

PA-RISC to IA-64: Transparent Execution, No Recompilation



HP's Aries emulator combines fast code interpretation with dynamic translation to execute PA-RISC applications transparently and accurately on IA-64 systems running HP-UX. HP plans to bundle Aries on all such systems, so users will merely install and run applications as they would on a native PA-RISC platform.

Cindy Zheng
Carol Thompson
Hewlett-Packard

Making the transition to a new architecture is never easy. Users want to keep running their favorite applications as they normally would, without stopping to adapt them to a different platform. For some legacy applications the problem is more severe. Without all the source code, it is impossible to recompile the application to a new platform. Thus, porting these legacy applications is not just slow—it is impossible.

Binary translation helps this transition process because it automatically converts the binary code from one instruction set to another without the need for high-level source code. There are many kinds of binary translation (see Eric R. Altman et al., "Welcome to the Opportunities of Binary Translation," p. 30), but users typically must trade off between some form of interpretation (or emulation) and static translation. *Interpretation* requires no user intervention, but its performance is slow. *Static translation*, on the other hand, requires user intervention but provides much better performance.

To help PA-RISC (precision architecture-reduced instruction set computing) users migrate to its upcoming IA-64 systems, Hewlett-Packard has developed the Aries software emulator. As this article goes to press, Aries is the only software-based IA-64 migration product available. It is also unique because it combines fast interpretation and dynamic translation. Using its fast interpreter, Aries accurately emulates a complete set of PA-RISC instructions with no user intervention. During interpretation, it monitors the

application's execution pattern and translates only the frequently executed code into native IA-64 code at runtime. At the end of the emulation, Aries discards all the translated code without modifying the original application. Thus, dynamic translation provides fast emulation *and* preserves the integrity of the emulated PA-RISC application.

With this combination of fast interpretation and dynamic translation, users can expect to execute PA-RISC applications transparently, accurately, and efficiently on any IA-64 system running the HP-UX operating system. HP plans to bundle Aries on all such systems, so users will merely install and run applications as they would on a native PA-RISC platform.

This combination also provides instruction set architecture (ISA) emulation with a lower cost and higher performance relative to other emulation approaches. The fast interpreter emulates instruction blocks that rarely execute, while the dynamic translator boosts emulation performance by translating instruction blocks that frequently execute into native IA-64 code. For common applications that spend most of their time in a small part of code, emulation costs go down significantly because Aries translates only the most frequently executed code into native IA-64 instructions. At the same time, Aries boosts performance because translated code executes significantly faster than emulated code.

The resources in the IA-64 architecture also make it easy to translate PA-RISC instructions. The IA-64 architecture has 128 general and 128 floating-point registers, versus the 32 general and 32 floating-point

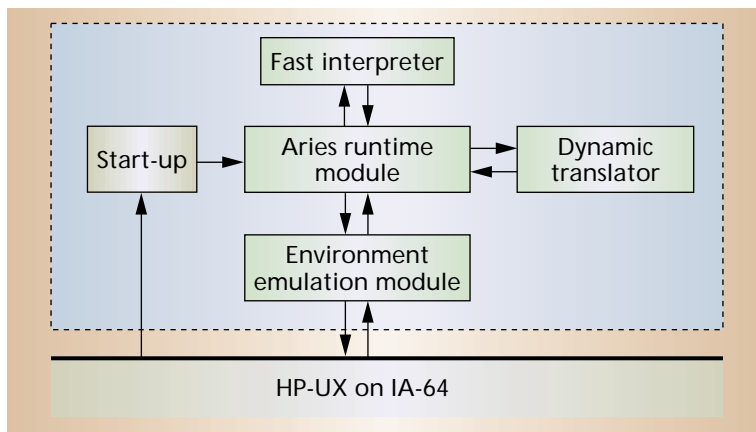


Figure 1. Aries' five main components. The runtime module controls most of the emulation process, so there is no need for the user to do anything offline. The start-up module has only one job: to invoke the Aries runtime module in response to the HP-UX operating system kernel.

registers in the PA-RISC architecture. Aries can use the additional IA-64 registers to remove resource constraints in the original PA-RISC binaries. IA-64 also contains more parallel execution units than PA-RISC, which means an emulator can exploit more instruction-level parallelism (ILP).

Finally, runtime information helps the dynamic translator produce more efficient code because it then knows more about how the code will behave when executed. For example, if it knows that 95 percent of the time an indirect branch goes to the same target, it can generate a direct branch to that target along with the generic indirect branch. Thus, it turns a hard-to-predict indirect branch into a branch that is 95 percent predictable. This is particularly important in reducing the overhead of indirect branch prediction, which is usually high in application execution.

HOW IT WORKS

Figure 1 shows Aries' five main components—the start-up module, fast interpreter, dynamic translator, environment emulation module, and Aries runtime module. Aries boosts emulation performance because it translates code at runtime and interprets only instruction blocks that have executed a certain number of times. When the user executes an application of any kind on the IA-64 system, the HP-UX kernel checks the application's header to see if it is a native IA-64 or PA-RISC executable. If it detects a native IA-64 executable, the HP-UX kernel starts a normal execution process. If it detects a PA-RISC executable, it maps the executable's text and data segments, loads the Aries start-up module, and transfers control to it. The start-up module then loads the other Aries components and transfers control to the Aries runtime module, which invokes the interpreter to start the actual emulation.

The runtime module tracks how often each block executes. If a block has executed a fixed number of times (meets a translation threshold), the runtime module then invokes the dynamic translator, which translates the PA-RISC block into a block of native IA-64 instructions that are functionally equivalent to that PA-RISC block. These instructions are called dyncode. For any subsequent emulation of that PA-RISC block, now a "hot block," the Aries runtime module

will call the hot block's dyncode instead of invoking the interpreter.

In addition to using fast interpretation and dynamic translation, Aries maps PA-RISC registers onto native IA-64 registers whenever possible to boost emulation performance. When Aries is executing dyncode, it maps all general PA-RISC registers onto designated IA-64 registers for quick reference. When Aries is interpreting, it maps these registers in memory so that it can easily use a high-level language to manipulate them. When Aries switches back to dyncode execution, it loads general PA-RISC registers into designated IA-64 registers. When it goes back again to interpretation mode, Aries flushes the PA-RISC registers back into memory.

As Aries translates more PA-RISC blocks into dyncode, it spends more time in dyncode, and there are fewer mode switches. In contrast, Aries maps PA-RISC floating-point registers in memory throughout emulation because single floating-point registers are paired in the PA-RISC architecture.

A PA-RISC application may request a system service, such as opening a file, from the underlying operating system during its execution. When Aries emulates such an application, it invokes the environment emulation module to handle these system service requests, which are primarily system calls and signals, by emulating their behaviors on the IA-64 platform. Because the IA-64 platform also runs HP-UX, Aries can map most PA-RISC/HP-UX system services directly to the corresponding system services on IA-64/HP-UX systems.

Start-up and runtime modules

The start-up module is not really part of the actual emulation process; it is there to load the other Aries components and transfer control to the Aries runtime module when the HP-UX kernel detects a PA-RISC application. Both Aries and the Aries start-up module are built as shared libraries to preserve the identity of a PA-RISC application during emulation. Thus, when the user sees the running process, he sees the original PA-RISC process, not an Aries process.

The runtime module is the hub of Aries. It is responsible for steering the control flow within a running emulation process. It tracks how many times a block has executed and decides when to invoke the dynamic translator to translate a PA-RISC block. It also determines when to invoke the environment emulation module to handle a system service request.

Interpreter

Figure 2 shows the detailed instruction emulation path. The fast interpreter is the emulation safety net because it handles all possible blocks, including those not yet translated or too complex to translate. Certain PA-RISC instructions have no simple matching IA-64

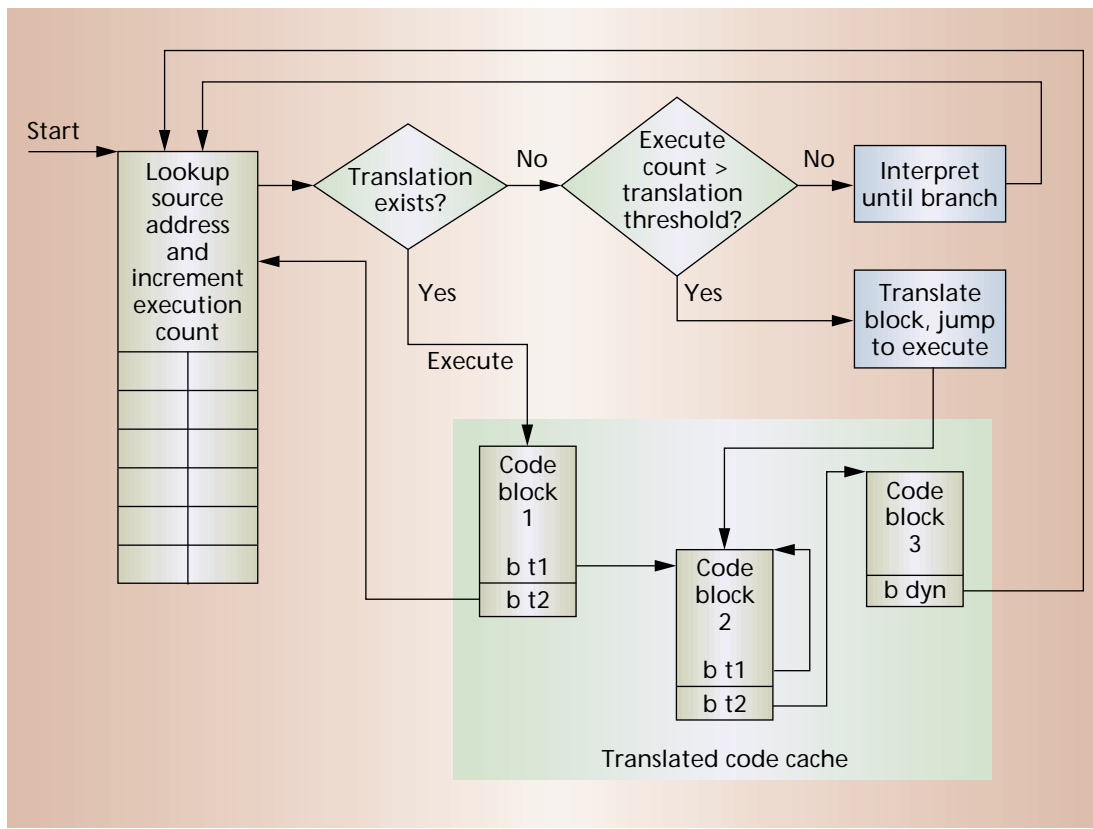


Figure 2. How Aries interpretation and dynamic translation work. When Aries starts emulating a PA-RISC application, it invokes the interpreter to emulate instructions. The interpreter interprets one instruction at a time until it reaches a branch. It calculates the branch target and returns it to the Aries runtime module. The runtime module looks up the returned branch target from an address map table to determine whether or not Aries has translated the target block into dyncode (translated code containing native IA-64 instructions). If it has, the Aries runtime module switches control to the corresponding dyncode and executes it directly. If it has not been translated, the runtime module looks at how many times that block has executed and compares it against a translation threshold to determine if the target block is ready for translation. If it does not qualify, the Aries runtime module increments the execution count for that block and returns control to the interpreter. If the block has reached the translation threshold, the Aries runtime module invokes the translator to produce dyncode. Aries then stores the dyncode in a dedicated translated code cache for subsequent execution.

instructions that are functionally equivalent. For these, interpretation makes more sense than translation into native IA-64 instructions. For example, the floating-point status register (FPSR) is mapped onto FR0 in PA-RISC architecture. To manipulate the value of the FPSR, an application uses a load or store instruction with FR0 as the target register. Aries cannot easily translate such a PA-RISC instruction into a simple sequence of IA-64 instructions because it does not map the emulated PA-RISC FPSR entirely onto the native IA-64 FPSR during emulation. Thus, Aries interprets any PA-RISC basic block that contains this kind of instruction instead of translating it.

The interpreter must also collect runtime profile information, such as taken branch targets. The Aries runtime module uses this information later to help it select and translate hot blocks (blocks that have met the translation threshold).

The interpreter also identifies blocks that are difficult to translate, as in Figure 3, marking them as “bad blocks” so that they will never be sent to the dynamic translator.

Dynamic translator

The dynamic translator translates a basic block of PA-RISC instructions into dyncode and stores the translated code into the Aries code cache for subsequent use. It has four subcomponents.

PA-RISC preprocessor. The PA-RISC preprocessor scans through each PA-RISC instruction in the block and records useful information for subsequent code generation. It also performs some pretranslation optimizations that are specific to PA-RISC architecture. For example, most PA-RISC arithmetic instructions generate carry/borrow bits that are rarely used. To minimize the redundant generation of carry/borrow bits, the preprocessor tracks information about where a resource (like a register) is defined and where it is being used in an execution sequence. Thus, the code generator produces only the necessary carry/borrow bits. The preprocessor also performs constant and copy propagation to reduce dependencies among PA-RISC instructions so that the scheduler can exploit more ILP.

Code generator. The code generator translates the preprocessed PA-RISC instructions into native IA-64

Original PA-RISC block:

0x2000	CMPB,=,N r1,r2,	0x1000	(B1)
0x2004	CMPB,<, r3,r0,	0x3000	(B2)

Possible execution sequences:

Cycles	Case 1 B1 taken B2 taken	Case 2 B1 taken B2 fallthru	Case 3 B1 fallthru B2 any
T	0x2000	0x2000	0x2000
T+1	0x2004	0x2004	0x2004 (N)
T+2	0x1000	0x1000	0x2008
T+3	0x3000	0x200C	0x200C
T+4	0x3004	0x2010	0x2010

Figure 3. A bad instruction block in Aries. This block ends with a nullifying backward branch with a delay slot (instruction following the branch) that is also a branch. Executing this basic block can produce three possible execution sequences. When both branches (B1 and B2) are taken, as in case 1, the execution sequence jumps directly from instruction 0x1000 to instruction 0x3000 without executing a block-ending branch instruction for the block starting at 0x1000. This violates Aries' translation rules. The Aries interpreter is responsible for identifying these blocks so that Aries never attempts to translate them. This kind of bad block is extremely rare in normal PA-RISC applications, however.

instructions. Because Aries maps all PA-RISC general registers onto designated IA-64 registers in dyncode, the code generator can use the corresponding IA-64 registers to reference the PA-RISC general registers directly. This eliminates the need to fetch register values from memory. The code generator also resolves any mode differences between PA-RISC and IA-64 processes. For example, if Aries is emulating a 32-bit PA-RISC application, it must adjust address references to 64 bits. For each memory-related PA-RISC instruction, the code generator must generate an extra IA-64 instruction, `addp4`, to do the conversion before it generates a load or store instruction. This process is called *address swizzling*.

Optimizer and scheduler. The code generator then passes the IA-64 instructions to the lightweight optimizer for optimizing and scheduling. Because it performs optimizations at runtime, they must be fast and effective. Aries uses several techniques to promote efficient optimization, including, among others:

- *Dead code elimination.* Aries removes redundant instructions to reduce the final translated code size.
- *Address swizzling reduction.* Aries replaces certain `addp4/load` or `addp4/store` instruction pairs with a single load or store instruction to reduce total code size and total execution cycles. Figure 4 shows how the optimizer uses address swizzling reduction to optimize a sequence of consecutive memory access instructions.
- *Memory aliasing reduction.* Aries distinguishes the

memory access instructions it generates for register fetching from the normal memory instructions it generates for the emulated PA-RISC application. These two types of memory instructions access different memory segments and do not overlap. Aries can safely move one type of memory instructions across the other type to improve ILP.

Aries' list scheduler bundles instructions for each generated IA-64 block so that all instructions fit into IA-64 templates. It starts by building a directed acyclic graph (DAG) to capture all IA-64-specific and microarchitecture-specific dependencies such as write-after-read (WAR), read-after-write (RAW), and write-after-write (WAW) hazards between instructions. On the basis of the DAG, it then selects instructions that are free to be scheduled in each cycle. Finally, the scheduler uses a state machine to bundle instructions in each cycle, inserting NOPs (no operations) as necessary. It also uses a heuristic to reduce the number of NOPs inserted.

Instruction packer. The instruction packer packs the scheduled IA-64 instructions into binary code and writes it into the Aries code cache. The Aries runtime module then updates the address map table to reflect the state change for the translated PA-RISC block. For subsequent emulations of that block, the Aries runtime will use the translated code instead of invoking the interpreter.

Aries implements a *backpatch* technique that allows a dyncode block to directly branch to another dyncode block without going through a target lookup, making it more efficient to transition between blocks. The Aries runtime module keeps track of the dyncode block that has just been executed. If the next block to be executed is the target block of the previous one, Aries modifies the final branch instruction in the previously executed dyncode block so that it can jump directly to the target dyncode block.

Aries can also translate dynamically generated code and self-modifying code in an emulated PA-RISC application, treating both types of code as regular PA-RISC blocks in an emulated application. When Aries encounters a `sync` instruction, which indicates the existence of self-modifying code, it simply erases the current content of the code cache so that subsequent emulations will not use any translations of the old code.

Environment emulation module

The most common system services that the environment emulation module must handle are system calls and signal delivery.

System calls. All HP-UX system calls enter kernel space through a common system-call-gateway page. The environment emulation module captures system calls made in an emulated PA-RISC application at the gateway page and calls the corresponding emulation

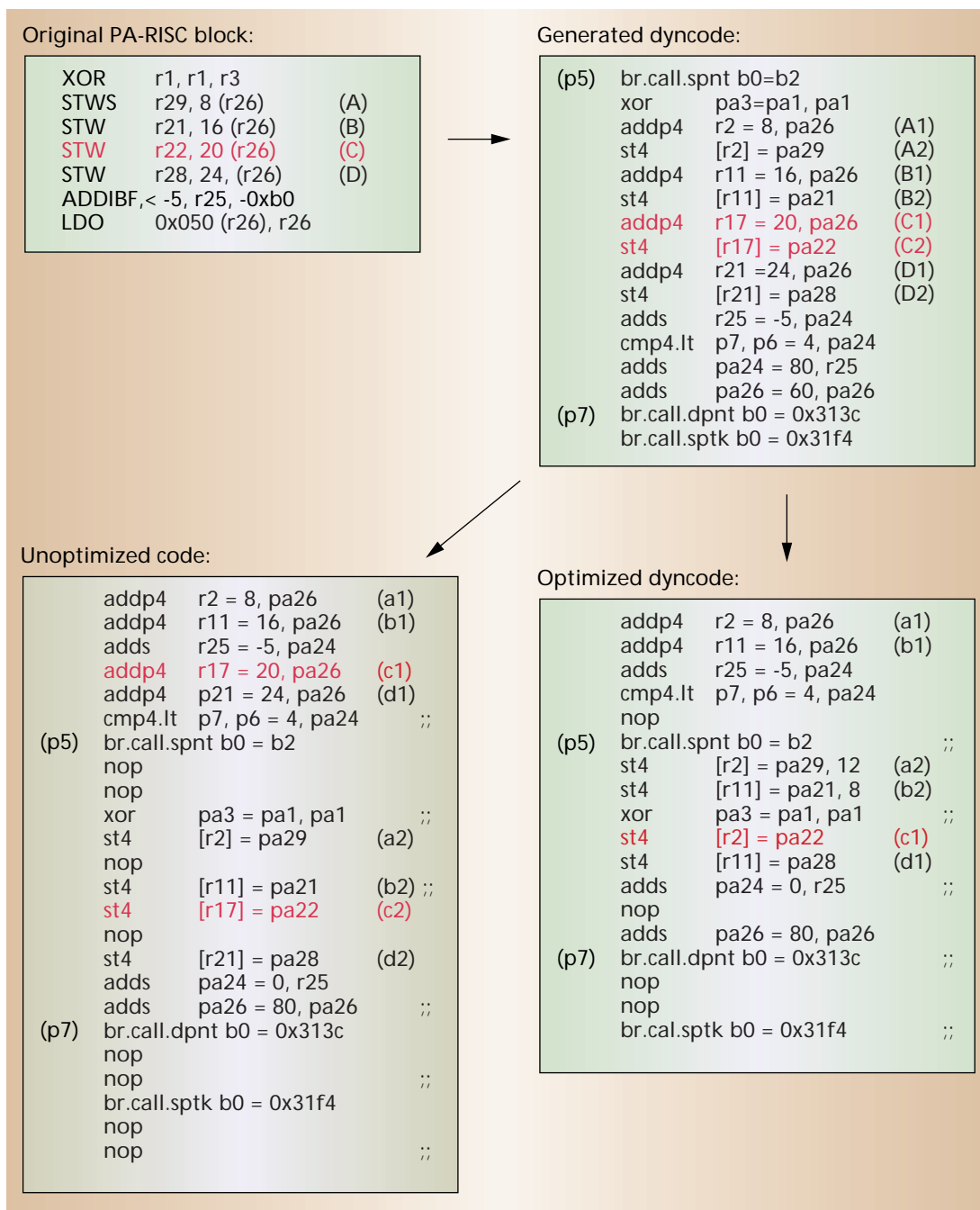


Figure 4. How the Aries optimizer reduces address swizzling (adjusting 32-bit addresses to 64-bit addresses) to optimize a sequence of consecutive memory access instructions. The optimizer replaces an addp4/st4 instruction pair (instructions C1 and C2 in generated dyncode) with a single st4 instruction (instruction c1 in optimized dyncode) that performs simultaneous base updates. The optimization shortens the total execution from six cycles to five and reduces the instruction count from 24 to 18.

routines. Most system-call emulation routines are simple stubs that invoke the native system calls directly on the IA-64/HP-UX platform. Other system calls require special handling before the native system calls are made. For example, when a thread in a multithreaded PA-RISC application requests the operating system to suspend another thread, Aries cannot simply pass this request to the underlying IA-64 kernel because it could cause a deadlock on shared Aries resources. Aries must first acquire all the Aries shared

resources before sending the native suspension request to the kernel.

Signal delivery. Signal delivery also requires special handling. The HP-UX operating system can deliver both synchronous and asynchronous signals to a PA-RISC application. It delivers a synchronous signal, such as a protection violation on a load, immediately to an application at the instruction that caused the exception. An asynchronous signal, such as a kill or suspend, on the other hand, is not associated with a

One of Aries' most important goals is to emulate all user-level PA-RISC applications on IA-64 platforms—including applications not yet developed.

particular instruction, so the system can deliver it to the application any time, and the time at which it arrives may differ from run to run.

Aries registers a master signal handler to handle the delivery of all signals it receives, whether synchronous or asynchronous, to an emulated PA-RISC application. When the HP-UX kernel detects an exception that a PA-RISC application generates, it delivers a signal to the Aries process that is emulating the PA-RISC application by invoking the Aries master signal handler. The signal handler then determines how the signal should be delivered to the emulated application. Aries does not always deliver asynchronous signals as soon as they occur. Instead,

it queues up the asynchronous signals it receives and delivers them to the emulated application at the earliest locations where it can construct a correct PA-RISC signal context for the emulated application. Aries handles the synchronous signal delivery immediately. When Aries receives a synchronous signal in dyncode, where the PA-RISC signal context may not be up to date, Aries constructs a recovery block for the dyncode. It then executes that recovery block to synchronize the PA-RISC context before delivering the signal to the emulated application.

VERIFYING EMULATION AND TRANSLATION

One of Aries' most important goals is to emulate *all* user-level PA-RISC applications on IA-64 platforms—including applications not yet developed. It is impossible to even run all the available PA-RISC applications on Aries to verify its emulation correctness. Moreover, most existing PA-RISC applications are compiler generated. Because compilers use only a subset of the ISA to generate executables, it would be hard to get 100 percent coverage on ISA emulation using application testing. We therefore adopted other ways to verify Aries.

We developed a random testing framework to stress test the correctness of Aries ISA emulation. We used this framework to randomly generate PA-RISC instruction sequences and then execute each instruction sequence twice—once on a PA-RISC processor and once under Aries running on an IA-64 system. We compared the final states and labeled any inconsistency between them as an Aries emulation failure. With this framework, we thoroughly verified all ISA emulations, including scenarios that can never be generated in a real application.

We also built a runtime cross-verification mechanism into Aries so that we would know the exact location of any emulation failure. This is important when the application is large and complex, because a failure may not show up in an identifiable format (such as output) until some time after it has occurred. We

can identify the instruction block where Aries emulation failed. With this mechanism, Aries can run any large and complex application and report emulation failure at the exact place it occurred—without user intervention.

In contrast, verifying translation correctness has been a challenge, because we did not have a real IA-64 system when we developed Aries. To overcome this problem, we injected an IA-64 instruction emulator into Aries to act as an execution bed only for dyncode. We built the rest of Aries' emulation components—the interpreter, dynamic translator, environment emulation module, and runtime module—as a PA-RISC application and ran them on a PA-RISC platform. This verification approach improved our dyncode testing efficiency by up to 300 times, compared to the traditional verification method using a full-blown IA-64 simulator.

Aries can emulate most user-level applications built for HP-UX/PA-RISC systems, including ones still under development. However, there are a few exceptions. For example, it cannot correctly emulate a debugger built for HP-UX/PA-RISC systems because of optimizations in the translated code. Also, it cannot yet emulate applications that link in both PA-RISC and IA-64 shared libraries.

We view dynamic translation as an important migration method, but software migration is only one of the many areas that could benefit. This technology could aid runtime instrumentation and profile gathering in a performance analysis toolkit, for example. It could also become central to runtime optimization in products such as the Java virtual machine. In the meantime, we see it as essential in helping HP customers enjoy an effortless and successful transition to more powerful IA-64 systems. ★

Cindy (Qinghua) Zheng is a senior software design engineer in the Adaptive Systems section at Hewlett-Packard's Enterprise Java Lab. She received a BSc and an MEng in electrical engineering and computer science from the Massachusetts Institute of Technology. Contact her at cindy_zheng@hp.com.

Carol Thompson is the C/C++ compiler architect in Hewlett-Packard's Development Environment Solutions Lab. Her background includes optimization and architecture definition for the IA-64 and PA-RISC architectures. She received an MS in computer science from the University of California, Davis. Contact her at carol_thompson@hp.com.