Porting Operating System Kernels to the IA-64 Architecture for Presilicon Validation Purposes

Kathy Carver, IA-64 Processor Division, Intel Corporation
Chuck Fleckenstein, IA-64 Processor Division, Intel Corporation
Joshua LeVasseur, IA-64 Processor Division, Intel Corporation
Stephan Zeisset, IA-64 Processor Division, Intel Corporation

Index words: presilicon, validation, COSIM, postsilicon, Mach, Linux

ABSTRACT

To provide additional vehicles for presilicon validation and postsilicon debug of the Intel Itanium™ processor, we ported two operating system kernels to the IA-64 architecture. The Mach3* microkernel was ported first, followed by the Linux∗ 2.2.0 kernel, and these have helped track the overall health of the Itanium™ processor's RTL model for the last two years. These operating system (OS) kernels also helped presilicon performance analysis and compiler-generated code analysis.

The Mach3 kernel (the IA-64 port was called Munster internally) was ported because it contained features similar to Microsoft Windows NT[*], such as tasks, threads, interprocess communication (IPC), and symmetric multiprocessing (SMP). Mach3 allowed us to exercise parts of the Itanium processor's model in a similar way to Windows NT, but at a reduced scale and without device support.

Linux (the IA-64 port was called IPD-Linux) was ported because its source is readily available and 64-bit clean, it is highly configurable, and it would exercise the model in a different way than Mach3. We started with a released 2.2.0 version of the source and ported the kernel using a non-GNU C Compiler (GCC). The difficulty of porting the Linux kernel without GCC made the task more challenging.

Besides porting the architecture-specific portions of the kernels, modifications were necessary to both kernels to remove certain dependencies on external devices and BIOS initialization. Also, the OS initialization paths executed prior to user-level programs had to be shortened to accommodate the simulation speed of the RTL environment. The kernels had to be extremely configurable in order to run in diverse simulation environments.

Both kernels were tested from processor reset to user-mode code execution to validate the significant parts of the RTL that an operating system would exercise during the boot process. Kernel initialization, virtual memory management, context switching, trap handling, system call interfaces, and user-mode context paths were all exercised on the actual RTL model. This effort uncovered several errata in the RTL model and in the IA-64 tools (such as the compiler and linker). It also provided us a model regression sanity check for each new RTL release. We believe this presilicon effort was instrumental in allowing Windows NT to boot just days after first silicon. In this paper, we discuss the porting of kernels to the IA-64 architecture for presilicon operating system validation.

[*] Other brands and names are the property of their respective owners.

INTRODUCTION

One of the major goals for early silicon is to boot a commercial operating system (OS) shortly after the arrival of first silicon. In order to increase the probability of success we decided to use an operating system kernel to validate the processor in addition to using conventional presilicon testing methods. Traditional microprocessor validation includes feature validation, unit testing, and random instruction testing. The potential shortfall of these methods is that they often don't exercise the processor in the same environment in which it is later expected to run. In other words, an operating system programmer often thinks of a different, but legal way, of exercising processor functionality that might not be covered by conventional methods of validation. Therefore, running an operating system kernel to exercise key OS-related features presilicon turned out to be a worthwhile effort. The following sections detail the issues we had to resolve during this effort.

RUNNING AN OPERATING SYSTEM IN THE PRESILICON ENVIRONMENT

The two main constraints the presilicon environment imposes on an operating system are as follows:

1. The simulation speed of the RTL simulator effectively restricts test runs to a few million cycles of the simulated processor clock, and it causes turnaround times in the order of multiple days.
2. The simulated environment lacks devices.

The constraint in simulation speed had two major consequences for porting. First we had to reduce the number of instructions executed by the kernel during its initialization sequence. Second, we had to test the kernel thoroughly on functional simulators before committing it to a run on the RTL model.

To cope with the slow simulation speed, we wrote a tool that allowed us to run our kernel up to an arbitrary point in the functional simulator and then to continue simulation on the RTL model from that point on. To accomplish this, our tool read the saved architectural state from the functional simulator and used it to generate a sequence of IA-64 instructions that restored the architecture to this state. Then it read the memory image saved from the functional simulator and used it to generate a new binary, with the state restoration sequence placed at the processor reset vector. When we ran this new binary on the RTL model, the processor went through the state restoration sequence and then continued at the point where the state was saved on the functional simulator. Our two primary uses of this tool were (1) to skip the kernel initialization sequence and have the RTL simulation start directly with the execution of user-mode programs, and (2) to improve the latency for running the kernel initialization sequence on the RTL model by subdividing it into multiple parts and running the parts in parallel. To minimize the danger of processor errata being obscured by cold caches, we allowed for heavy overlap between the parts.

Reducing the Instruction Count

Our initial profiles of the kernel startup sequence for both Munster and IPD-Linux[*] showed that a large portion of time was spent in the routines for zeroing and copying memory, and in the initialization of a few key data structures, the most prominent being the structures used for virtual memory management. Our solution, therefore, included the following:

- Optimize the routines for zeroing and copying memory (bzero/memset, bcopy/memcopy).
- Reduce the amount of physical memory presented to the kernel. This reduced the time spent initializing page management information.
- Add delayed initialization for some kernel data structures.

These changes, however, did not reduce the functionality of the kernels.

One example of how we modified the Mach kernel to reduce the instruction count during kernel initialization was through changes to the zone allocation code. Most memory allocation for kernel data structures is done through zones, which act as buckets for fixed-size blocks of memory (zone entries) whose typical size ranges from a few bytes to a few hundred bytes. When an entry was allocated from a zone that had no entries in its free list, the free list was replenished by allocating one page of memory and splitting it up into zone entries, all linked together in a free list. The number of instructions required for doing this initialization for dozens of zones was quite high when keeping the speed of RTL simulation in mind. Therefore we changed the mechanism for replenishing a zone. Instead of immediately entering a whole page into the free list, a "free space" pointer was kept. The pointer was initially set to the newly allocated page, and it was used to carve out new zone entries one by one at the time they were actually needed.

Testing in Different Environments

Figure 1 shows our available simulation environments. Except for device support, matching environments were available on the functional and the RTL simulator. Even though no device support was available on RTL, we still needed to test our kernels with devices on the functional level to prepare for postsilicon.

[*] Other brands and names are the property of their respective owners.

Our kernels had to run in several different simulation environments. We ported the kernels so they could be configured with or without devices and run with or without external interrupts, etc. Our kernels were flexible enough to run in simulation environments that ranged from just one processor with memory to a full simulation of a multiprocessor platform with devices.

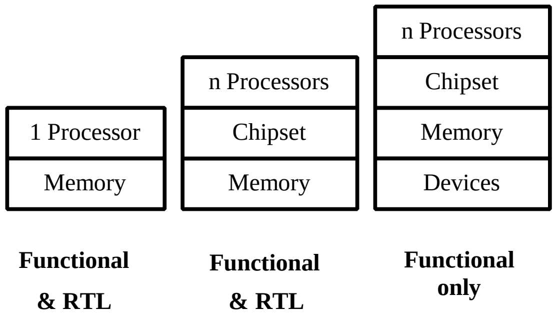

Figure 1: Available simulation environments

We used two functional simulators, Giza and SoftSDV [8], to test and debug the presilicon operating system kernels before running in RTL. Both simulators were utilized in our development process in order to debug code quickly. Giza was also designed to be used as a checker against the RTL model, so we always ran our code through it before running in RTL. Since the SoftSDV simulator is already described in “SoftSDV: A Presilicon Software Development Environment for the IA-64 Architecture” in this issue of the *Intel Technology Journal*, we only describe the Giza simulator.

Giza is built around an instruction accurate software simulator for Itanium processor’s ISA (Sphinx). It supports critical implementation specific registers, SAPIC, a non-blocking memory hierarchy (TLB+caches) that handles both synchronous and asynchronous traffic between the CPU and the external sub-system, and multiple CPU instances (multiprocessor). Implementation-specific registers are modeled to support firmware execution. SAPIC, non-blocking memory hierarchy, and multiprocessor (MP) are modeled to support characteristic subsystem traffic for typical IA-64 platforms. A functional accurate software model that mimics the Itanium processor’s front-side-bus (FSB) is designed to schedule CPU events and dispatch the resulting transactions to and from memory and I/O subsystems. Software models for the Itanium processor’s chipset and Itanium processor’s standard devices represent the latter.

By using functional simulators, we avoided wasting precious RTL cycles that could be used by conventional tests. We began with uniprocessor (UP) versions of the functional simulators and OS kernels. Once we passed the UP functional simulator test, the code would run on the RTL. These jobs often took over a million cycles to complete so the ramifications of simple code mistakes were great and had to be eliminated before being run on the RTL models. Once the kernels passed a UP functional and RTL simulator run, they were moved onto the multiprocessor path. Each symmetric multiprocessing (SMP) version of the kernel was debugged via a functional simulator. The MP RTL environment, known as COSIM, allowed modeling of multiple IA-64 RTL processor models, chipset models, PCI busses, and external interrupt controllers. This environment allowed us to exercise SMP kernels on many of the platform components before silicon was available, which taught us valuable lessons and uncovered errata that were not uncovered during conventional methods of testing.

Since operating system code is not “self checking,” the Munster and IPD Linux kernels were run in RTL with an RTL checker running at the same time. The RTL checker is a functional simulator that runs in conjunction with the RTL simulation and compares the architectural state after the retirement of each bundle. If a state mismatch occurs, then an error condition is flagged, and further analysis can be done to isolate the root of the problem.

Porting Challenges

There were many challenges in porting the kernels to run presilicon in RTL. We encountered a number of tool problems since we were on the leading edge as far as running code with the actual RTL model is concerned. The early tool sets often worked for running code in the functional simulator, but had problems with generating correct code for running in the RTL simulator. We had to write a utility called the AfterBurner to post-process compiler-generated assembly code and to fix problems that were preventing the code from running in RTL.

During the project, the compiler-generated code quality (correctness and performance) improved, as did the modeling of the architecture by the functional simulators. However, for some sequences of legal C code, the compiler produced semantically incorrect as well as architecturally incorrect code. In certain cases, due to the sequential nature of the functional simulator, architecturally incorrect code would appear to function correctly. In other cases, architecturally incorrect hand-written code would appear to execute correctly within the functional simulator (e.g., missing serialization instructions required by the architecture went undetected).

Benefits of Using Two Different Kernels

The benefit of porting multiple kernels to The Itanium processor was the ability to share some of the low-level start-up, trap handling (TLB faults, etc.), bcopy, port IO usage, and other code between the two kernels. It took us roughly two weeks to obtain a linkable Linux IA-64 kernel, and much of that time was spent on accommodating a non-GNU [11] C compiler. Much of the low-level code was already done from the Mach* port and just had to be merged into the Linux source tree. Another benefit of a second port was that it allowed us to redesign some of the code to make it cleaner and more efficient.

Porting two kernels allowed us to test some of the IA-64 Instruction Set Architecture in a slightly different way. We achieved a broader validation of some features such as Instruction Level Parallelism (ILP), speculation, predication, use of the large register files, the Register Stack Engine (RSE), and advanced branch architecture. Our “common trap handler,” the common path for saving and restoring state when entering/exiting the kernel, for the IPD Linux was very different from the Mach* version in both the design of the operating system and in the area of performance. As a result, the processor was exercised in an alternate way.

Both kernels supported Seamless mode, which is the ability to run IA-32 binaries on top of an IA-64 operating system kernel. We ran IA-32 user-mode programs on the kernels in presilicon RTL as another validation test.

ISSUES SPECIFIC TO PORTING MUNSTER

Mach3 [6] was the first kernel to be ported so the architecture-dependent code had to be written from scratch. We received some example code from other Intel groups, but some of it didn’t fit very well into the Mach3 architecture. One of the biggest issues that we encountered was Mach3’s ability to come into kernel mode on one stack and leave on another [15]. This added complexity to the trap handler due to the fact that all of the required IA-64 state had to be saved on to the Process Control Block (PCB) and restored into the new stack state. In Mach*, instead of a static assignment between threads and kernel stacks, the assignment is dynamic, and a thread that blocks in kernel context while waiting for some event can hand off its stack to the thread that is next in line. This is beneficial in terms of cache locality, but it complicates handling of the register stack engine because the kernel backing store is part of the kernel stack. As such, it does not persist between the time a thread enters the kernel and the time it returns. When entering the kernel from user mode, we first had to flush the dirty RSE registers into a dedicated area in the process control block before switching to the kernel backing store. Then, when we returned to user mode, we had to load the flushed RSE registers from the process control block into the physical register file before switching to the user backing store.

There were also LP-64 issues in the Mach3 source code where assumptions were made that ints, longs, and pointers were all the same size. This caused pointers to be truncated in some cases.

The Munster kernel port involved IA-64 start-up code, fault handling, TLB handling, context switching, system calls, interrupt handling, interprocess communication, LP 64-bit clean efforts, and user-mode libraries. We also had to port the Mach* build tools to UnixWare* before the Mach3 kernel could be built.

ISSUES SPECIFIC TO PORTING IPD-LINUX

Linux* was ported as another presilicon operating system validation kernel. (The source for the Trillian* kernel, which was demonstrated during the 1999 Intel Developer’s Forum was not available when this port began.) The main issue with porting Linux to IA-64 presilicon was the lack of a complete IA-64 GNU C compiler [13] at the time we began the port. We used the Intel Electron C compiler to compile the kernel. This required us to conditionally compile around the heavy usage of GNU C extensions [12] within the Linux kernel. Extensions such as inline C and inline assembly functions are not supported by the Electron compiler so this made the port more difficult than if we had a GNU C compiler available. The majority of our work in this port was in the two architecture-dependent directories that we added (include/asm-ia64 and arch/ia64). We also added an inline directory under arch/ia64 as a substitute for those routines that are normally inlined by the GNU C compiler. Since we didn’t have access to a GNU C compiler early in our development cycle, a basic user-mode shell, Josh, was written and used to launch Linux tests for validation purposes. Without a full GNU C compiler it proved very difficult to port the GNU C Library (GLIB C), which forms the basis for the full set of user-level shells and commands. Attempts were made to port GLIBC without the GNU C compiler, but they were unsuccessful in presilicon.

* Other brands and names are the property of their respective owners.

ISSUES SPECIFIC TO SUPPORTING PRESILICON PERFORMANCE ANALYSIS

The Linux port was also chosen as a vehicle to facilitate architectural performance research. The study of performance phenomena related to the microarchitecture, architecture, operating system, software, and tools required an open source workload. Besides offering a complete source, Linux offers SMP support, 64-bit clean code, runtime C library, and a kernel designed to work with multiple architectures. All of these features were integral to quickly satisfy the group's goals.

To automate data collection and analysis, the Linux kernel was augmented with many hooks to communicate with the simulation infrastructure. The hooks reported information about the internal state of the kernel, which were recorded within an event trace. The simulator and post processing tools correlate the information with architecture events, debug information, and compiler annotations. The kernel was also instrumented to support efficient branch and data trace collection when run on silicon.

To integrate the Linux kernel with the trace environment, and to support the silicon trace collection, a common feature set was added to the kernel. The kernel additions collect information about context switches, process creation and termination, and modifications to the address space (such as through mmap() or munmap()). This type of data collection has proven useful for other domains such as checkpoint/restore (the EPCKPT project), and kernel profiling (Intel Vtune™ performance analyzer [14]).

PRESILICON RESULTS

By running the operating systems presilicon, we found several unique RTL errata that would have affected commercial operating systems postsilicon. Errata were found in operating system-specific code, compiler-generated code sequences, speculative execution, platform interrupt paths, and tools. Other testing methods did not find these errata. Therefore, since this was a new architecture, it was worthwhile to incorporate an operating system kernel test presilicon to rule out major issues and to provide an indicator of the overall RTL model health.

Operating System-Specific Errata

The first errata we uncovered was related to the return from interrupt (rfi) instruction that is used by operating systems to return from an interruption/fault. The error occurred when the operating system start-up code used this instruction in the process of switching from physical mode to virtual mode, and the new instruction address was only valid in the new addressing mode. In checking the validity of the target address of the rfi instruction, the processor was using the previous (physical) addressing mode instead of the new (virtual) one, generating an exception. This prevented our kernels from booting and could have affected other operating system kernels like Trillian Linux and Windows NT*.

Errata Uncovered by Compiler-Generated Code Sequences

Presilicon OS runs uncovered an error in the Itanium processor's RTL where certain combinations of floating-point instructions produced an incorrect result because sequences of multiply-accumulate instructions with register dependencies were not properly stalled. The significance of this error is that this exact instruction sequence is used by the C compiler to implement the integer modulo operation. Since C is the primary language for software development, this error would have been encountered by most applications running on the processor.

Code Example:

```
int i, x, y ;

i=x%y ;
```

where the generated code would contain a sequence like the following (note the pseudo registers for the example):

```
fma fz=fa,fb,fc
;;
fma fw=fz,fk,fj
```

The result of the first fma is not available for several cycles, so the second fma instruction should have been stalled until fz was available.

Speculative Execution Errata

The Itanium processor does extensive branch prediction and speculatively executes instructions at the predicted branch target long before it is known if the branch will be taken. We found a problem with a conditional call to a subroutine, where the subroutine was short enough to execute a return instruction before it was known if the conditional call should have been executed. This caused the processor to permanently commit some of the state changes caused by the return instruction, even if the conditional call was incorrectly predicted and was not supposed to be executed.

* Other brands and names are the property of their respective owners.

Example:

// where p3 is false	**foo:**
(p3) br.call b0 = foo	alloc …
	mov … ;;
	br.ret b0

In this pseudo code example the predicate p3 was false, but the code at **foo** was speculatively executed, and its results were erroneously committed.

Platform Interrupt Errata

Several interrupt-related errata were uncovered by writing small tests that exercised the interrupt paths utilized by an operating system on a typical Itanium platform. This testing was accomplished in a simulation environment (COSIM) that combined multiple Itanium processor models, the chipset model, and platform component models such as the external interrupt controller model. This allowed exercising the path from a simulated device to the processor.

Unique errata were uncovered by generating interrupts in the modeled environment, causing the execution of specific interrupt flow paths. One of the interrupt errors was uncovered by redirecting interrupts to a particular processor based upon the priority of the processor in a multiprocessor simulation.

Tools Errata

During the development and testing of Munster and IPD Linux, many bugs were found in software development tools such as compilers and linkers, and also in various simulators used to run the kernels. Munster and IPD Linux were run in every simulator that was available to us and were crucial in detecting multiprocessor functional simulator errors.

POST-SILICON RESULTS

The great advantage of using a kernel for postsilicon debug that has been validated in the presilicon environment is that it removes potential software bugs from the list of unknowns during initial bring-up.

The extensive presilicon testing allowed the bring-up team to concentrate on mechanical, electrical, and silicon issues and provided them with a metric for assessing bring-up progress.

First Itanium processor's silicon was very healthy, so our kernels were able to run without modification as soon as initial platform issues were resolved and a stable operating range for the processor was found.

CONCLUSION

Testing an RTL model using an operating system kernel consumes many RTL cycles. The tests typically run for over one million cycles and take up cycles that could be used by shorter focus tests. This is the reason that our code was debugged on a functional simulator before launching the tests on the RTL model. We expected the kernels to boot if the model was healthy and if there were no infrastructure problems with the testing environment. The advantages of using an operating system far outweigh the disadvantages. We were able to find errata presilicon that normally would not have been detected until hardware was available. At that point, the problems can be difficult to isolate and expensive to fix. The kernels were also valuable during the first few days of Itanium processor's postsilicon bring-up. Our kernels were able to run without modification as soon as the silicon and platform hardware were stable. This allowed us to use our kernels as an indicator of hardware health during the first few days.

ACKNOWLEDGMENTS

These people were always there to support this unique project: Piyush Desai, Andy Pfiffer, Rahul Bhatt, Stan Smith, Rumi Zahir, Drew Hess, Tony Porterfield, Moenes Iskarous, Dennis Marer, Ken Arora, and Randy Anderson.

REFERENCES

[1] Carver, Fleckenstein, et al., "A System Centric Reference Model and an OS Microkernel for Pre-Silicon CPU Platform Validation," *Intel DTTC 1998*, (internal document).

[2] Alessandro Rubini, "Linux Device Drivers," O'Reilly & Associates, February 1998, ISBN: 1-56592-292-1.

[3] Linux Kernel Source Archives (best porting reference), ftp://ftp.kernel.org/pub

[4] M. Beck, H. Bohme, et al., Linux Kernel Internals (Second Edition), Addison-Wesley, 1998, ISBN: 0-201-33143.

[5] *IA-64 Application Developer's Architecture Guide*, Order Number: 245188-001, May 1999. http://developer.intel.com/design/IA64

[6] CMU CS Project Mach* Home Page, http://www.cs.cmu.edu/afs/cs.cmu.edu/project/mach/public/www/mach.html

[7] OSF Kernel and Server Programming Manuals, http://www.cs.cmu.edu/afs/cs/project/mach/public/www/doc/osf.html

[8] R. Uhlig, R. Fishtein, O. Gershon, H. Wang, "SoftSDV: A Presilicon Software Development Environment for the IA-64 Architecture," *Intel Technology Journal*, Q4 1999.

[9] Gollakota, Naga, "COSIM," *Intel DTTC* 1998.

[10] H.P. Messmer, *The Indispensable PC Hardware Book*, Addison-Wesley, 1997.

[11] GNU's Not Unix !, http://www.gnu.ai.mit.edu

[12] "Using and Porting the GNU Compiler Collection (GCC), Extensions to the C Language Family",

http://www.gnu.org/software/gcc/onlinedocs/gcc_4.html#SEC62

[13] "GCC Home Page",

http://www.gnu.org/software/gcc/gcc.html

[14] "Vtune™ Performance Analyzer," http://developer.intel.com/vtune/analyzer/index.htm

[15] Draves, R.P., Bershad, B., Rashid, R. F., and Dean, R.W., "Using Continuations to Implement Thread Management and Communication in Operating Systems," in *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, 1991, ftp://ftp.cs.cmu.edu//afs/cs/project/mach/public/doc/unpublished/internals_slides/Continuations/sosptr.ps

[16] Wheeler, Bob, "Porting and Modifying the Mach* 3.0 Microkernel," *Third USENIX MACH* Symposium*, 1993, ftp://ftp.cs.cmu.edu/afs/cs/project/mach/public/doc/published/porting.tutorial.slides.ps

[17] Rusling, David, "The Linux Kernel," http://www.redhat.com/mirrors/LDP/LDP/tlk/tlk.html

AUTHORS' BIOGRAPHIES

Kathy Carver received a B.S. degree in computer science and mathematics from Western Kentucky University in 1981. She joined Intel Corporation in 1992 and worked on compiler validation and integration for Intel's IPSC860, Paragon, and TFLOPS systems. She is currently part of the Itanium processor Architecture Validation group in IPD. Her e-mail is kat@co.intel.com

Chuck Fleckenstein is currently working for the IPD Itanium processor validation group. He received an M.S. degree in computer science from Wright State University in 1988. His interests include distributed programming languages and operating systems. His e-mail is cfleck@co.intel.com.

Joshua LeVasseur works for the IPD IA-64 Architecture group. He currently focuses on system architecture performance, IA-64 optimizations, and simulator/analysis infrastructure. His e-mail is joshua.levasseur@intel.com

Stephan Zeisset received an M.S. degree in computer science from the Munich University of Technology in 1994. Since then he has been working on the operating system of the TFLOPS system and its predecessors, with a specialization in distributed memory management. Stephan is currently working on Itanium processor validation for IPD. His e-mail is sz@co.intel.com