SoftSDV: A Presilicon Software Development Environment for the IA-64 Architecture

Richard Uhlig, Microprocessor Research Labs, Oregon, Intel Corp.
Roman Fishtein, MicroComputer Products Lab, Haifa, Intel Corp.
Oren Gershon, MicroComputer Products Lab, Haifa, Intel Corp.
Israel Hirsh, MicroComputer Products Lab, Haifa, Intel Corp.
Hong Wang, Microprocessor Research Labs, Santa Clara, Intel Corp.

Index words: presilicon software development, dynamic binary translation, dynamic resource analysis, processor performance simulation, IO-device simulation

Abstract

New instruction-set architectures (ISAs) live or die depending on how quickly they develop a large software base. This paper describes SoftSDV, a presilicon software-development environment that has enabled at least eight commercial operating systems and numerous large applications to be ported and tuned to IA-64, well in advance of Itanium™ processor's first silicon. IA-64 versions of Microsoft Windows* 2000 and Trillian Linux* that were developed on SoftSDV booted within ten days of the availability of the Itanium processor.

SoftSDV incorporates several simulation innovations, including *dynamic binary translation* for fast IA-64 ISA emulation, *dynamic resource analysis* for rapid software performance tuning, and *IO-device proxying* to link a simulation to actual hardware IO devices for operating system (OS) and device-driver development. We describe how SoftSDV integrates these technologies into a complete system that supports the diverse requirements of software developers ranging from OS, firmware, and application vendors to compiler writers. We detail SoftSDV's methods and comment on its speed, accuracy, and completeness. We also describe aspects of the SoftSDV design that enhance its flexibility and maintainability as a large body of software.

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

INTRODUCTION

The traditional approach to fostering software development for a new ISA such as IA-64 is to supply programmers with a hardware platform that implements the new ISA. Such a platform is commonly known as a software-development vehicle (SDV) and suffers from a key dependency: it cannot be assembled until first silicon of the processor has been manufactured. This paper describes how Intel eliminated this dependency for IA-64 by building an SDV entirely in software through the simulation of all processor and IO-device resources present in an actual hardware IA-64 SDV. This simulation environment, which we call SoftSDV, has enabled substantial development of IA-64 software, well in advance of an Itanium processor's first silicon.

A principal design goal for SoftSDV is that it support development all along the software stack, from firmware and device drivers to operating systems and applications (see Figure 1). The performance of each of these layers of software is dependent upon optimizing compilers, which themselves must be carefully tuned to IA-64 [1, 2]. Each of these types of software development has a different set of requirements with respect to simulation *speed*, *accuracy*, and *completeness*.

Application developers, for example, are primarily interested in simulation speed, whereas optimizing-compiler writers value accuracy in processor-resource modeling so that they can evaluate the effectiveness of their code-generation algorithms. OS, firmware, and device-driver developers, on the other hand, require completeness in the modeling of platform IO devices and

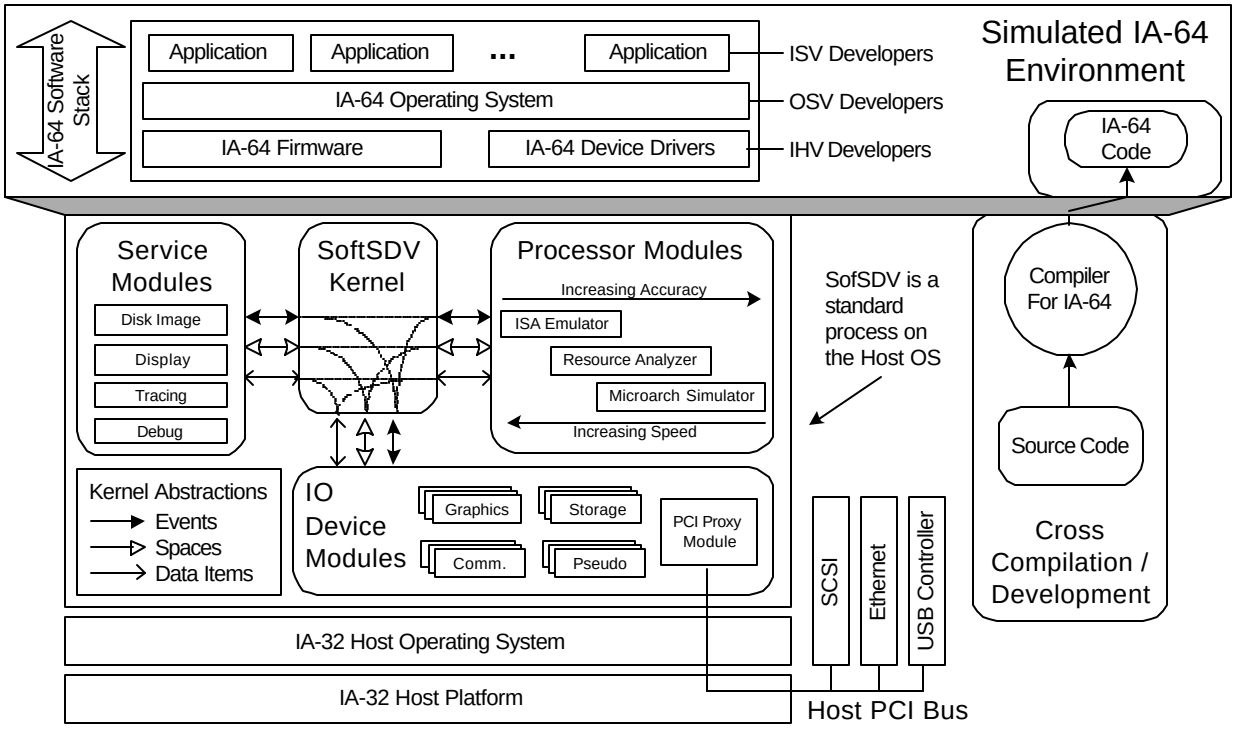

Figure 1: The SoftSDV software architecture

system-level processor functions (e.g., virtual-memory management, interrupt processing, etc.).

Not only do requirements vary based on the type of code, but their relative importance shifts depending on the stage of software development. Early efforts to port and debug code to IA-64 are best aided by very fast ISA emulation, whereas successive tuning of code for performance requires ever-increasing levels of simulation accuracy. Similarly, in the early stages of porting an OS to IA-64, it is sufficient to model a basic set of boot IO devices. However, once that OS is up and running, the meaning of simulation "completeness" expands to include arbitrary new IO devices as OS developers seek to port as many device drivers to IA-64 as possible.

Often overlooked, but equally important features of a simulation infrastructure are its *flexibility* and *maintainability*. SoftSDV has been under development for nearly as long as IA-64 and has had to track and rapidly adapt to improvements in the ISA definition; flexibility and maintainability of the simulation infrastructure were absolutely essential. But here too, the requirements change over time. Early in the development of a new ISA, design changes are frequent, and a simulation environment must adapt quickly. Later, as the hardware definition becomes more concrete, flexibility gradually becomes less important, and it can be traded for increased simulation speed or accuracy.

These diverse and shifting requirements underscore a fundamental truth of simulation: a single technique or tool cannot meet the needs of all possible types of software development at all times. SoftSDV acknowledges this fact through an extensible design that accommodates the best features of multiple innovative simulation technologies in a single, common infrastructure. The resulting system enables IA-64 software developers to select the combination of simulation speed, accuracy, and completeness that is most appropriate to their particular needs at a given time, while at the same time preserving the flexibility and maintainability of the overall simulation infrastructure.

In the next section, we review related work and then briefly overview the hardware components typically found in an IA-64 SDV. We then present the software architecture of SoftSDV and describe each of its component processor modules and IO-device modules in detail. We conclude with a discussion of results and a summary of our experiences and lessons learnt.

RELATED WORK

A survey of recent simulation research reveals a continuous tension between the conflicting goals of simulation speed, accuracy, completeness, and flexibility.

A good example of a technique that achieves very high simulation speeds at the expense of accuracy is *dynamic binary translation*. This method for fast ISA emulation works by translating instructions from a *target ISA* into equivalent sequences of *host ISA* instructions. By caching and reusing the translated code, an emulator can dramatically reduce the fetch-and-decode overhead of traditional instruction interpreters. Cmelik and Keppel describe an early implementation of this method in their Shade* simulator, and they provide an excellent survey of several related techniques [3]. The Embra simulator has shown that dynamic binary translation can be extended to support the simulated execution of a full OS by emulating privileged instructions, virtual-to-physical address translation, and exception and interrupt processing [4].

Fast ISA emulators such as Shade and Embra are unable to predict processor performance at a detailed clock-cycle level. When simulation accuracy and flexibility are of primary importance, a better approach is often that of a simulation tool set, such as SimpleScalar, which enables rapid construction and detailed simulation of a range of complex microarchitectures [5]. SimpleScalar has indeed exhibited a high degree of flexibility as evidenced by its extensive use in the research community for a variety of microarchitecture studies. But this flexibility and accuracy comes at a cost: detailed SimpleScalar simulations can be more than 1,000 times slower than native hardware, whereas fast ISA emulators like Shade and Embra exhibit slowdowns ranging from 10 to 40, depending on the type of workload that they simulate. These numbers illustrate the compromises that simulators must make between speed, accuracy and flexibility, with Shade and Embra representing one end of the spectrum and SimpleScalar situated near the other end.

Many other intermediate points along the speed-accuracy-flexibility spectrum are possible. FastCache, for example, extends dynamic binary translation techniques to simulate the performance of simple data-cache structures with slowdowns in the range of 2-7, but it is limited with respect to other forms of microarchitecture simulation [6]. At the expense of some flexibility, FastSim uses memoization (result caching) to model a full out-of-order processor microarchitecture with slowdowns ranging from 190 to 360, a speedup of roughly 8-15 relative to comparable SimpleScalar simulations [7]. Another way to trade speed for accuracy is *sampling*: running only certain portions of a target workload through a detailed performance simulator. If the samples are chosen carefully and are of sufficient length, they can predict the performance of the entire workload with far less simulation time, but at the expense of some increased error [8].

SimOS [9] and SimICS* [10] are both good examples of simulation systems that have attained a high level of completeness. Both extend their simulations beyond the processor to include IO-device models, and they are able to support the simulated execution of complete operating systems as a result.

SoftSDV uses many techniques similar to those described above. However, because of the unique capabilities of IA-64 and the diversity of software development that SoftSDV must support, we found we had to reexamine many of the techniques in a new context. To explain some of the issues, we briefly overview the components in a typical IA-64 SDV in the next section.

COMPONENTS OF AN IA-64 SDV

At the core of an actual hardware IA-64 SDV platform is one or more Itanium processors that implement the IA-64 ISA. For the purposes of this paper, the most relevant aspects of IA-64 are the following:[1]

- *Memory Management*: An IA-64 processor translates 64-bit virtual memory accesses through split instruction and data TLBs, which are refilled by software miss handlers with a hardware assist. The TLB enforces page-level read, write, and execute permissions for up to four privilege levels.
- *Predication*: Most IA-64 instructions can be executed conditionally, depending on the value of an optional predicate register associated with the instruction.
- *Data Speculation*: IA-64 supports an advanced-load operation, which enables a compiler to move loads ahead of other memory operations even when the compiler is unable to disambiguate neighboring memory references. The address from an advanced load is inserted into an Advanced Load Address Table (ALAT), which must be checked for data dependencies on subsequent memory operations. A load-check operation examines the ALAT and invokes fix-up code whenever necessary.

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

[1] More details regarding IA-64 are available from the Intel Developer's Web Site [2].

Module		Characteristics	Typical Usage
Processor Modules	Fast ISA Emulator	Highest speed, no cycle-accurate perf. data	Rapid IA-64 app. and OS development
	Resource Analyzer	Medium speed/accuracy, limited flexibility	Large application, OS and compiler tuning
	Microarch Simulator	Highest accuracy and flexibility	Advanced compiler and microarch co-design
IO-device Modules	Basic IO Devices	Provide platform-modeling completeness	Early OS porting with standard boot devices
	IO-Proxy Module	Links to arbitrary PCI and USB devices	Advanced device-driver development

- *Control Speculation*: An IA-64 compiler can move loads before branches that guard their execution. If a speculative load causes an exception, the exception is deferred, and a Not-a-Thing (NaT) bit associated with the destination register is set instead. NaT bits are propagated as a side effect of any uses of the speculative load value, and they are checked after the controlling branch finally executes.
- *Large Register Set:* IA-64 includes 128 general registers and 128 floating-point registers, along with numerous other special-purpose registers. IA-64 registers assist loop-pipelining scheduling through their support of *rotating registers*.

Predication, speculation, and abundant registers are all part of a key principle behind the design of IA-64: to enable a compiler to explicitly expose instruction-level parallelism (ILP) to the hardware microarchitecture [1, 2]. By helping to relieve the hardware of finding ILP, IA-64 makes possible higher-frequency processor implementations. The compiler expresses instruction parallelism to the hardware by collecting instructions into *instruction groups*, which are independent instructions that can be executed at the same time. As we will see, the above IA-64 features create new challenges and opportunities for processor-simulation techniques.

In addition to an Itanium processor, an IA-64 SDV typically contains a chipset (e.g., 460GX), with support for PCI and USB IO-device busses, and a basic collection of IO devices suitable for running an operating system. This includes an IO streamlined APIC interrupt controller and an assortment of keyboard, mouse, storage, network, and graphics devices. Since a hardware SDV typically contains a number of expansion PCI slots, and a USB host controller, the range of devices it supports is limited only by the availability of devices designed to these bus standards. Our goal with SoftSDV was to provide the same level of IO-device support.

SOFTWARE ARCHITECTURE

The software architecture of SoftSDV is based on a simulation kernel that is extended through the addition of *modules* (see Figure 1 and Table 1). A SoftSDV module either models a hardware platform component (such as a processor or IO device), or it implements a simulation service (such as a trace-analysis tool). We call these two types of modules *component modules* and *service modules*, respectively.

SoftSDV modules share data and communicate with one another through a set of abstractions provided by the kernel: *spaces*, *events*, *data items* and *time*.

- *SoftSDV spaces* are used to model how platform components communicate through physical-address space, IO-port space, PCI-configuration space, and other linearly addressable entities such as disk images and graphics framebuffers. Modules can create spaces and then register *access-handler* functions with the kernel for a certain address region in a space. When another module reads or writes an address in that range, the SoftSDV kernel routes the access to the registered handler. SoftSDV spaces enable an IO-device module, for example, to specify how its control registers behave by mapping them to specific addresses in IO-port space.
- *SoftSDV events* enable a module to request notification of some occurrence inside another module. Events can be used to model IO-device interrupts, or to collect event traces, such as a sequence of OS context switches, which might be analyzed by a trace-processing module.
- *SoftSDV data items* provide a mechanism for modules to name and share their state with other modules. Named data items enable a processor module, for example, to make its simulated register values available to a debugging tool. Some data items are managed by the SoftSDV kernel itself and they are sometimes used to specify configuration data that modules can query

to determine how they should function during a simulation.

- *SoftSDV time* enables modules to synchronize their interactions to a common clock and to schedule the execution of future events. A disk-controller model, for example, can register a callback function with the kernel to be called after some pre-computed delay that represents the time for a simulated seek latency.

SoftSDV has a standard set of tracing and debugging modules that are built upon the abstractions above. SoftSDV events enable the construction of tracing modules that specify, by event name, items to be traced. Traceable events include instructions, addresses, IO-device accesses, OS events, etc. SoftSDV traces are suitable for trace-driven cache and processor performance simulators, and they can also provide input to the Vtune™ performance analyzer [11]. The SoftSDV debugger uses SoftSDV events and data items for setting instruction and data breakpoints, reading memory values, examining register values for a specific simulated processor, etc.

SoftSDV also includes a standard set of component modules, including three IA-64 processor models, each offering different levels of simulation speed, accuracy, and flexibility. For completeness in platform modeling, SoftSDV also includes an assortment of IO-device models and a special IO-proxying module that links the simulation to actual PCI and USB devices installed in the simulation host machine. Table 1 summarizes these standard component modules and serves as an outline for the remainder of this paper, in which we detail the techniques used by each of these modules.

FAST IA-64 EMULATION

SoftSDV's fastest processor module is an emulator that implements the full IA-64 instruction set, including all privileged operations, address translation, and interrupt-handling functions required to support operating systems. The principal goal of the emulator is speed; it is unable to report cycle-accurate performance figures.

SoftSDV's emulator uses dynamic binary translation to convert IA-64 instructions into sequences of host (IA-32) instructions called *capsules*, which consist of a *prolog* and a *body* (see Figure 2). The capsule body emulates the IA-64 instruction in terms of IA-32 instructions executed on the host machine. In the example shown in Figure 2, a 64-bit *Add* instruction is emulated by a sequence of IA-32 instructions that loads the required source registers from a simulated IA-64 register file (held in host memory), performs the simulated operation (a 64-bit *Add* using the 32-bit *Add* operations of IA-32), and then stores the result back to the simulated register file.

The capsule prolog serves two purposes. First, it implements behavior that is common to all IA-64 instructions, such as predication and control speculation. A portion of the prolog code checks each instruction's qualifying predicate, and it jumps over the capsule to the next instruction if the predicate is false. Similarly, another portion of the prolog examines the NaT bits of all source operands, and it propagates any set values to destination registers, or it generates a fault as dictated by the semantics of the instruction. A second use for the prolog is to implement a variety of simulation services, such as execution tracing, instruction counting, and execution breakpoints, which we discuss later.

The emulator translates instructions only as needed in units of basic blocks, which it caches in a translation database. Since capsules require on average 25 IA-32 instructions for each IA-64 instruction that they emulate, a large simulated workload can quickly consume host memory, causing the host OS to begin paging to disk. The emulator limits the maximum size of its translation cache to prevent host-OS paging, choosing instead to retranslate IA-64 instructions, a far faster operation.

SoftSDV capsules eliminate the need for a full fetch-decode-execute loop. Instruction emulation begins with an indirect branch that goes directly to a capsule corresponding to the current simulated instruction pointer (IP). Since capsules are linked directly from one to another, execution proceeds from capsule to capsule until an untranslated instruction is encountered, and control

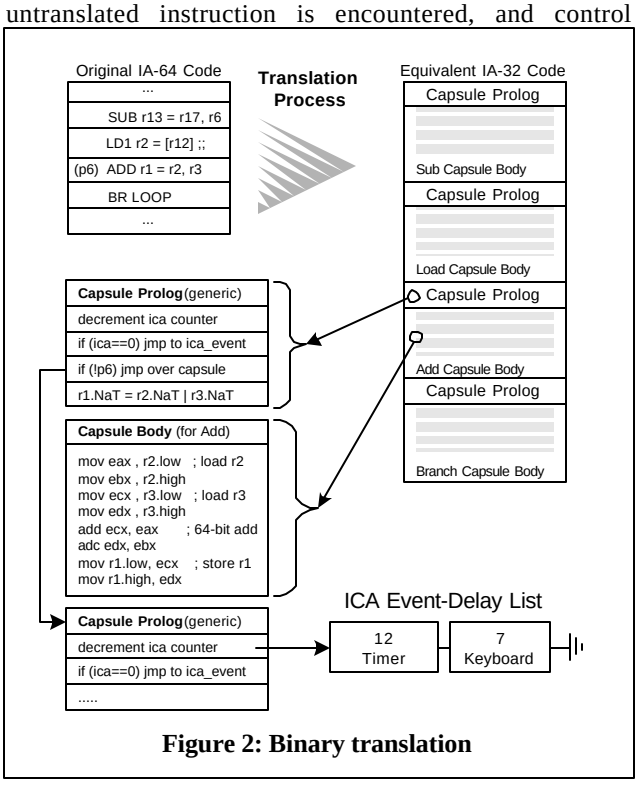

Figure 2: Binary translation

transfers back to the translator.

SoftSDV executes nearly all IA-64 instructions as capsules, but certain complex instructions are emulated with calls to C* functions. Throughout the development of SoftSDV, we considered moving to capsule-based implementations for various complex operations, but we frequently found that the resultant reduction in flexibility and maintainability of the emulator did not justify the optimization. We also considered more aggressive intercapsule optimizations, such as those used by Shade* and Embra (e.g., avoiding loads/stores to the simulated registers in host memory when neighboring instructions use the same register). Unfortunately, with the very large IA-64 register set, and the limited number of IA-32 host registers, we found such optimizations ineffective. Intercapsule optimizations suffer the further side effect of limiting the granularity at which debug breakpoints can conveniently be set.

Address Translation

SoftSDV models simulated physical memory by allocating pages of host virtual memory to hold the contents of simulated data. For any simulated memory reference, the emulator must first translate a simulated virtual address to a simulated physical address (V2P), and then further translate this physical address to its allocated host address (P2H). The composition of these two translation functions provides a full translation from virtual to host memory (V2H = P2H(V2P(Virtual))).

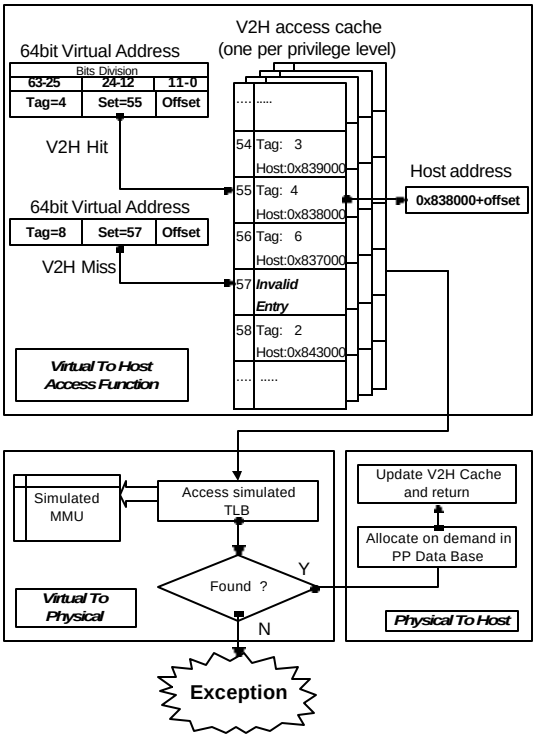

Figure 3: Emulating IA-64 address translation

The emulator uses a data structure called a V2H cache to accelerate address translation (see Figure 3). When an IA-64 load/store operation or branch/call instruction references a virtual address, the emulator first searches a V2H cache using highly efficient assembly code called from the capsule. In the common case, the translation is found, and the memory reference is quickly satisfied. In the case of a V2H miss, a full translation sequence (V2P and P2H) is activated by simulating an access to the TLB and an OS-managed page-table structure. If the translation succeeds, the V2H table is updated, and execution returns to the capsule. If the search fails to find a valid translation, then a simulated page fault or protection violation has occurred, and SoftSDV raises an exception for the simulated OS to handle.

The emulator implements V2H tables as variable-sized caches of valid address mappings. This is in contrast to Embra's *MMU relocation array*, which is a fixed-size 4-MB table that maps every possible page in a 32-bit virtual address space [4]. While the Embra approach guarantees a 100% hit rate for any valid mapping, we found this approach unusable for modeling a 64-bit virtual address space. Since V2H refills are a principal source of slowdowns, the emulator uses a number of optimization techniques to accelerate refills, and it supports configurable V2H table sizes to tune hit rates to the requirements of a given workload.

Page Protection

The emulator implements page protections by building a V2H cache for each type of memory access: *V2H-read*, *V2H-write*, and *V2H-execute*. A read-only page, for example, present in the V2H-read cache does not contain an entry in the V2H-write cache. By dividing the V2H caches in this way, the emulator avoids performing protection checks explicitly in each capsule; it merely generates code to access the V2H cache appropriate to the desired operation (e.g., load, store, branch), and a V2H miss enforces the protection. The emulator builds a set of read-write-execute V2H caches for each privilege level (i.e., 12 V2H caches in all) and simply switches between them when privilege levels change. The use of multiple V2H caches is a memory-speed tradeoff. At the expense of extra memory devoted to the V2H caches, the emulator simplifies the protection-checking code in each capsule, thus accelerating overall instruction-emulation times.

Speculative Data Accesses

Data speculation with advanced loads requires special treatment by the emulator. The capsule for an advanced

load includes code that inserts the load address into a simulated ALAT. The capsules for any subsequent store instructions include highly optimized code (11 host instructions) that searches the ALAT for an address match. Any entries with matching addresses are removed from the ALAT so that a subsequent load-check operation will invoke the appropriate fix-up code.

Instruction Fetching

Rather than point directly to translated code capsules, V2H-execute entries reference a page of pointers to code capsules. This level of indirection solves two problems. First, since the emulator lazily translates instructions a basic block at a time, many portions of a code page may be untranslated; these cases are handled by pointing to a *do-translate* function, rather than a capsule entry point. Second, since capsules are of varying size, their entry points are not simple to calculate given only a virtual address for an instruction; the pointers-to-capsules page greatly simplifies simulation of branch instructions and arbitrary jumps to code, such as from a debugger.

The emulator avoids V2H-execute lookups by directly chaining a branch-instruction capsule to its target-instruction capsule, provided they reside on the same page. For cross-page branches, the chain passes through the V2H-execute cache to ensure that the code page is accessible.

The emulator supports self-modifying code by preventing a page from residing in both the V2H-write and V2H-execute caches at the same time. Any attempt to write to a code page causes a V2H-write miss, which causes the page to be removed from the V2H-execute cache. Any subsequent attempt to execute instructions from the page causes a V2H-execute miss, which results in a lazy retranslation of the modified code page.

Interrupt Processing

The emulator can potentially run for long periods of time without ever leaving its capsules. This presents a problem because other SoftSDV modules (such as simulated IO devices) might need to interrupt processor execution. The emulator provides a solution to this through a mechanism called an *instruction-count action* (ICA). ICA is based on a sorted list of event-delay pairs that define a set of callback functions to be executed after some number of instructions have executed. The delay of the event at the head of the ICA list is decremented in the prolog of each capsule (see Figure 2). When the delay reaches 0, the event is triggered by calling its associated callback function. After completion of the callback function, the event-delay pair is deleted from the ICA list.

The ICA mechanism is integrated with the SoftSDV time abstraction, providing a coarse-grained notion of time based on number of instructions executed. It is also ideal for implementing certain IA-64 debugging facilities, such as single-step instruction execution, which would otherwise require retranslation of instruction capsules.

Multiprocessor Simulation

The emulator can simulate a platform with up to 32 processors in a symmetric shared-memory configuration. Only one processor is simulated at a time, with switches between simulated processors scheduled in a simple round-robin order with a configurable time slice. All processors share memory, translations, and all simulated platform devices. Capsules indirectly access simulated processor state and processor-specific emulator state (e.g., V2H caches, ICA lists, etc.) through a pointer, which enables rapid switching between simulated processors via a simple pointer change.

DYNAMIC RESOURCE ANALYSIS

The emulator described in the previous section is ideal for rapid porting of operating systems and applications to IA-64. Once the porting is complete, a software developer may be interested in tuning code for performance, which requires trading some simulation speed for increased accuracy. SoftSDV's second processor module offers just this: it aims to achieve a level of performance-prediction accuracy that is within 10% of a detailed Itanium microarchitecture simulator (which we describe in the next section), while retaining the highest possible simulation speed.

SoftSDV applies three principles to achieve this goal. First, it tightly integrates performance analysis with its fast IA-64 emulator, enabling it to overcome the bottlenecks of traditional trace-driven performance simulation. Second, it assumes an in-order processor pipeline and selectively models only those processor resources that have the greatest effect on overall performance (e.g., first-level caches, branch prediction, functional units, etc.). Third, it caches the results of its resource analysis to speed future performance simulation. We call this collection of methods *dynamic resource analysis*, and we refer to this SoftSDV module as the *resource analyzer*.

Processor and Memory Resources

Processor performance analysis ultimately boils down to accounting for the resource requirements of executing code. Take, for example, the code shown in Figure 4, which shows a *subtract* and a *load* in one instruction group and an *add* and a *branch* in a second instruction

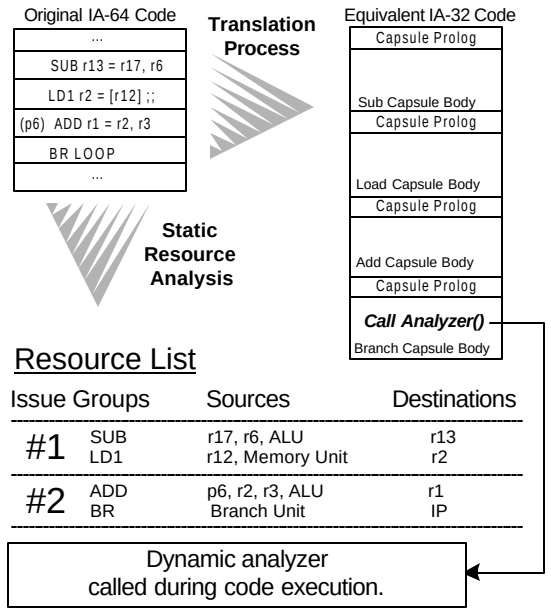

Figure 4: Resource analysis for two issue groups

group (the notation ";;" separates instruction groups). The *load* operation:

LD1 r2 = [r12] ;;

requires the resources of a source-address register (*r12*), a destination register (*r2*), and a data-cache line and read port. Similarly, the *add* instruction:

(p6) ADD r1 = r2, r3

requires the resources of an ALU functional unit, a predicate register (*p6*), two source registers (*r2* and *r3*), and a destination register (*r5*).

A processor examines instruction groups to find collections of instructions (called an *issue group*) that can be dispatched in parallel subject to the resource constraints of its execution pipeline. When all the required resources of all instructions in an issue group are available, the instructions can execute immediately; otherwise, they stall. In the example in Figure 4, the *load* feeds the *add* through register *r2*. If the *load* stalls because of a data cache miss, then the *add* will stall as well because it requires the *r2* resource. The resource analyzer identifies such resource dependencies by dividing its operation between two phases, each of which is tightly integrated with the IA-64 emulator. A *static resource-analysis* phase is invoked by the emulator's translator, and a *dynamic-analysis phase* is invoked by translated code capsules.

Static Resource Analysis

Like the IA-64 emulator, the resource analyzer examines code lazily, as it is first encountered. Whenever the IA-64 emulator completes translation of a new basic block of code, it calls the resource analyzer, which examines the basic block to find issue groups and then statically determines the microarchitectural resources required by each issue group. The analyzer caches this *resource list* to avoid the cost of repeating the analysis each time the issue group executes (see Figure 4). The resource list includes both *sources*, which are required for the issue group to execute, and *destinations*, which are resources that will be available after the issue group executes.

Dynamic Analysis Phase

The dynamic phase of the analyzer is invoked by emulator capsules after execution of each basic block. The analyzer examines instruction resource requirements by modeling a simplified Itanium microarchitecture consisting of a front end, a back end, and caches.

The analyzer's back end keeps track of the availability of core pipeline resources (e.g., register files, functional units, etc.) that could stall the execution of an issue group. Each resource that is required for the execution of the current issue group is checked for its availability in the current cycle. If it is not available, then the issue group stalls execution, and the cycle counter is advanced to the time when all required resources will be available and the issue group can enter the execution stage. Next, the resource state is updated to reflect the results of the current issue group. For each destination of the instructions in the issue group (excluding loads), the clock cycle in which these resources will be available for use by subsequent instructions is calculated by adding each instruction's latency to the cycle in which the issue group was dispatched.

The analyzer's front end models branch prediction and instruction-fetching mechanisms in accordance with the Itanium microarchitecture. Since all instructions traverse the front end several cycles before they reach the back end, the resource analyzer maintains two separate cycle counts, one for the back-end execution (as described previously) and one for the front end. The two cycle counters are synchronized through a model of a decoupling buffer, which specifies the cycle in which an issue group can be dispatched. If the issue group is not ready, the decoupling buffer stalls the back end. Conversely, if the decoupling buffer is full, it causes front-end stalls.

The analyzer models instruction and data caches to account for performance lost due to cache misses. These cache models are referenced by both the front end (for instruction fetches) and the back end (for data references).

The IA-64 emulator maintains its own copy of the first-level data cache and achieves speed through integration with its V2H translation tables in an approach similar to that of Lebeck's FastCache [6]. This copy contains a subset of the data-cache lines simulated by the resource analyzer. When the emulator detects a hit, it continues execution without calling the analyzer. In the rare case when the emulator suspects a cache miss, it calls the analyzer for verification. When a miss occurs in the analyzer cache, the analyzer updates its own cache contents and informs the emulator about the cache line it replaced.

The end results of the analysis described above are a set of performance metrics (e.g., branch-prediction accuracy, cache misses, stall cycles, etc.), which are mapped onto SoftSDV data items so that they can be read by a performance visualization tool, such as the Vtune™ performance analyzer.

DETAILED PROCESSOR MICROARCHITECTURE SIMULATION

The dynamic resource analyzer described previously is ideal for rapid tuning of compilers, operating systems, and large applications, where approximate performance of a fixed microarchitecture is acceptable. For other applications, such as compiler tuning for a new microarchitecture under development, a further shift along the speed-accuracy-flexibility spectrum is needed. To deal with such situations, SoftSDV works together with a simulation toolset for exploration of IA-64 microarchitectural designs. SoftSDV's third processor module, a detailed model of the Itanium microarchitecture, is written using this toolset, as are other future IA-64 microarchitectures currently under development by Intel.

IA-64 Microarchitecture Simulation Toolset

The toolset consists of two main components: an event-driven simulation engine, and a set of facilities for functional evaluation of IA-64 instruction semantics.

The event-driven simulation engine provides a flexible set of primitives for modeling microarchitectural resources, arbitration, and accounting of processor cycles. A processor pipeline is modeled as a set of communicating finite state machines through which *tokens* representing instructions or data are evaluated on a cycle-by-cycle basis. Whenever a token cannot acquire a certain resource (e.g., a latch or port), a stall condition is encountered and its cycle penalty is accounted for. The total execution time of a workload is derived from the accumulation of cycles spent by all tokens that traverse the pipeline.

The functional evaluation of IA-64 instruction semantics is provided by a set of four interfaces called by different stages of the event-driven microarchitecture model:

- *Fetch* provides a decoded instruction, given an IP.
- *Execute* computes the results of an instruction in the context of some microarchitectural state.
- *Retire* commits the microarchitectural state to the permanent architectural state.
- *Rewind* rolls back the unretired microarchitectural state to previously defined values.

By dividing the interfaces in this way, a processor model is able to express complex microarchitectural interactions involving speculative instruction execution.

Speed-Accuracy Modes and Sampling

Like the resource analyzer, the Itanium microarchitecture model is tightly integrated with the IA-64 emulator through an interface that enables the sharing of the architectural state (memory and processor registers). This interface makes it possible to dynamically change simulation speed and accuracy as a workload runs. It is possible, for example, to rapidly advance through the simulated boot of an OS with the fast IA-64 emulator. Then, prior to the execution of some workload of interest, the state of the processor's caches is initialized using memory traces produced by the emulator. When detailed simulation is desired, the microarchitectural simulator reads the current machine state from the emulator, and begins its simulation. After running in detailed mode for some time, execution can return to the fast mode to advance deeper into a workload's execution.

SoftSDV supports both *uniform sampling* at some regular, predefined period, and *event-based sampling*. For event-based sampling, special *markers* are compiled around regions of interest in a workload. SoftSDV dynamically recognizes such markers as a workload executes, and generates corresponding SoftSDV events, which are monitored by the processor modules to determine when they should switch between speed-accuracy modes.

IO-DEVICE MODULES

SoftSDV provides a standard collection of IO-device models suitable for supporting the simulated execution of a complete IA-64 operating system, including its basic device drivers. These include selected 460GX chipset and boardset functions (such as interrupt controllers, periodic timers, serial port, mouse, keyboard, etc.) as well as models for assorted storage and graphics controllers (such as IDE/ATA, ATAPI CD-ROM, VGA, etc.).

A SoftSDV device model typically consists of two halves: a front half that implements the device interface as expected by a simulated device driver, and a back half that simulates the device function. The implementation of many device models is a straightforward conversion between device commands and host-OS services. Models for the keyboard and mouse, for example, field host-OS window events (keypress up/down, pointer movement, etc.) and convert them into equivalent device events (a keyboard interrupt and scancode). Similarly, the serial-port model connects to the actual serial port of the host machine through host-OS calls, and it ferries data back and forth between the simulated and actual serial ports.

IO-Device Services

Since many devices offer a similar function (e.g., storage), but with differing interfaces (e.g., IDE/ATA, ATAPI, SCSI), we identified points in common among device models and structured them as reusable *IO-device services*, encapsulated in their own SoftSDV modules. A *disk-image service*, for example, provides functionality useful for any storage model. It holds the contents of a simulated disk in a file or a raw disk partition in the host machine, and has the ability to log all changes to the simulated disk's contents. An off-line utility can then either commit or discard those changes at the end of a simulation. The disk-image service maps itself onto a SoftSDV space, where its functionality can be accessed by any disk-interface model.

Similarly, a *display service* maps a generic flat framebuffer surface onto a SoftSDV space, and it reflects accesses to this surface in a host-OS window, or directly to a second dedicated display attached to the simulation host machine. This structuring relieves graphics-adapter models from the details of how a framebuffer is displayed so that they can focus instead on simulated processing of graphics commands. It also enhances SoftSDV maintainability, since it decouples graphics models from the host-OS windowing system; when porting to a new host OS, only the display service and not each graphics model must be rewritten.

Pseudo Devices

Some SoftSDV modules model a fictitious device, or only some aspect of an actual device. We have, for example, built a pseudo-device module that maps the resource requirements of multiple devices into PCI configuration space. Although not backed by actual PCI-device models, these headers present to a simulated OS the illusion of a platform with multiple PCI devices, and thus enable rapid testing of the OS's device configuration and plug-and-play algorithms. Such a test environment is, in fact, far more convenient than an actual hardware platform since

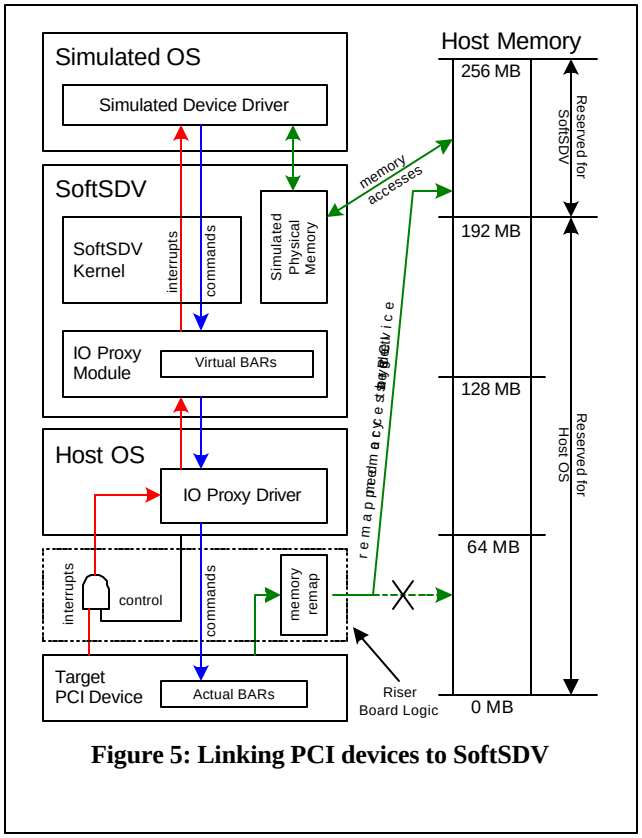

Figure 5: Linking PCI devices to SoftSDV

there is no need to physically populate actual PCI slots with various devices to create different test cases. We have experimented with other types of pseudo devices, such as an IO-monitoring service, which would enable the replaying of keyboard and mouse input to a graphical windowing system in a reproducible and OS-independent manner.

IO-PROXY MODULE

Due to the broad diversity and sheer number of IO devices in existence, we realized early in the development of SoftSDV that it would be impossible to write software models of all IO devices of interest. Indeed, some IO devices are so complex that modeling them in software would be considerably more work than the ultimate goal of porting their device drivers to IA-64.

An alternative solution is to link existing hardware devices directly to SoftSDV so that they can be accessed and programmed by simulated IA-64 device drivers under development. SoftSDV accomplishes this through a combination of hardware and software components, shown in Figure 5. The hardware components consist of an arbitrary target PCI device plugged into a custom-designed riser board that performs memory and interrupt-remapping functions. The software components include a SoftSDV module and a host OS device driver, which function as proxies for interactions between the actual

hardware PCI device and the simulated device driver. The proxy components support three forms of interaction between the target device and the simulated device driver running on SoftSDV:

- commands issued from the driver to the device via IO-port and memory-mapped IO accesses
- physical-memory accesses by the device
- interrupts from the device to the simulated driver

To support the largest possible set of devices, SoftSDV enables each of these types of interactions without requiring any specific knowledge of the target PCI device. The following sections detail how each of the three forms of interaction are supported.

Command Proxying

The key difficulty in proxying commands from the simulated device driver is determining the location of the control registers for arbitrary target devices. Fortunately, standard PCI configuration mechanisms provide a means for accomplishing this.

All PCI devices support a *configuration header* visible to software through well established locations in IO-port space. The PCI configuration header includes a set of *Base Address Registers* (BARs), which the host proxy device driver reads to determine the size and location of the device's control registers in IO-port space or physical-address space. The proxy driver passes this information up to the proxy SoftSDV module, which registers access handlers for those regions of address space with the SoftSDV kernel. The registered handler ensures that the proxy module is called whenever the simulated device driver sends commands to the device. The proxy module blindly passes such commands down to the proxy driver, which in turn issues them to the actual hardware device.

The actual solution is somewhat more complex because by the time the SoftSDV simulation starts running, the host OS will have already configured the target device's BARs to avoid conflicts with other host IO devices.[2] The problem is that SoftSDV models an entirely different (simulated) view of the platform, so the simulated OS may attempt to configure the target device's control registers to a different set of locations that might conflict with other host IO devices. The proxy code solves this problem by presenting a set of *virtual BAR* values to the simulated OS, and it remaps these values to the actual BAR values used by the target device.

Remapping Physical Memory Accesses

After the simulated driver sends the device commands, the target device will typically try to access physical memory to retrieve or deposit data associated with the command. This is a problem because the device is directed to perform the operations in simulated physical memory, but the device will operate on actual host memory belonging to the host OS, causing it to crash.

The solution is to partition the host physical memory between the host OS and the simulated OS. During boot, the host OS is configured not to use a certain amount of host physical memory (in the example in Figure 5, the reserved region is 64MB, starting at 192MB in host memory). The reserved region is then mapped to the simulated memory in the SoftSDV user process.

This alone is not sufficient to solve the problem since the device will still be programmed to use physical memory in the range of 0-64MB when it should instead be accessing memory in the range of 192-256MB. The proxy code could, in principle, interpret and modify the commands it intercepts from the simulated driver and remap addresses as appropriate before passing the commands to the actual device. Unfortunately, such a solution requires specific knowledge of the device and its commands. SoftSDV instead uses a small amount of hardware remapping logic located on the riser board between the target device and its host PCI slot. The remapping logic relocates all memory accesses made by the target device to the upper partition of memory, based on flexible configuration settings that specify the size of memory and the location of the partition.

Interrupt Proxying

When the target device generates an interrupt, it is fielded by the proxy code and sent to the simulated device driver. Two problems make it difficult to perform these steps in a device-independent manner. First, since interrupt lines are commonly shared between PCI devices, the proxy code must determine whether the interrupt is coming from the target device or from some other host device. Second, the interrupt line must be temporarily deactivated; otherwise, SoftSDV (which runs as an ordinary user-level process of the host OS) will be continually interrupted, and the simulated device driver will never have a chance to run.

Both of these operations are performed with the help of some additional logic on the riser board that enables the host proxy driver to sense and mask the interrupt from the device, before it is driven onto the shared PCI interrupt line. When an interrupt occurs, the proxy driver uses this

[2] BAR registers are programmable to enable plug-and-play software to assign conflict-free locations for device control registers.

logic to determine if the interrupt is in fact originating from the target device. If so, it temporarily masks the interrupt and passes control to the simulated device driver running in SoftSDV. The simulated driver, which understands the specifics of the device, then properly dismisses the interrupt at its source in the target device.

RESULTS AND EXPERIENCES

Table 2 summarizes the speed of SoftSDV's three processor modules when running selected SPEC95 benchmarks and when booting Microsoft 64-bit Windows* 2000. It also reports accuracy of the performance analyzer relative to the microarchitecture simulator.[3] All experiments were performed on a 500-MHz Intel® Pentium® III processor-based system with 512 MB of memory.

The emulator offers simulation speeds ranging from 10 to 25 million instructions per second (MIPS) for the SPEC benchmarks, and about 3 MIPS for the simulated Windows* boot. The lower execution rate for the OS boot is due to frequent address-space generation and process switching, which increases V2H translation overheads. Even so, at an average rate of 3 MIPS, SoftSDV is able to perform a simulated Windows boot in less than seven minutes, and is able to boot most other IA-64 operating systems in under ten minutes. The performance accuracy of the emulator, however, is limited to reporting the total number of instructions executed by a workload.

With error rates ranging from about 1% to 7% and averaging 3%, the dynamic resource analyzer exceeded our goal of predicting cycle-level processor performance to within 10% of the detailed Itanium microarchitecture simulator, and it does so at simulation rates ranging from about 160 to 250 thousand instructions per second (KIPS). These speeds are sufficient for tuning compilers and large applications, which often require millions if not billions of instructions to be executed. These results suggest that by explicitly exposing ILP, IA-64 compilers not only enable simpler, higher-frequency processor implementations, but they also make possible very fast processor performance analysis. The methods used by the analyzer do, however, have their limitations. The analyzer depends on the existence of a reference microarchitectural simulator against which it can be calibrated. Also, its flexibility is somewhat limited, since it assumes an in-order microarchitecture and is unable to model hardware-controlled speculative execution.

Workload	Emulator Speed (MIPS)	Analyzer		Microarch Simulator Speed (KIPS)
		Speed (KIPS)	Accuracy (% error)	
go	15.4	237	1.18%	15.0
m88ksim	14.8	252	1.04%	34.8
gcc	10.5	224	4.96%	17.2
compress	25.3	180	7.19%	18.4
li	13.5	227	4.06%	27.2
ijpeg	28.3	162	1.97%	27.8
perl	10.5	212	1.21%	18.6
vortex	11.8	158	2.79%	22.7
NT boot	3.00	211	—	—

When flexibility and highest accuracy are required, there is no substitute for detailed microarchitecture simulation. The IA-64 microarchitecture toolset has proven itself flexible enough to rapidly explore design options, both for Itanium processors and for future IA-64 microarchitectures currently under development at Intel. The tradeoff for this flexibility and accuracy is a much lower speed of simulation, in the range of 1 to 2 KIPS.

We found building state-sharing mechanisms between the three processor models to be a very powerful capability. Had each processor simulator only been able to work independently, methods such as sampling performance over extended regions of large workloads would never have been possible. With sampling, we are able to freely select the level of speed and accuracy required for a given simulation. For the SPEC benchmarks, our experience has been that uniform sampling ratios of between 15:1 and 80:1 yield simulation results that are statistically very close to full simulation. Table 2 reports effective KIPS rates for detailed microarchitecture simulation when a sampling ratio of 40:1 is used. The speedups relative to full simulation are not a full factor of 40 because simulation speeds are somewhat lower during the beginning of each detailed sample, and because the simulation still includes the overhead of running the emulator/analyzer during fast mode. Nevertheless, sampling effectively increases the speed of simulation by more than an order of magnitude.

Our original plans were to develop software models for all important IO devices, but we quickly realized that this approach was intractable, which led us to develop the IO-proxy module. The IO-proxying approach became

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

[3] Table 2 does not report the accuracy of the other two processor modules because the emulator does not provide performance results and because we use the microarchitecture simulator as the baseline for performance (i.e., for the purposes of this paper, we consider its error to be 0%).

particularly important for more complex device types, such as SCSI controllers, ethernet interfaces, graphics adapters, and USB host controllers, all of which successfully work with the technique. Not only did this approach save effort in writing IO-device models in software, but it also better supported a broad range of OS's. Had we selected particular SCSI or ethernet controllers to model, we would be limited to supporting OS's that had drivers for those particular devices. With the IO-proxy module, OS developers can select virtually any PCI or USB device for which their OS has a corresponding driver.

We achieved our goals of supporting IA-64 software development all along the software stack. SoftSDV successfully runs:

- several commercial operating systems, including Microsoft 64-bit Windows* 2000, Trillian Linux*, IBM Monterey-64*, Sun Solaris*, Compaq Tru64*, Novell Modesto*, and HP-UX*.
- device drivers for at least a dozen complex PCI devices ranging from graphics and ethernet adapters, to SCSI and USB host controllers.
- numerous large applications
- three well-tuned IA-64 optimizing compilers

These layers of code were all working together, all exercised before silicon, and all ready for bring-up.

The real test for SoftSDV came after the availability of actual hardware SDVs. IA-64 versions of Windows 2000 and Trillian Linux* that were developed on SoftSDV booted within ten days of the availability of Itanium processor's first silicon. Drivers for complex devices, such as SCSI disks, ethernet adapters, and graphics controllers, quickly followed over the next week, and the first MP operating systems were running a week after that. Many of the problems encountered were due to setup and configuration issues with the hardware SDV platform. Once resolved, other operating systems have been brought up even more quickly. IBM Monterey-64, for example, was up and running in under three hours on a qualified Itanium processor SDV.

For further discussion of the issues involved in porting operating systems to IA-64, please see "Porting Operating System Kernels to the IA-64 Architecture for Presilicon Validation Purposes" [12], which appears in this same issue of the *Intel Technology Journal*.

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

CONCLUSION

Central to the design of SoftSDV is extensibility. By building a core simulation kernel with a general set of abstractions, we were able to unify several different simulation technologies, each with their own unique capabilities with respect to speed, accuracy, completeness, and flexibility.

SoftSDV has proven that substantial amounts of complex software can be developed, presilicon, even for an entirely new ISA such as IA-64. As a general, extensible simulation infrastructure, SoftSDV is now being used in several other efforts throughout the company, including IA-32 presilicon software development, performance simulation of future IA-based microarchitectures, and processor design validation.

ACKNOWLEDGEMENTS

SoftSDV is the product of dozens of contributors spanning three of Intel's development sites. The fast IA-64 emulator and resource analyzer were developed in the Microprocessor Products Lab (MPL-Haifa) under the leadership of Tsvika Israeli. Methods for linking IA-64 microarchitecture models to SoftSDV were developed by the Microprocessor Research Lab (MRL) in Santa Clara, while the platform and IO-device modules were developed in MRL-Oregon. The design and implementation of the SoftSDV kernel and its extensible API was a joint effort of all three groups. The IA-64 microarchitecture simulation toolset was developed in the IA-64 Product Division (IPD) by Ralph Kling, Rumi Zahir, and their teams.

Many thanks to all those who contributed: Jonathan Beimal, Rahul Bhatt, Baruch Chaikin, Stephen Chou, Maxim Goldin, Tziporet Koren, Mike Kozuch, Ted Kubaska, Etai Lev-Ran, Larry McGlinchy, Amit Miller, Kalpana Ramakrishnan, Edward Rubinstein, Nadav Nesher, Paul Pierce, Rinat Rappoport, Shai Satt, Jeyasingh Stalinselvaraj, Gur Stavi, Jiming Sun, and Gadi Ziv.

As key users of SoftSDV, special thanks go to Daniel Aw, Alan Kay, Bernard Lint, Sunil Saxena, Stephen Skedzielewski, Fred Yang, and Wilfred Yu for their always helpful feedback, and especially for making OS-porting efforts to IA-64 an enormous success.

And finally, much credit goes to our managers, Shuky Erlich, Jack Mills, Wen-Hann Wang and Richard Wirt, who somehow kept us all working together productively!

REFERENCES

[1] C. Dulong, "The IA-64 Architecture at Work," *IEEE Computer*, Vol. 31, No. 7, pp. 24-32, July 1998.

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

[3] R. Cmelik and D. Keppel, "Shade: A Fast Instruction-Set Simulator for Execution Profiling," in *Proceedings of the ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pp. 128-137, Nashville, Tennessee, May, 1994.

[4] E. Witchel and M. Rosenblum, "Embra: Fast and Flexible Machine Simulation," in *Proceedings of ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pp. 68-79, Philadelphia, May, 1996.

[5] D. Burger and T. Austin, "The SimpleScalar Tool Set," Version 2.0, *University of Wisconsin-Madison Computer Sciences Tech Report #1342*, June, 1997.

[6] A. Lebeck and D. Wood, "Active Memory: A New Abstraction for Memory System Simulation," *ACM Transactions on Modeling and Computer Systems (TOMACS)*, Vol. 7, pp. 42-77, 1997.

[7] E. Schnarr and J. Larus, "Fast Out-of-Order Processor Simulation Using Memoization," in *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pp. 283-294, San Jose, CA, October, 1998.

[8] T. Conte, M. Hirsch, K. Menezes, "Reducing State Loss for Effective Trace Sampling of Superscalar Processors," in *Proceedings of the 1996 International Conference on Computer Design (ICCD)*, Austin, TX, October 1996.

[9] M. Rosenblum, E. Bugnion, S. Devine, and S. Herrod, "Using the SimOS Machine Simulator to Study Complex Computer Systems," *ACM Transactions on Modeling and Computer Systems (TOMACS)*, Vol. 7, pp. 78-103, 1997.

[10] P. Magnusson, F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenström, and B. Werner, "SimICS/sun4m: A Virtual Workstation" in *Proceedings of the 1998 Usenix Annual Technical Conference*, New Orleans, Louisiana, June, 1998.

[11] J. Wolf, "Programming Methods for the Pentium® III Processor's Streaming SIMD Extensions Using the VTune™ Performance Enhancement Environment," *Intel Technology Journal*, Q2 1999.

[12] K. Carver, C. Fleckenstein, J. LeVasseur, S. Zeisset, "Porting Operating System Kernels to the IA-64 Architecture for Presilicon Validation Purposes," *Intel Technology Journal*, Q4 1999.

Authors' Biographies

Richard Uhlig is a senior staff researcher in Intel's Microprocessor Research Labs in Oregon. His research interests include exploring new ways to analyze and support the interfaces between platform hardware architectures and system software. Richard earned his Ph.D. in computer science and engineering from the University of Michigan in 1995. His e-mail is richard.a.uhlig@intel.com.

Roman Fishtein is a senior software engineer in the MicroComputer Products Lab in Haifa. He has been with Intel since 1995 and has been involved in many aspects of the SoftSDV project, which he is currently leading. Roman obtained B.S. and M.S. degrees in electrical engineering from the Tashkent Electrotechnical Institute of Communication. His e-mail is roman.fishtein@intel.com.

Oren Gershon joined the Intel Development Center in Haifa, Israel in 1990 and is currently a staff software engineer with MicroComputer Products Lab in Santa Clara. Oren pioneered the design and development of several key SoftSDV technologies and has been involved with the project since its inception in 1993. Oren earned the Intel Achievement Award for this work. During a recent stint in MRL Santa Clara, Oren has been researching advanced performance analysis and visualization tools for future IA-64 implementations. Oren obtained a B.S. degree in computer science from the Technion, Israel Institute of Technology in 1992. His e-mail is oren.gershon@intel.com.

Israel Hirsh is a senior software engineer in the MicroComputer Products Lab in Haifa, Israel. He has been with Intel since 1992 and has made major contributions to SoftSDV during the five years that he led the project. Israel obtained a B.S. degree in computer science from the Technion, Israel Institute of Technology in 1978. His e-mail is israel.hirsh@intel.com.

Hong Wang is an engineer working in the IA-64 Architecture and Microarchitecture Research Group of Intel MRL in California. He has been active in building and enhancing state-of-the-art simulation infrastructures for IA-64. Hong is an avid crank on entropy, FFT, and variational principles of physics. His current interests are in discovering unorthodox ideas and applying them to next-generation IA-64 processor designs. Hong earned his Ph.D. degree in electrical engineering in 1996 from the University of Rhode Island. His e-mail is hong.wang@intel.com.