Forum (IDF). This provides a lot of material to talk about. Here we focus mostly on the virtual memory and interruptions mechanisms to help understand how the Linux/ia64 kernel has been designed.

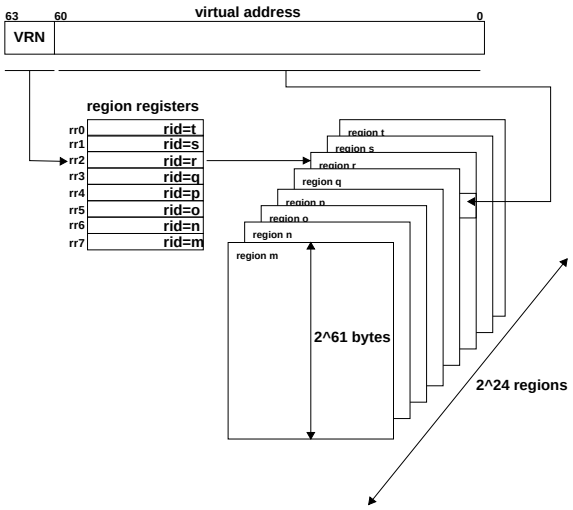

Figure 4: Virtual address space

2.5.1 Virtual memory

IA-64 provides a flat linear 64 bits address space. The Translation Lookaside Buffer (TLB) supports different page sizes (from 4KB to 256MB) and is composed of a translation cache (TC) and a set of translation registers (TR) used to pin entries. There are at least 8 instruction TRs (ITR) and 8 data TRs (DTR). Itanium has 8 ITRs and 48 DTRs. The size and structure of the TC is implementation specific, for instance Itanium implements a 2 level data TC (with 32 and 64 entries respectively) and a single level instruction TC with 96 entries. IA-64 also supports a Virtual Hash Page Table (VHPT) which is an extension of the TLB, i.e., a hardware walker, which looks for mappings in memory. As its name indicates it is mapped into the virtual address space.

The 64 bits address space is split into 8 regions of 2^{61} bytes each. The upper 3 bits of the virtual address are used to select the virtual region number (VRN) as shown in Figure 4. Each VRN identifies a region register (RR) which contains a 2^{24} bits wide unique region identifier (RID). Thus, at any time, a program can access up to 8 of the 2^{24} possible regions. Those values represent the architected maximals, Itanium, for instance, implements:

- 2^{54} bytes virtual address space (54=3+51)
- 2^{44} bytes physical address space out of 2^{63} bytes possible. The 64th bit is used as a memory attribute, i.e., cached/uncached
- 18 bits for the width of the RID

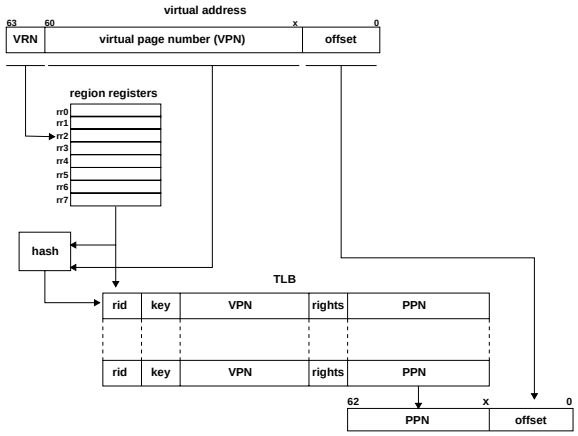

Figure 5: TLB lookups

A virtual address is decoded to yield the physical address as described in Figure 5. Given that multiple page sizes are supported, the width of the VPN field can vary and is denoted in the figure by the `x` symbol. The virtual page number (VPN) as well as the region identifier (RID) are hashed and a matching entry is searched in the TLB. If a match is found and permissions and protections keys are valid, you get the physical address by concatenating the physical page number (PPN) with the offset. If a translation is not found and if the VHPT has been enabled, the processor looks in the memory structure for a match. If a match is found it is automatically inserted into the TC. Otherwise a TLB miss fault is generated and the software miss handler is invoked.

Protection keys registers (PKR), just like protection identifiers (PID) for PA-RISC [10], provide an additional method to restrict permissions by tagging virtual pages with a unique identifier. The architecture requires at least 16 keys and the maximum width is 24 bits. Itanium implements 18 bits. If the executing context possesses the key and assuming regular permissions are validated, access is granted, otherwise access is rejected. It is up to the OS to manage those keys.

2.5.2 Interruptions

IA-64 understands four types of interrupts: aborts, interrupts, faults and traps.

Aborts are generated by hardware error machine checks (MCA) or processor resets and are handled by the processor specific firmware layer called PAL [7] (Processor Abstraction Layer).

Interrupts are asynchronous events generated by I/O devices, platform management interrupts (PMI) or initialization interrupts (INIT). They are handled by an interrupt vector table (IVT) that the OS defines and which is pointed to by the `cr.iva` register. To allow for efficient interrupt processing, this table contains actual code and not just an indirection. Some entries can have up to 64 bundles and others up to 16. If the interrupt processing is short, it can be entirely treated inline without any costly branching outside of the table.

Faults occur when an action cannot be accomplished and are thus synchronous with the execution. Finally traps occur when an operation requires software assist, like some floating point operations or when single stepping. Both faults and traps are dispatched via the IVT.

Interrupts from devices are called external interrupts. IA-64 supports up to 256 such interrupts grouped in 16 priority classes of 16 each. To help get efficient interrupt processing, IA-64 provides the OS programmer with a set of shadow registers after the interrupt occurred. Registers `[r16-r31]` are, in reality, banked registers, i.e., you get 2 sets. On interrupt, the processor automatically switches from bank 1 to bank 0, actually giving you 16 “free” registers.

2.5.3 IA-32 emulation

Although IA-64 is a completely new architecture, it allows IA-32 applications to run unmodified on top of an IA-64 OS using a hardware assist.

Clearly, the goal is to allow progressive transition of applications from 32 to 64 bits. In the context of Linux this is not that big an issue because for most applications, source code is available and recompilation or adaptation to 64 bits is possible. But for a small set of applications, like Netscape Navigator, where only a binary form is available, it is of high importance to make them run with no modifications right away.

The processor can be set in two modes of executions, IA-64 or IA-32. The Instruction Set bit (`IS`) of the PSR indicates the current mode of execution. To switch from one mode to the other, you first need to setup the correct register state and then you can use one of the three switching instructions to toggle the mode. At the user level, `br.ia` is used to switch to IA-32 and `jmpe` to switch to IA-64. At the kernel level, the `rfi` instruction can be used to switch to either mode based on the IS bit of the PSR that is being restored.

2.5.4 Performance monitors

For IA-64, most of the performance responsibility is on the compiler. A lot of information can be extracted from the source but oftentimes, it is necessary to actually run the program to see how it behaves. Once traces are collected they can usually be fed back into the compiler using a technique called Profile Based Optimization (PBO) which most modern optimizing compilers support. The IA-64 architecture provides a unique set of tools to help programmers get detailed profile information about the behavior of a program.

The Performance Monitoring Unit (PMU) offers a set of registers called Performance Monitoring Data (PMD) and Configuration (PMC) registers which can be programmed to capture single or multi-occurrence (per cycle) events. Depending on how you setup the registers, you can monitor only your application while it's running in user mode or only in kernel mode or both. You can also monitor only the kernel or the entire system. You can get a complete cycle break down of the program. Some counters have thresholding capabilities that let you count how many times did event A take more than X cycles. You can restrict the monitoring to specific piece of code or data. You can also monitor a set of instructions by specifying a template opcode. Although cycles accounting gives you a picture of how the programs behaves, it does not really help you find where the bottlenecks are. Some of the PMD/PMC are called Event Address Registers (EAR) and they are used to record the code location of where some cache or TLB miss events occurred.

For obvious space reasons we cannot go into more details about all the other aspects of the architecture but you can refer to [5, 6, 7] for more information. We have covered just enough to go into the kernel design description.

3 Kernel internals

The work on the kernel started almost two years ago at HP Labs using the HP IA-64 instruction set simulator (ski). The goal of the project was to produce a straight port of Linux to IA-64. This means that we have tried to minimize as much as possible changes to the machine independent part of the kernel. Most of the new code is located under `arch/ia64` and `include/asm-ia64` directories. We also decided to follow closely the development branch of the official kernel, i.e., the 2.3 branch. Although this may seem like an extra overhead due to the rapid evolution of this branch, it turned out that, in the end, it was a win because the final integration into the mainline went very smoothly. Since February of this year and as of 2.3.42, most of the IA-64 patch has been merged into the mainline kernel by Linus. We are constantly tracking the current development (we are at 2.4 today) and update our kernel patch accordingly. A usable Linux/ia64 source tree can be found at the official kernel web site at http://www.kernel.org/[4].

In this section we will cover some of the key machine dependent subsystems of the kernel.

3.1 General properties

Type	Size	Alignment
`char`	1	1
`short`	2	2
`int`	4	4
`float`	4	4
`long int`	8	8
`long long int`	8	8
`void *`	8	8
`double`	8	8
`long double`	16	16

Table 1: Various data types

The Linux/ia64 kernel has been designed from the beginning as a 64-bit operating system. It uses little-endian byte ordering model for obvious compatibility reasons with IA-32. IA-64 can be configured to run in big-endian as well, so with little extra support, it is conceivable to have big-endian processes.

Linux/ia64 uses the `LP64` data model like all other 64-bit Unix systems on the market. This means that `long` and pointer variables are 64 bits whereas `int` variables are 32 bits. Table 1 defines the sizes of the various data types[5].

Wherever possible we have tried to follow all the standards defined for IA-64. This includes the software calling conventions [4], the application binary interface definition [9] (ABI) and also the Developer Interface Guide [1] (DiG64) which establishes a set of basic system building blocks and defines the required and optional interfaces between the hardware, firmware and the OS. By being DIG64 compliant we ensure that

[4] look into pub/linux/kernel/ports/ia64

[5] for `long double`, current `gcc` limits size=8, align=8

Linux/ia64 will run on any hardware that follows the guidelines.

In order to make porting of existing Linux applications easier, we have strived to make ABIs compatible with the Linux/ia32 whenever feasible. For example, we picked identical numerical values for `ioctl()` commands, signal numbers and error codes, for instance. This is especially useful for "not so well behaved" applications which have hardcoded those values.

3.2 Register usage

Name	Content
`r13`	current task pointer
`ar.k0`	legacy I/O base address
`ar.k5`	floating pointer high partition owner
`ar.k6`	physical address of current task structure
`ar.k7`	physical address of page table

Table 2: Kernel register usage

One of the benefits of having a large number of registers is that you can dedicate some of them for a single purpose, thus avoiding memory latency to reload the information each time it is needed. In the kernel, we decided to fix the usage of a few registers which point to information that is heavily used. The best example is the `current` global variable which points to the currently executing process. It is referenced many times throughout the code. Interestingly enough, one of the general registers, i.e., `r13`, is also known as the thread pointer, so we decided to use it to hold the `current` task pointer. To tell the compiler not to touch this register, we use the `-ffixed-`*X* option of `gcc`, which forbids the usage of the register specified (*X* here). Table 2 summarizes current fixed register usage. We'll see later what `ar.k5` and `ar.k7` are used for.

For optimization reasons, we also restrict the kernel from using some floating point registers (`[f10-f15]` and `f32-f127]`). Using floating point variables inside the kernel is never a good idea. However on IA-64, some integer operations, like multiplication, are performed in floating registers. For this reason, the rest of low partition is accessible to the kernel.

3.3 Process subsystem

In the Linux kernel, each process is represented by a `task_struct` data structure which encapsulates most

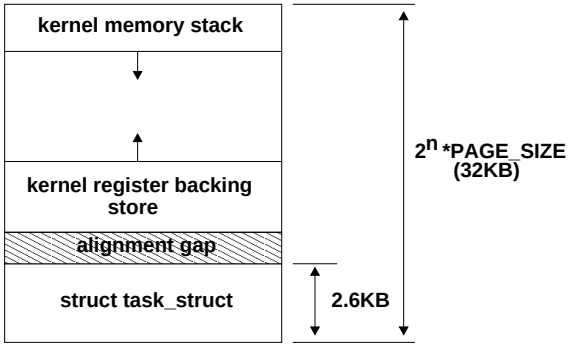

Figure 6: The `task_struct` allocation

of the state. We have seen that IA-64 requires every process to have 2 stacks, this yields a total of 4 stacks to deal with between user and kernel mode executions. When entering the kernel, the memory and register backing store stacks are switched from the user to the kernel pair. To simplify the allocation of the kernel stacks, we "bundle" them with the `task_struct` as shown in Figure 6. The software convention dictates that a memory stack grows down whereas the backing store grows up. To detect collision we made them grow towards each other.

The register context of a process is decomposed into 3 data structures in a typical Linux kernel. The layout of the each structure is strongly influenced by the calling convention which defines which registers are scratch, i.e., saved by caller (free to use directly), versus preserved, i.e., must be saved prior to being used. The layout we use is as follows:

1. the `struct pt_regs` ($\approx$ 400 bytes), which contains mostly the scratch registers, is saved every time the kernel is entered (synchronously or asynchronously).

2. the `switch_stack` ($\approx$ 500 bytes), which mostly contains the preserved registers, is only saved when the process goes to sleep, i.e., there is a context switch, or during some debug operations.

3. the `thread_struct` ($\approx$ 1700bytes) contains additional state like the debug registers, kernel memory stack pointer and the high floating point partition.

Except for the `thread_struct`, which is part of the `task_struct`, the rest of the state gets saved onto the kernel memory stack of the process.

As alluded to earlier, we use a lazy method to save the floating point state, i.e., we save it only when this becomes necessary. This means that the floating point context of a process can still be live in the CPU even though the process has been switched out. This technique is used only for the high floating point partition which includes `f32-f127` and represents 1.5KB when saved to memory. To guarantee that the state is not lost, we simply disable the high partition by setting `PSR.dfh` to 1 when a process is switched out. As soon as another process needs to access a register in that partition we get a fault. At this point, we save the state of the original process, which task structure is pointed to by `ar.k5`, into `thread_struct`. After saving, we change the ownership and re-enable the partition. So unless another process requires the partition, no saving occurs. As of today, we save this partition on every context switch in case of an SMP kernel. It should be noted that we use a similar "lazy" technique to save the debug registers.

3.4 System calls

```
     alloc r2=ar.pfs,1,0,8,0
     mov r15=1028
     break.i 0x100000
     cmp.eq p6=-1,r10
(p6) br.cond.spnt __syscall_error
     br.ret.sptk.many b0
```

Figure 7: System call stub

User and kernel mode executions do not happen at the same privilege level for obvious security reasons. To enter the kernel for a system call, a process has to go through a special stub usually found in the C library. On IA-64, one way to enter is to cause a trap with a `break` instruction as shown in Figure 7. On a trap, execution continues in the interrupt vector table (IVT) in the `break` entry. There, the code checks whether or not this is an actual system call by looking at the `break` value which, in this case, is `0x100000` as defined by the ABI manual [9]. Register `r15` holds the identifier for the call: here `1028` corresponds to `open()`. This value is an index into the system call table. Once the function is located, it is simply called. Upon return, we check for errors and set `r10` to be `-1` in case of failure. The error code, `errno`, is contained in `r8`. To restart execution in user mode, the kernel restores the user state and eventually executes a return from interrupt (`rfi`) instruction.

Systems calls are very often part of the critical path of an application so we must make sure their invocation is as lightweight as possible. An important factor is how parameters are passed between user and kernel mode. As per the calling convention, parameters are generally passed in stacked registers. In the case of system calls however, this is a little bit more difficult because of the way the call is implemented.

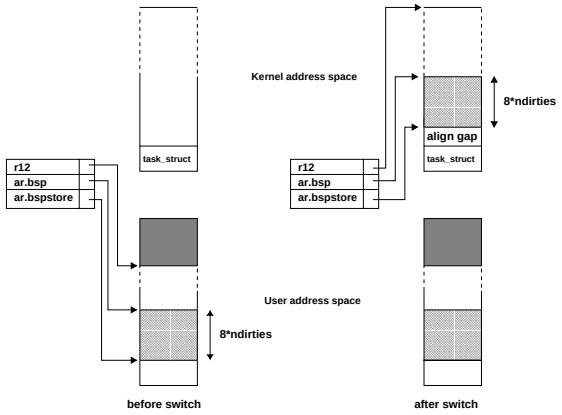

Figure 8: Stack swicthing during syscall

A `break` instruction causes a synchronous interruption of the execution. Thus when we enter the kernel, the state of the registers is as follows:

- preserved: if the assembly code does not need them then we have nothing special to do. They will be saved as needed by the C compiler.
- scratch: we need to save them for tools, like the debugger, which need to access the process state. They end up being saved on the `pt_regs` structure.

During system call, we must make sure that system integrity is not compromised by kernel state flowing back to the user for no reason. Therefore stacks must be switched. Special care must be taken with scratch registers as well. Today, because we save them on entry and restore them on exit, no kernel state transpires. However in future optimizations we will likely try to avoid saving scratch registers but we will still need to, at best, clear them on exit.

Switching the memory stack simply requires changing the stack pointer `r12`. For the register backing store, it's a little bit more difficult because it can operate in the background. Figure 8 depicts what happens when we switch RBS. You must first stop RSE operations, i.e., put it in enforced lazy mode, so that it does not try to spill unless necessary. We pay special attention not to trigger any request for more stacked registers, i.e., no `alloc`.

At this point `ar.bsptore` needs to be switched to the kernel backing store (KRBS). This will automatically cause `ar.bsp` to move as well, such that:

```
ar.bsp=ar.bsptore + ndirties*8
```

In the equation, `ndirties` represents the number of registers (each being 8 bytes long) that were still "live" from previous stack frames and awaiting to be spilled. Once `ar.bsptore` is switched, we can restart RSE operations. In the case of a system call, `ar.bspstore` is always switched to point to the next 16-byte aligned address above the `task_struct`, i.e the KRBS base, as we described in the previous section. The memory stack is switched to point to the top of the 32KB region, which starts at the `task_struct`, as it grows down. The first data structure stored on this stack is the `pt_regs`. One interesting aspect of this "exercise" is that some user state may have been saved in the user backing store while the rest will be saved in the kernel backing store as execution continues. So upon leaving the kernel, just before switching back the the user backing store, we make sure that whatever might have been saved onto the KRBS gets reloaded back into the physical register set using the `loadrs` instruction.

During this switching phase, the parameters to the system call are completely preserved in the stacked registers. Once we are ready to call the first C function, i.e., the actual syscall, we simply need a `br.call`. This is very efficient however there is one known problem: system calls can be restarted. Normal C convention allows parameters to a function to be modified as any other local variables. Although this is perfectly fine for regular functions, this causes some problems on system calls because of the restart mechanism. We pass system calls arguments straight from the user code, we have no private copy of their values upon entry, so if they are modified and the kernel needs to restart we are in trouble. The solution we use is to hint the compiler about the nature of certain C functions within the kernel. All system calls use a special attribute named `syscall_linkage`. This informs the compiler that parameters are to be treated as read-only: if a modification is required then a copy must be made.

As a future optimization, we plan on using the enter privilege code `epc` instruction. This one is similar to the `B,GATE` instruction of PA-RISC. It raises the current privilege to the one of the page it is placed on (would be 0 to enter the kernel). The major difference comes from the fact that the entire system call now looks like a function call, no more interruption thus more of the context can be assumed as saved by the compiler, i.e., less work to do on every call. Moreover this instruction does not generate as much disruption at the microarchitecture level compared to an interruption.

3.5 Virtual memory

As described in Section 2, the architecture supports different page sizes. The current mainline kernel however, does not have support for variable page sizes. The Linux/ia64 kernel supports compile-time configurable page sizes. It currently supports sizes: 4, 8, 16 or 64KB. From a programmer's point of view, this means that any program relying on knowledge of the page size, must use the `getpagesize()` system call instead of hard-coding a value. The reason for supporting different page sizes are as follows:

- if you want to minimize TLB misses, you want larger page sizes especially knowing the code expansion factor of IA-64 vs. IA-32. Page sizes of 8 or 16KB are better for native binaries.
- larger page size yields larger address space (see later).
- 4KB page size is the best for IA-32 emulation.

Reg	Usage	pg_size	Scopel	Map
7	cached	256MB	G	I
6	uncached	256MB	G	I
5	vmalloc guard gate	8KB	G	P
4	stack segment	8KB	P	P
3	data segment	8KB	P	P
2	text segment	8KB	P	P
1	shared memory	8KB	P	P
0	IA-32 emulation	8KB	P	P

scope: G=global, P=process; map: I=identity, P=page table

Table 3: Regions usage

The IA-64 architecture splits the virtual address space in 8 regions. Table 3 gives the breakdown per region for Linux/ia64. The top two regions are for kernel use only. The cached and uncached regions are identity mapped and are used to access physical space with a different caching policy. The kernel code and data reside in region 7. To limit the number of TLB entries consumed by the kernel we use a large page size and we pin one entry for data and one for code using translation registers (TR).

Region 5 is reserved for the kernel `vmalloc()` allocator. It also holds the gate page used for signal delivery and potentially the future `epc` code (see section 3.4). This region being directly above the user regions, it contains a guard page (no access permissions) which is used to protect against malicious buffer overruns. It also allows to speed up all the routines doing copy from user mode because it greatly simplifies bounds checking:

1. before copying we simply check whether the start of the buffer is below `TASK_SIZE`[6] (base of region 5). If so we start copying
2. Linux uses an exception scheme to copy to/from user: just do the blind copy and if something goes wrong, the exception will be generated and execution will abort with an error code. The guard page simply ensures that if the first test passes but the length of the buffer exceeds the boundary then it will be caught by the exception mechanism even when running at the most privileged level.

For regions 0 to 5, the page size is taken from the kernel configuration. Region 5 has its own kernel-wide page table whereas regions 0 to 4 share the same per process page table. Regions 1,2,3,4 are used for IA-64 process execution. The stack region contains both the memory stack and the register backing store growing towards each other. Finally region 0 is reserved for Linux/ia32 binaries. It is large enough (2^{40} bytes) to encapsulate the largest IA-32 address space and does not require any modification to existing programs.

Although each region is 2^{61} bytes, Linux cannot take full advantage of it because of the current design of the virtual memory subsystem which is based on a 3-level forward mapping page table. In the following discussion we assume the page size is set to 8KB.

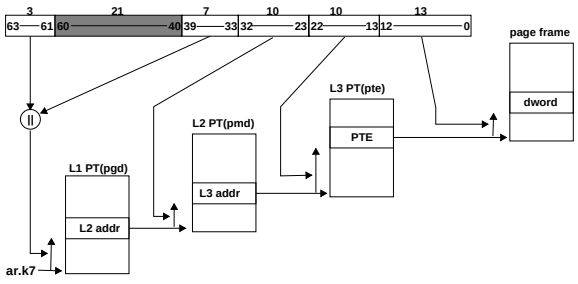

Figure 9: User region page table

In order to use only one page per level each intermediate level represents only 10 bits of the virtual address, i.e., how many 8-byte pointers you can fit into an 8KB page ($2^{13-3} = 2^{10}$). Therefore we have a virtual address space of: $2^{30} * 2^{13} = 2^{43}$ bytes. Each user region ([0-4]) represents $1/8$th of the 43 bits space because of a single page table.

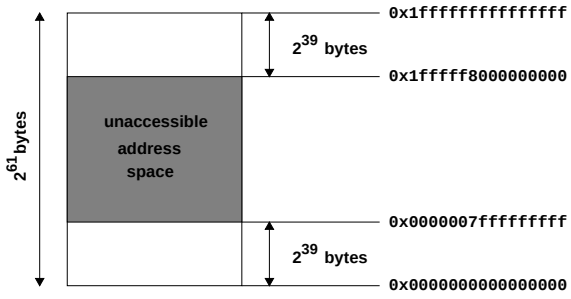

Figure 10: User region layout

Figure 9 shows how a user region is broken down in 3 levels. As we mentioned earlier, `ar.k7` is pointing to the base of the page table tree for the current process. The top level page table (pgd) offset is constructed by a logical or of the 3 bits of the virtual region number (VRN) and bits [33-39]. Bits [40-60] are a signed extension of bit 39 which creates a software imposed unaccessible region in the middle of the address space. This gives the layout depicted in Figure 10 where you have a large hole where any access would generate a `SIGSEGV`.

In contrast, region 5 has its own page table pointed to by `SWAPPER_PG_ADDR`, therefore it has access to the full 43 bit address space as depicted in Figure 11. Similarly, bits [43-60] are a signed extension of bit 42.

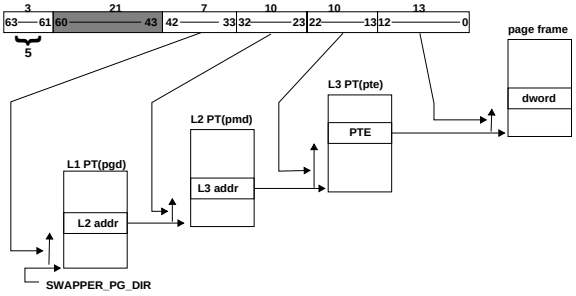

Figure 11: Region 5 page table

In this scheme, increasing the page size actually increases the size of the address space available because the intermediate levels can store more pointers. Table 4 shows the possible configurations.

In section 2 we introduced the virtual hash page table (VHPT) which is used by the TLB hardware walker to

[6] `TASK_SIZE`=0xa000000000000000

Page size	Address space size (bits)
4KB	39 (512GB)
8KB	43 (8TB)
16KB	47 (128TB)
64KB	55 (32PB)

Table 4: Page size/address space

extend TLB lookups in memory. Linux/ia64 takes advantage of this feature for regions [0-5]. The VHPT can be configured in two different modes:

- in the long mode, it behaves like a global hash table for the entire address space. It is virtually anchored in one of the eight regions.
- in the short mode, you have one linear page table per region which typically consists of the leaf level of the OS-maintained page table. For each region, the table is anchored into the address space.

In either mode, the table is virtually mapped therefore an access by the walker requires a TLB entry as well. The architecture provides separate fault handlers to help distinguish what is going on. The kernel also makes sure there is no need for a page table to get the mapping for the walker page.

For Linux/ia64 we decided to use the short mode because it is a perfect fit for the way virtual memory is managed: forward mapping page table. The leaf pages, i.e., L3 in Figure 9, are installed into the user region to satisfy a VHPT walker TLB miss. The structure of the information in L3 PTE has been designed to match exactly the one expected by the walker. One benefit of this approach is that for every VHPT miss, you actually make "visible" to the walker 1024 mappings (with 8KB pages) covering about 8MB (1024*8KB) worth of address space. Thus if the address space is densely populated you, indeed, reduce your chances of getting subsequent TLB misses for that area. The downside is that you may need up to twice as many TLB entries as with the long mode. Also this mode is less flexible in terms of variable page sizes per region.

The table must be mapped inside each region. A good place to put it is where the user has no access, i.e., in the hole created by the sign extension of bit 39 as depicted in Figure 12.

As alluded to earlier, there is an architected maximum virtual address space size of 2^{64} where 3 bits come

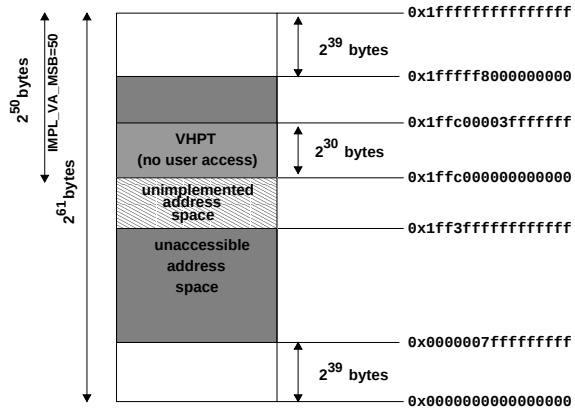

Figure 12: VHPT in a user region

from the region number (VRN). Implementations are free to implement fewer bits but never less 50. The constant `IMPL_VA_MSB` identifies the most significant bit of the implemented address space (other than the VRN), so for Itanium which implements 54 bits, this value is 50, i.e., 51-1. The architecture defines bits between [60-`IMPL_VA_MSB`+1] to be a sign extension of `IMPL_VA_MSB`. Any other combination would generate a fault. Just like we "virtually" create a zone of no access, this restriction creates yet another one which is much smaller and represents the "real" unimplemented area of the virtual address space.

We use the area where bits [51-60] are set to place the VHPT in each region. This puts the table just above the unimplemented virtual address space hole. To prevent any malicious access by the user, the VHPT is mapped with no access at privilege level 3 (user). Any access would end up in the process being killed by a `SIGSEGV`.

In the short mode, each VHPT entry (1entry/page) is exactly 8 bytes, thus the maximum size (with 8KB pages) of the VHPT is: $2^{40-13+3} = 2^{30}$ bytes. So there is enough room to fit it above the unimplemented virtual address space and below the top user accessible region. The same scheme is used to place the VHPT for region 5.

To take full benefit of the architecture, future changes potentially include going to a 4 level page table with the top level used to index regions giving full 43 bits worth space per region. This requires of course a redesign of the generic Linux VM subsystem and would need to be coordinated with all other platforms.

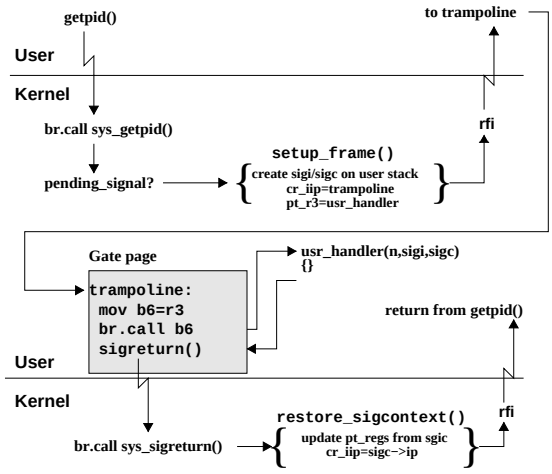

Figure 13: Signal handling

3.6 Signal handling

When calling the user handler for a signal, Linux writes a `siginfo` and `sigcontext` structures on the user stack. It also dynamically generates, on the same stack, the stub code to call the handler and return back to the kernel via a `sigreturn()` system call. The, potentially modified, `sigcontext` is used to update the currently interrupted context and eventually execution resumes at the user level.

For the Linux/ia64 kernel, the principle remains the same. The implementation, however, differs slightly from the above model. We do not generate stub code on the user stack because it is not very efficient and would require a cache flush and potentially a TLB miss every time a signal is delivered. Instead, we chose a, somewhat more elegant, scheme which uses a code trampoline installed in the address space of each process.

A sample signal delivery is shown in Figure 13, with a pending signal detected on return from a `getpid()` system call. The first step is to generate the stack frame needed by the user handler. This task is accomplished by the `setup_frame()` function. Both the `siginfo` and `sigcontext` structures are generated on the user memory stack, as usual.

Next, instead of generating code, we simply modify the current `pt_regs` structure, which contains most of the information needed to get out of the kernel, to resume execution at the trampoline code. This is accomplished by modifying the `cr_iip` field which represents where the interruption occurred and also where to resume. When leaving the kernel, the `cr_iip` field is copied back into `cr.iip` then used by `rfi` as the resume point.

The trampoline is code installed in the gate page which is execute-only at the user privilege level. As described in Section 3.5, this page is anchored in region 5, which is globally shared. The trampoline code "accepts" three parameters passed in registers via the `pt_regs`: the signal number, the address of the C handler and where the `siginfo` and `sigcontext` are on the stack.

The first thing this code does is to setup the parameters for the C handler then it does the call. Upon return it calls the `sigreturn()` system call and passes the `sigcontext` via the user memory stack. Back in the kernel, the `pt_regs` context is updated with the content of `sigcontext` then the normal return from system call code is called and eventually the `rfi` instruction is executed. Unless the `iip` field of `sigcontext` was modified, execution will resume at the return from the `getpid()` system call in our example.

3.7 IA-32 support

Using the code contributed by Intel and VA Linux Systems, the Linux/ia64 kernel can be configured to run pure, i.e., no bi-mode processes, Linux/ia32 ELF binaries using the `CONFIG_IA32_SUPPORT` option. Although the CPU is able to run native IA-32 code directly, some kernel support is needed to load the binaries in memory and provide the Linux/IA-32 system call API. We only support IA-32 Linux user mode applications, you cannot insert an IA-32 kernel module.

Linux/ia64 incorporates two loaders: one for ELF32 (IA-32) binaries and one for ELF64 (IA-64) native binaries. On `execve()` the kernel checks the type of a binary and dispatches to the correct loader. The file image is then loaded in memory and the application registers (AR) pertaining to the IA-32 emulation, like `ar.eflag` for `EFLAGS`, are initialized. Eventually the initial thread is created with the IS (Instruction Set) bit of `cr_ipsr` set to 1 in `pt_regs` indicating IA-32 instruction set is to be used upon `rfi`.

From now on the process is running in IA-32 mode. From a user's point of view it is just a matter of invoking the program at the shell prompt. Dynamically linked IA-32 binaries are supported. The IA-32 dynamic loader `ld-linux.so.2` is used. To avoid a name conflict, the IA-64 loader is called `ld-linux-ia64.so.1`. The runtime linker program support for architecture spe-

```
% /sbin/ldconfig -p
libc.so.6 (libc6)
=> /usr/ia32-pc-linux/lib/libc.so.6
libc.so.0 (libc6,IA-64)
=> /lib/libc.so.0
libm.so.6 (libc6)
=> /usr/ia32-pc-linux/lib/libm.so.6
libm.so.0 (libc6,IA-64)
=> /lib/libm.so.0
...
```

Figure 14: Architecture specific libraries

cific shared objects has been enhanced slightly, especially the search algorithm. Figure 14 shows an excerpt of the output of the `ldconfig` command where you can see the distinction between architectures. When searching for `libm`, for instance, the linker, will also check the architecture to see whether or not it matches the architecture of the binary. If not, it will continue searching. The figure also shows that IA-32 libraries are placed in a different directory than the IA-64 ones. It is to avoid any name conflicts. The specific location used is up to the system administrator and is indicated in the `/etc/ld/so.conf` file as usual.

During its execution, the IA-32 process is going to make a system call using the IA-32 software triggered interruption instruction; `int 0x80`. At this point the Linux/ia64 kernel takes over and execution continues in the IVT in IA-64 mode at the IA-32 Interrupt entry. The handler then checks whether or not the interrupt was generated for a system call: was it vector `0x80` ? If that's the case, the IA-32 system call number normally in `EAX` and hosted in `r8` is used to index the IA-32 specific system call table. The IA-32 system call arguments are "hosted" on IA-64 fixed registers, i.e., not stacked. Therefore they must be copied from the `pt_regs`, where they were saved on entry, to stacked registers to call IA-64 code. It should be noted that most of the principles described for IA-64 system calls still hold here: the stacks are switched.

Because IA-32 and IA-64 Linux implementations use a different data model, i.e., LP32 versus LP64, each IA-32 system call must be inspected to look for potential differences. Problems are only encountered when dealing with `long` integers and pointers because their sizes differ in the two models.

For many system calls there is no impact because there is either no parameters, like for `getpid()`, or the types of arguments or data structures passed are identical in both cases.

```
struct timeval32
{
   int tv_sec, tv_usec;
};
struct timeval {
   long       tv_sec;
   long       tv_usec;
};
static inline long
put_tv32(struct timeval32 *o,
         struct timeval *i)
{
   return
   !access_ok(VERIFY_WRITE,o,sizeof(*o))
 || (__put_user(i->tv_sec,&o->tv_sec)
  | __put_user(i->tv_usec,&o->tv_usec)));
}
asmlinkage long
sys32_gettimeofday(struct timeval32 *tv,
                   struct timezone *tz)
{
   struct timeval ktv;
   do_gettimeofday(&ktv);
   if (put_tv32(tv, &ktv))
      return -EFAULT;
   ...
}
```

Figure 15: 32/64 bits conversion

For others, like `gettimeofday()` a systematic conversion is needed because the data structure used in each case is different. In figure 15 we give the example of `gettimeofday()` where the `timeval` structures differ: `long` fields (64bits) must be converted back to `int` (32bits).

Finally, some system calls vary depending on the value of the arguments passed. For example, with `ioctl()` the type of the third argument depends on the command type, i.e., the 2nd argument. This requires a case by case inspection of all possible combinations. Fortunately the cases where conversions are needed are rare. Our experience so far seems to verify this hypothesis.

One of the benefits of using region 0 for IA-32 binaries is that the kernel does not have to worry about 32-bit pointers interpreted as 64-bit pointers. Everything is passed in 64-bit registers and the top 32 bits will be zero no matter what because a load of 4 bytes into an 8-byte registers will automatically clear the top 4 bytes.

Another design choice we made was to reuse as much as possible some of the constants used on Linux/ia32. This is true for `ioctl()` command code, signal numbers, errno values. Thus no value conversions are required.

Another interesting system call is `mmap()` when the page size is different from the native IA-32 value of 4KB. For applications which don't specify a fixed map address, there is no problem. For others which ask for a 4KB aligned address, the `mmap()` shim code falls back to doing a copy from the file instead of a pure memory mapping when the kernel is not using 4KB pages.

So far this code allowed us the run serious IA-32 applications like Netscape Navigator, AcrobatReader, Applix-5.0, WordPerfect and also some simpler tools like the IA-32 toolchain.

There are many other aspects of the kernel that would need to be explained, like how we interact with the platform firmware, how do we boot, how do we figure out how the machine looks like in terms of devices, etc ? For space reasons we do not cover these here but all the kernel code is publicly available and is a unique source of information. Also there is a mailing list available for discussing Linux/ia64 related topics. All the information to subscribe is available at http://www.linuxia64.org.

4 User land

The kernel is a key piece of the system but, in reality, it represents only a small portion of what needs to be present to get a complete system. In this section, we give a quick overview of where we are in terms of user level support: libraries, tools, environments, etc.

4.1 Libraries

Most of the key libraries are available in both static and shared forms. The current IA-64 libc/libm is based on GNU libc v2.1. The port of the version 2.2 is in progress. Some EPIC optimizations are present in the C library for some performance critical routines, like `memcpy()`, `strcpy()`, etc. The default math library however is not optimized at all and uses the generic C code. When the kernel source got released, Intel contributed an optimized version of the math library which provides the same API. It is entirely written in assembly language and highly optimized. The pthread support is present in the current `libc` and is mostly derived from the Java work going on at HP Labs.

4.2 Development tools

As we mentioned during our introduction to the architecture the compiler is the key component when it comes to exploiting the capabilities of the machine. Today everything is based on the GNU compiler that Red Hat (formely Cygnus) has been working on and which is based on the early toolchain from HP Labs. This toolchain is robust enough to compile entire distributions and Linux kernels. Today, however, it does not yet support any of the EPIC optimizations like speculation, software pipelining, etc. It should soon get the support for predication and the rest is planned. Although it is lacking those key features, it produces decent code in terms of density (filling up the bundle correctly) and is very stable.

A few weeks ago SGI also released in source and binary forms their IA-64 compiler suite for Linux. It includes the C, C++ and also Fortran90 compilers. It is GNU compatible and can therefore be used to recompile most of the packages used by Linux. The current binary distribution is meant to be used with the IA64 SDK (see section 5) and is a cross compiler only at this point in time.

The other development tools available include the GNU debugger `gdb`, `strace` to get system call traces for both IA-64 and IA-32 binaries and also the GNU profiler, `gprof`.

A port of the Performance Counter Library[7] (PCL), contributed by IBM, exists and uses some of the capabilities of the Performance Monitoring Unit (PMU). The kernel provides the `sys_perfmonctl()` system call to control how monitoring is done.

For kernel developments, a kernel debugger, contributed by SGI and Intel, called `kdb` is available as a separate patch to the kernel. It provides assembly level debugging of a running kernel using the local console or a serial line. The latest version is available from the HP Labs FTP site[8].

4.3 X11 environment

We are currently using version 3.3.6 of the XFree86 distribution. Work is under way to get the latest 4.0 version working. The GNOME environment, including

[7] see http://www.fz-juelich.de/zam/PCL
[8] see ftp://ftp.hpl.hp.com/pub/linux-ia64

`enlightenment` and `gimp`, is available. Other window managers known to work include, `fvwm` and `WindowMaker` to name a few.

We have also successfully been able to get live video displayed with the `xawtv`[9] program. The `video4linux` kernel code was used to interface with either a USB web camera or directly with the `bttv` driver controlling a frame grabber card.

4.4 Java

Several Java implementations are available on Linux/ia32. Fairly direct ports of the Sun Java Virtual Machine are available from Sun itself and from `blackdown.org`. IBM has made available ports of their Sun-derived virtual machines. Several less complete open source Java environments, notably `kaffe`, are also available. We expect several of these to be ported to IA-64 by their respective developers.

At HP Labs, we have been focusing on the Java to native code approach using the `gcj` compiler. This is a front end to the GNU compiler which translates Java source code or bytecode into native code. The Java runtime environment is recreated using a runtime library, which includes an interpreter for dynamically loaded code. Although it currently provides incomplete library support, it offered us an existing IA-64 compiler back-end, the opportunity to immediately look at Java run-time issues with compiled client code, very competitive performance, fast and convenient multi-language support, and the ability to generate fast-starting native applications. We expect this may become an important complement to a traditional Java Virtual Machine.

With cooperation from Red Hat and others, we have successfully ported the runtime library to Linux/ia64. The compiler required essentially no porting beyond the (`gcc`) port itself. The IA-64 port is now part of the standard (`libgcj`) distribution. We have since been concentrating on enhancing the performance and multiprocessor scalability of the garbage collector, both generally, and on IA-64 specifically.

4.5 Distributions

All the major Linux distributors (Caldera, Red Hat, SuSe, TurboLinux) are part of the IA-64 Linux project and are actively porting their distributions to the IA-64 platform to be ready for product launch.

Today, the TurboLinux and Red Hat "alpha" distributions are directly available from their respective web sites. You need real hardware to use them but most of the packages could be used with the IA64SDK (see section 5). Although they are not yet fully complete and require semi-automatic installation, they all come with shared libraries, C/C++ development tools, X server, X desktop components, apache, perl, python, emacs and also some IA-32 packages.

5 The IA-64 Linux SDK

We recently released an IA-64 software development kit for Linux/ia32 (IA64SDK). This environment allows developers without IA-64 hardware access to create and port user level applications to Linux/ia64 on any Linux/ia32 systems. In this environment, not only can users compile applications and generate IA-64 binaries but they can also run those binaries directly at the shell prompt. Linux/ia64 kernel developments are also possible.

This package is based on the NUE (Native User Environment) developed at HP Labs during the early phase of the project when machines were not yet available and when the system was not yet self-hosted. The idea is to create an environment as close as possible from what a user would get on a real Linux/ia64 system. This means that you get a shell, you can type commands, edit files, compile programs and run them. The magic is to make this happen on an IA-32 system of today.

At the core of the package is the HP IA-64 instruction set simulator, which simulates an IA-64 CPU but not the entire platform (no BIOS nor PCI). This simulator has two modes of execution:

- the user mode: allows to run IA-64 applications directly on top of a Linux/ia32 kernel. The simulation stops at the system call level. Each call is translated into its Linux/ia32 equivalent. Most of the time only 32 to 64 bit parameter marshaling is required.
- the kernel mode: the entire IA-64 CPU is simulated including virtual memory and interrupts behaviors. In this mode, it is possible to boot a Linux/ia64 kernel.

[9] see http://me.in-berlin.de/~kraxel/xawtv.html

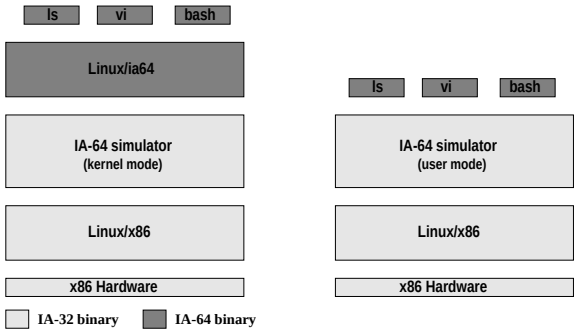

Figure 16: Execution modes

Linux provides a kernel module called `binfmt_misc` which allows one to dynamically bind command interpreters to binaries using the magic numbers. When you combine the batch mode execution of the simulator (running user mode) with this capability, you, indeed, get transparent invocations at the shell prompt. With this mechanism in place one can execute Linux/ia32 and Linux/ia64 binaries transparently. This provides the execution capability.

The other facet of NUE is to hide the cross-development nature of the platform. The IA-64 compiler must be called `/usr/bin/cc`, the linker `/usr/bin/ld`, the standard headers files must be in `/usr/include`, etc. This makes life so much easier when porting existing packages. In a typical cross-development environment those files are usually found in "exotic" places because they cannot conflict with the native system of the platform. Moreover a lot of fiddling is usually required to `Makefiles` to change pathnames. It is even worse with `configure` scripts, i.e., autoconf, because here, not only paths must be changed but also special care is required for the test programs used: they must capture IA-64 characteristics not IA-32. Many tests programs require actual execution to generate results, like a test for the size of a long integer, for instance.

To alleviate the problem NUE has its own "shadow" tree that is installed under `/nue`. In this tree, you find mostly Linux/ia32 binaries for tools like editors, network commands, etc. But `/nue/usr/bin/cc` is a Linux/ia32 binaries that produces IA-64 code, i.e., it is a cross-compiler. Under `/nue/usr/include` you find the Linux/ia64 headers files, under `/nue/lib` and `/nue/usr/lib` you find IA-64 shared and static libraries, etc.

The last piece of magic is to make this look like the native IA-64 environment by getting rid of the `/nue` prefix. This is accomplish by simply doing a `chroot(2)`

```
% /bin/arch
i686
% /usr/bin/nue
% /bin/arch
ia64
% ld -V
GNU ld version 2.9-ia64-000216
(with BFD 2.9-ia64-000216)
Supported emulations:
   elf64_ia64
% file /usr/bin/ld
ELF 32-bit LSB executable, Intel 80386,
version 1, dynamically linked, stripped
% cc hello.c -o hello
% file hello
hello: ELF 64-bit LSB executable, IA-64
version 1, dynamically linked
(uses shared libs), not stripped
```

Figure 17: output of commands in /nue

which is typically called by the `nue(1)` command to "enter" the environment. Once in NUE, the system acts like a Linux/ia64 systems as shown by the few shell commands in figure 17.

You can simply recompile existing RPM packages using a simple `rpm` command as shown in Figure 18 and unless some porting effort is required, you will get the equivalent binary RPM automatically.

Source level debugging of user applications is possible using the simulator explicitly on an IA-64 binary.

```
# rpm --rebuild 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 18: Rebuilding RPMs

Kernel developments are possible using the simulator. To get a working kernel you need to compile it for the HP simulator and enable the simulated device drivers we developed. Basically, you need a simulated serial console (`simserial`), a simulated scsi controller (`simscsi`) and, optionally, a simulated Ethernet controller (`simeth`) to get full functionality. More details on how some of those drivers work can be found in [2]. The SDK is shipped with a mini disk image containing a simplified TurboLinux/ia64 alpha distribution that can be used to boot the Linux/ia64 kernel. Even though hardware is now becoming available, we think that hav-

ing a simulator is very valuable for debugging purposes. It is extremely useful for debugging very low level code early on in the development cycle. Figure 19 shows the simulated serial console at the top and the simulator control window at the bottom.

Although we have been focusing on Linux development with this simulator, there is nothing that precludes developments of other Open Source OSes, like FreeBSD or NetBSD, for instance.

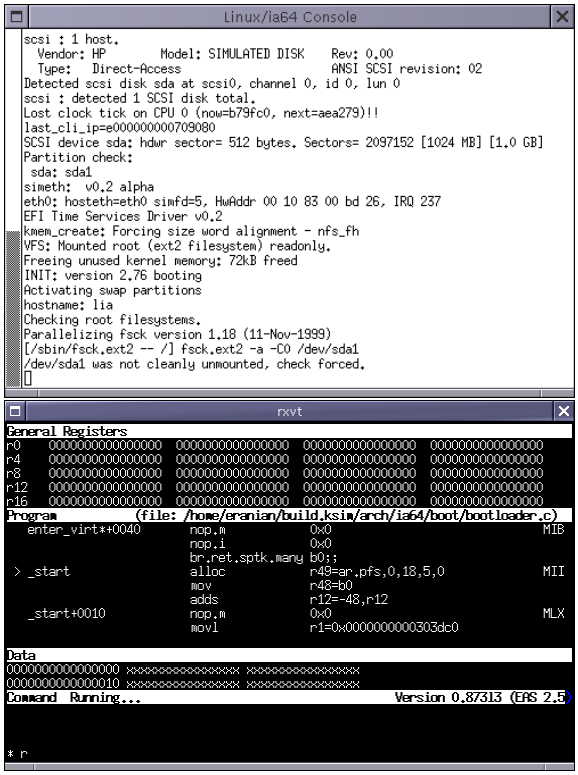

Figure 19: Booting Linux/ia64

The IA64-SDK is available in Red Hat RPM packages from the HP software depot web site[10].

6 Conclusion

In this paper, we have described some of the IA-64 system architecture features and how the Linux/ia64 kernel has been designed to use them as much as possible. We explained in details how the key kernel subsystems are working and the current status of the user mode support.

Since February 2000, as promised by the members of the project, the entire source code for the kernel and other key components of the system are publicly available making it the first OS available for this new architecture. The kernel support has been merged by Linus into the official mainline kernel.

Although there is still a lot of work to do in terms of performance tuning and debugging, the system is already very usable. Most of the packages one would expect to find on a Linux system are present including debuggers, toolchains and a graphical environment. We will continue to actively develop the system and expect to use it as the basis for our system research.

Distributors are working to get full distributions by the time the product comes to market such that it would be very easy to bootstrap systems.

Now that all the sources are available, it becomes possible for anybody to port existing applications or develop new ones for this platform. Although Itanium prototypes exist, they are still hard to get. To help foster Open Source developments, we have released an IA-64 software development kit which allows users with no hardware access to get involved with Linux/ia64 today by simply using their Linux/ia32 machines. This kit allows user and kernel level developments and is not limited to Linux.

All the pieces are now available to really jumpstart an active Linux community around this new platform. Our effort proves that the Open Source development model is also possible across the industry and that competitors can join forces and contribute a major piece of software back to the community.

Acknowledgments

We would like to acknowledge the members of the IA-64 Linux project for all their contributions which made it possible for Linux to be the first operating system available for this new architecture. The project members are Caldera, CERN, HP, IBM, Intel, Linuxcare, SGI, Red Hat, SuSe, TurboLinux and VA Linux Systems.

We also would like to thank Hans Boehm for developing the first Java environment for Linux/ia64 and for his contributions to the linuxthreads support.

[10] see http://software.hp.com/ia64linux

References

[1] Compaq, Dell, Futjitsu, Hewlett-Packard, IBM, Intel, NEC. *Developer's Interface Guide for IA-64 Servers (DIG64)*, November 1999.
http://www.dig64.org/.

[2] Stéphane Eranian and David Mosberger. The making of Linux/ia64. Technical Report HPL-1999-100, Hewlett-Packard Laboratories, August 1999.
http://www.hpl.hp.com/techreports/index.html.

[3] Stéphane Eranian and David Mosberger. Porting Linux to IA-64. Technical Report HPL-1999-83, Hewlett-Packard Laboratories, July 1999.
http://www.hpl.hp.com/techreports/index.html.

[4] Intel Corporation. *IA-64 Software Conventions and Runtime Architecture Guide*, August 1999.
http://developer.intel.com/design/ia-64/.

[5] Intel Corporation. *Intel IA-64 Architecture Software Developer's Manual Volume 1: IA-64 application architecture*, 2000.
http://developer.intel.com/design/ia-64/manuals/index.htm.

[6] Intel Corporation. *Intel IA-64 Architecture Software Developer's Manual Volume 2: IA-64 system architecture*, January 2000.
http://developer.intel.com/design/ia-64/manuals/index.htm.

[7] Intel Corporation. *Intel IA-64 Architecture Software Developer's Manual Volume 3: instruction set reference, Volume 4: Itanium processor programmer's guide*, January 2000.
http://developer.intel.com/design/ia-64/manuals/index.htm.

[8] Intel Corporation. *Itanium Processor Microarchitecture Reference for software optimization*, March 2000.
http://developer.intel.com/design/ia-64/.

[9] Intel Corporation. *Unix System V Application Binary Interface*, January 2000.
http://developer.intel.com/design/ia-64/.

[10] Gerry Kane. *PA-RISC 2.0 architecture*. Prentice-Hall, 1996.