Application of full-system simulation in exploratory system design and development

J. L. Peterson
P. J. Bohrer
L. Chen
E. N. Elnozahy
A. Gheith
R. H. Jewell
M. D. Kistler
T. R. Maeurer
S. A. Malone
D. B. Murrell
N. Needel
K. Rajamani
M. A. Rinaldi
R. O. Simpson
K. Sudeep
L. Zhang

This paper describes the design and application of a full-system simulation environment that has been widely used in the exploration of the IBM PowerPC® processor and system design. The IBM full-system simulator has been developed to meet the needs of hardware and software designers for fast, accurate, execution-driven simulation of complete systems, incorporating parameterized architectural models. This environment enables the development and tuning of production-level operating systems, compilers, and critical software support well in advance of hardware availability, which can significantly shorten the critical path of system development. The ability to develop early versions of software can benefit hardware development by identifying design issues that may affect functionality and performance far earlier in the development cycle, when they are much less costly to correct. In this paper, we describe features of the simulation environment and present examples of its application in the context of the Sony–Toshiba–IBM Cell Broadband Engine™ and IBM PERCS development projects.

Introduction

Recent trends in processor technology suggest that future gains in system performance will rely increasingly on integrated solutions that leverage improvements across many system components, such as memory architectures, interconnect technologies, accelerators, operating systems, compilers, and application libraries. This development has led to increased design complexity in individual system components, in the introduction of new components, and in new component interactions. This growth in complexity has placed a new emphasis on the simulation of system behavior and characteristics as a key aspect of system exploration and design. To make effective design decisions, software developers and system architects require tools that assist in quantifying the merits and limitations of specific hardware and software architecture alternatives [1]. As interactions among processors and system components become increasingly important in system design, full-system simulation has become an indispensable tool for the evaluation of new systems.

The IBM full-system simulator, internally referred to as "Mambo" [2], has been developed and refined in conjunction with several large system design projects built on the IBM Power Architecture* [3]. The POWER* technology is an instruction set architecture (ISA) that spans applications from consumer electronics to supercomputers. Power Architecture encompasses PowerPC*, POWER4*, and POWER5* processors. As one of many full-system simulators [4–6], Mambo allows a full operating system (OS), such as Linux**, to run interactively in the simulation, thereby furnishing a complete environment for applications that require interprocess or complex OS interactions. In addition, a standalone environment is provided for self-contained applications. In this environment, Mambo intercepts and marshals the application system calls to the underlying host to optimize execution. Mambo itself runs atop a variety of host OS and platform combinations. **Figure 1** shows the Mambo execution and simulation stack.

Mambo facilitates experimentation with and evaluation of a wide variety of system elements, including ISAs, address translation mechanisms, memory

0018-8646/06/$5.00 © 2006 IBM

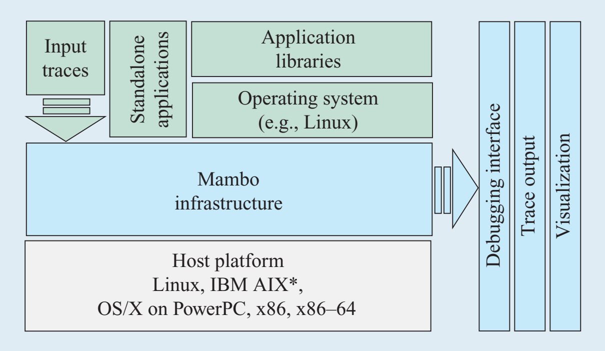


Figure 1

Mambo execution and simulation stack.

coherence protocols, hypervisor software and hardware support, hardware acceleration engines, and microarchitectural features and resource allocation. Mambo performance models may be driven by instruction or memory traces. Environment flexibility enables users to configure a simulation platform based on customer-driven application and analysis requirements.

Mambo is particularly useful in evaluating new designs that involve complex interactions among several different system components. Mambo can support detailed evaluation and testing of complex hardware interactions, such as locking and cache coherence protocols. Additionally, Mambo enables the development and tuning of production-level operating systems, compilers, and substantial application software before hardware prototypes are available. By facilitating the concurrent development of hardware and system software, Mambo can significantly shorten the critical path of development schedules. This early software development activity can also benefit the hardware development effort by identifying design issues that have an impact on functionality or performance far earlier in the development cycle, when the cost to fix such issues is considerably less.

Technology overview and design techniques

A Mambo simulation accurately models an entire system of processing elements, devices, interconnect topology, and protocols. By virtue of its execution-driven capabilities, it simulates core instruction execution and the subtleties of interactions with surrounding system components. At the heart of a Mambo simulation environment is its processor model. Mambo provides complete, bit-accurate models of several different PowerPC processors [7], ranging from the embedded 405GP and 750 processors to the PowerPC 970 (used in the Apple G5 systems and PowerPC-based blades) to the new Sony–Toshiba–IBM (STI) Cell Broadband Engine** (Cell BE) processor (used for the Sony PlayStation** 3). The Power Architecture defines the basic instructions and registers and their operation. The Mambo processor model simulates the entire instruction set, including both user and privileged-state operations, and optional features defined by the architecture, such as 32-bit and 64-bit modes and floating-point extensions. Each simulated processor is defined as a collection of these features.

In addition to the processor core, Mambo provides flexible PowerPC-based cache memory systems that contain both functional and cycle-accurate models for a data prefetch engine, L1 data and instruction caches, noncacheable unit, core interface unit, L2 cache, fabric bus controller, on-chip and off-chip L3 cache, memory controller, embedded memory, DDR2 and DDR3 memory, and the symmetric multiprocessor scalability port. Mambo also supports exploratory features, such as prefetch engine variants, nonuniform cache, L1 bypass, memory-in-processor, near-memory processing, data scatter and gather unit, high-perfomance hub, cache injection, and write-back history table.

The Mambo common bus model is designed to support pluggable bus agents, each of which is connected to a bus that behaves as the hub of a simulated system. A bus agent can be a processor, a memory controller, an input/output (I/O) unit, or a scalability port. To communicate with the rest of the system, each bus agent must first register itself on the bus, after which it communicates with the bus through a predefined programming interface.

Configurability of Mambo

A Mambo simulation incorporates a number of supported system component models, such as processors, memory subsystems, caches, buses, and I/O devices, into a full model of a system. To create a Mambo simulation, properties are defined for the simulated architecture components. In general, Mambo provides a high level of user control over the runtime specification and configuration of the simulated machine. Each component typically has an extensive set of parameters that can be specified to configure simulated system characteristics.

For example, the processor clock frequency can be defined by the user. The size of memory can be varied. Memory translation aspects, such as the size and shape of translation lookaside buffers (TLBs), can be declared. Additionally, Mambo cache memory models have more than 300 dynamically configurable parameters, giving users the ability to change almost every aspect of the simulated cache memory hierarchy. Cache parameters include size, associativity, replacement policy, latency, number of ports, number of slices, size of various queues, L3 position and use, coherence protocol, snooping sequence, and cast-out and cast-in policies. Bus parameters include topology (ring, crossbar, and point-to-point), unidirectional or bidirectional data path, frequency, and latency. Memory parameters include number of memory controllers, number of channels, read/write width of each channel, number of arrays, number of chips, number of banks, size of read/write/prefetch queue, and DRAM type and timing.

A command language is provided to model, configure, simulate, and tune components in a system. The command-line interface can also be used to control the execution of Mambo itself. Tcl [8] was chosen as the base command language and then extended with Mambo-specific commands to allow machines to be created, modified, and managed. Additionally, the selection of Tcl has facilitated the use of Tk and other packages, such as BLT [9], to provide an extensible graphical user interface (GUI).

Instrumentation of the simulated environment

Many Mambo models have been instrumented to recognize and count performance events that characterize behaviors, such as application instruction mix, pipeline stall conditions, branch and cache functions, memory and device transactions, and bus traffic. Processor, bridge, and device models can be written to expose system metrics by simulating the hardware performance monitor resources (as is done in the Mambo POWER4 processor model). Application binaries that interact with hardware performance counters and control registers can be executed without modification in simulation and on a corresponding real system, providing an invaluable way to determine how well Mambo matches hardware behaviors. Furthermore, the level and degree of instrumentation that can be incorporated in software models often surpasses what can be implemented in hardware. For example, using Mambo we are able to capture all performance events simultaneously, in contrast to current hardware, which can count only a limited number of events at a time. Mambo furnishes an inventory of mechanisms to expose event data collected in the course of simulation runs.

The Mambo emitter framework is intended to support very flexible and detailed analysis of a wide range of processor and system activity. The emitter framework decouples event generation from collection and analysis tasks. Event generation is performed by instrumentation within the simulation environment to detect events and produce emitter data. During a simulation, Mambo can identify a wide variety of architectural and programmatic events that influence system and software performance. Using configuration commands, the user can request that Mambo emit data for a specific set of events into a circular shared memory buffer.

Event processing is performed by one or more emitter readers that access and analyze emitter data. Analysis tasks generally include collecting and computing performance measurements and statistics, visualizing program and system behavior, and capturing traces for postprocessing. An Application Programming Interface (API) is provided to allow processes that read emitter data to register with Mambo and receive records buffered dynamically during a simulation run. The emitter framework is naturally suited to trace generation, and several emitter readers have been written to capture, filter, and rearrange the event stream into a number of trace file formats, such as instruction traces or memory traces. Decoupling event generation from processing affords greater flexibility in extending the data collection capabilities. SimOS [10] also separates the data collection of hardware-level event records from the data processing, which is accomplished externally to the simulation platform.

Also patterned after SimOS are Mambo *trigger* facilities. Similar in flavor to Tcl annotations, Mambo invokes user-supplied Tcl procedures associated with model behavior and simulation control events. Substring matches on text written to the simulated console may also fire trigger procedures. The body of such Tcl triggers may contain not only code to monitor and accumulate statistics, but sequences to control simulation behavior. For example, triggers can be used as a breakpoint apparatus sensitive to program counter address values, performance events, and console text strings.

Simulated applications can also control and query Mambo models through a set of *call-through* interfaces. Call-through instructions are placed in application code as specially formed no-op opcode or illegal opcode. Mambo intercepts these opcodes in simulation and performs the indicated operation on behalf of the application. Facilities exposed through these lightweight, minimally intrusive instructions allow applications to interact with the physical file system, interrogate statistics, control and annotate emitter record production, fire triggers, evaluate Tcl expressions, and direct simulator operations.

Call-through instructions are also a primary avenue for application interaction with novel devices for which no OS drivers yet exist. Device commands can be issued by the application as call-through functions that Mambo redirects to the device model. Early assessment of these novel devices is possible without having to invest in developing all the intervening paths through the software stack normally needed to access the hardware.

Mambo also supports a mechanism whereby a GNU debugger (GDB) may be used to debug the software running atop Mambo. Through this mechanism, GDB can attach to and control a Mambo simulation running on either the same or a remote machine. Mambo supports the standard debugger interface used by GDB to attach to a real system. The GDB debugger interface allows the debugger to query the state of the simulated machine (register and memory), set breakpoints, and advance the simulated machine.

Incremental performance modeling

The Mambo design has been driven by practical simulation needs. The intent behind it is to assist in a number of key aspects of the system design process: software characterization, bring-up, debug, tuning, examination of detailed architectural interactions, credible performance projections, and as a validation reference.

To address the user requirements for functional accuracy and fast execution time, Mambo uses compiletime feature selection to limit the alternatives that must be considered during runtime simulation. This results in multiple executable binaries: typically, one for each simulated system. The tradeoff between runtime feature selection and compiletime feature definition is driven by an effort to balance generality against execution speed.

Some users additionally require a cycle-accurate projection of the time to execute an application on real hardware. The most accurate projection of execution time demands modeling of structural effects of the system, particularly the parallelism present among multiple functional unit pipelines, elements of the memory hierarchy, buses, and so on. This more detailed cycle-accurate model typically results in substantially increased simulation execution time.

Mambo provides a range of options that compute more or less accurate execution time statistics for the simulated system. Mambo is designed to allow component models to evolve over time. During the early stages of system design, purely functional models for new processor and system design proposals are developed that execute quickly and are inexpensive to develop. As the hardware design matures, detailed implementation characteristics can be incorporated into the Mambo models, which can then be used for system and application performance analysis.

Mambo modeling constructs enable the development team to easily add performance model features to existing functional models. These constructs, representing execution parallelism, resource contention, delay, and transaction flow, can be incorporated incrementally. This approach permits both investigation and iterative refinement of critical performance paths in the architecture until simulation results correlate with design specifications, analytical projections, or measurement data captured from existing hardware. A Mambo model is then refined by the appropriate introduction of delays in simulation time to provide greater accuracy in timing information. For example, the time for cache operations can be approximated with constant latency, with stochastic delays, or according to precise modeling of delays based on architectural state and resource contention.

The level of timing accuracy and, consequently, the speed of simulation can be chosen as needed at runtime. Varying levels of timing accuracy can be used for the various Mambo components. For instance, to evaluate new cache designs, it may be sufficient to model the timing of the cache model in great detail while using simpler models of the processor, bus, and memory and thus allowing them to simulate more rapidly.

Mambo component models can be classified in the Virtual Socket Interface Alliance (VSIA) taxonomy [11] as either *functional* or *mixed-level*. All models are at least functionally accurate, and those that yield timing information do so by augmenting a functional representation. This approach differs from loose-coupling techniques that employ interface layering between otherwise independent cycle-accurate and functional simulation engines [12, 13].

Application of Mambo in the STI Cell BE project

In 2001, Sony, Toshiba, and IBM combined research and development efforts to create an advanced processor and system architecture for a new wave of devices in the emerging broadband era. As a result, the Cell Broadband Engine Architecture (CBEA) [14] was designed to take advantage of the most advanced IBM semiconductor process technologies to deliver high performance with good power efficiency. Shortly after project inception, STI engaged the IBM Austin Research Laboratory to develop a Mambo model for the processor and system architecture to support software simulation and performance testing. Mambo delivered vital feedback at various stages of the Cell BE project life cycle. An initial functional simulation model was developed for preliminary validation. This model was later enhanced to provide more accurate timing evaluations. The model was first based on design specifications, then validated against hardware-level simulators, and finally correlated with real hardware. The result was a completely integrated working system of Cell BE hardware and software that has been successfully launched alongside actual hardware delivery.

Because CBEA contains cores with a new ISA and a novel on-chip system structure, evaluating its performance with traditional (trace-driven) methods was not a viable option. Rather, a full-system, execution-driven simulation platform was needed to facilitate exploration of the programming model and to study interactions among processing elements and the memory hierarchy. For the STI Design Center, integrating Mambo in the product cycle enabled developers to iteratively develop multiple generations of operating systems, libraries, and game software well before any hardware was available. A clear measure of its success is demonstrated by the fact that the OS and multiple applications, which were first developed on Mambo, ran on first hardware without modification. As a result, the delivery of the STI Software Development Kit to third-party companies was accelerated, and game programmers were able to develop applications and operating systems several years before the hardware was available. This ability has been exceptionally important given that the Cell BE chip presents the game development community with a radically new programming model that requires significant rethinking of application design and implementation.

Early Cell BE environment

The availability of Mambo during the formative years of the CBEA opened an avenue for the STI software development team to contribute to the hardware design. Initially, the general design goal centered around a heterogeneous multiprocessor platform containing a 64-bit PowerPC processor and some number of smaller single-instruction multiple-data (SIMD) processors, each of which would operate out of its own local memory. The mechanism through which all of these processors would communicate was just beginning to take shape. The creation of a full-system simulation of the Cell BE design was considered to be the most effective vehicle for exploring the design and enabling the STI teams to collaborate on early software development of the entire software stack.

In addition to the directive to provide design guidance, the software team was challenged to bring up the entire software stack on the new system as soon as hardware was available. Full-system simulation of the CBEA was the only viable means of achieving this aggressive goal.

The Cell BE software team consisted of subteams responsible for developing and porting the following software components for the Cell BE: hypervisor, Linux, compilers and tools, runtime libraries, applications, debuggers, and the simulator. Coordination among the various teams necessitated that the architecture be fully specified in the design documentation. Once this was drafted, simulation support pertaining to these specifications was added. The hypervisor and OS teams then added their contributions to the simulated environment, followed by the runtime library and applications teams.

As a first step in building a Cell BE simulation model, we combined a 64-bit PowerPC processor available in the Mambo library and an early model of the synergistic processor unit (SPU) sufficient to run PowerPC and SPU code simultaneously. Because the communication mechanism between the PowerPC units and the SPUs was not yet fully specified, the simulation team added instrumentation (call-through instructions) to allow the application to perform communication tasks directly at a higher level. This first-order approximation in simulation enabled the Cell BE software team to begin application development three years in advance of hardware availability. Tight coordination among the simulation, architecture, and software development teams helped finalize the specification and enabled early feedback to be incorporated into the processor architecture.

Architecture evaluations

Every feature supported in the CBEA was exercised by the Cell BE software development team before the hardware was available. Mambo served as a proving ground for the specifications, both as a catalyst for practical discussion and as a platform to demonstrate the value of proposed features before committing them to the architecture. Mambo was key in providing support for design decisions, such as the number of SPUs, the size of the local store for each SPU, early definition and support for the memory flow controller (MFC) direct memory access (DMA) operations, operation of the synergistic processing element (SPE) channel interface (blocking as opposed to nonblocking semantics), operation of the SPE signal notification channels, and operation of the SPE atomic operations. Furthermore, Mambo served as the early proving ground for three different Cell BE programming models: data-parallel as opposed to task-parallel computation on SPEs, shared-memory parallel as opposed to message passing on SPEs, and SPE virtual device model.

Some design tradeoffs, such as the number of SPUs in the system or the size of SPU local storage, could be accommodated by simple Mambo configuration changes. Other modifications to the processor architecture were more pervasive and required substantial alterations to the Mambo models. For example, the addition of hardware multithreading to the Cell BE PowerPC processor required new thread support in Mambo. After the support was added to Mambo, changes were made to the OS to take advantage of the new feature. This change also affected the internal interrupt controller, which could now target a specific PowerPC thread. As a result, the final design was implemented in silicon with the knowledge that the OS support was written and successfully working in simulation.

The development of the research hypervisor demonstrates the value of Mambo for exploratory software investigation. The simulation environment allowed development to proceed on regular shared workstations (including x86-based systems), without the need for dedicated hardware. It was possible to trace and debug the hypervisor and step carefully through important transitions to verify that memory allocation, address translation, interrupts, and processor switching operated correctly. Providing this level of support was particularly important because crucial system software, such as the hypervisor and parts of an OS, were often very difficult to debug, especially on new hardware. With Mambo, debug trace statements were included to allow system developers to see a trace of address translation exceptions and interrupts down to the instruction level. Each exception, for example, was tagged with a message to indicate exactly the problem detected, which was of great assistance in explaining application operation, particularly in the early bring-up stages of development.

Performance evaluation tool chain

Over the course of the Cell BE project, the Mambo modeling of system performance grew as well. At first, simple fixed latencies were associated with high-level operations such as memory accesses. Caching was modeled probabilistically, using fixed cache-hit ratios. This level of modeling was useful for high-level performance analysis. Models for the caches, bus, and memory controller were developed initially to validate coherence protocols, but were subsequently extended to incorporate timing of low-level interactions. As described earlier, Mambo allows functional models to be easily extended to model timing, concurrency, and resource contention, thus preserving the initial investment in these functional models. These detailed timing models significantly improved the accuracy of Mambo performance results.

Initially, the accuracy of the performance models was evaluated on the basis of comparisons to theoretical performance estimates. After completion of the cache and bus timing models, we extended this evaluation with a comparison to SPCsim, an internal low-level performance simulator developed within the STI Design Center. At that time, access to hardware was severely constrained, which precluded a direct validation against hardware. We developed a collection of nanobenchmarks that performed a variety of low-level DMA operations from the SPUs to focus on memory interaction timings. By tuning low-level latencies and redesigning certain aspects of the models to allow greater concurrency, we achieved performance within 15% of the results from SPCsim for single SPU executions and within 26% of SPCsim in high-contention tests employing all eight SPUs.

During this same time period, we were given the task of projecting the performance of linear algebra workloads for several future SPU microarchitectural alternatives. The combined requirements imposed by these new studies and the early SPCsim comparison efforts led us to augment the Mambo data-collection capabilities. A generalized transaction-tracking facility was introduced to simplify the process of resolving discrepancies between SPCsim and Mambo MFC behaviors. Traffic could be traced by examination of tracking records presented through the Mambo emitter framework. However, this process proved to be labor-intensive, and the need for facilities more conducive to detailed performance debug and analysis became immediately apparent.

Consequently, three new graphical visualization tools were developed to give deeper insight into SPU internal operations. All GUIs were constructed as Tcl scripts exploiting the Tk and BLT functions integral to Mambo and extensions to the underlying data-collection features previously described. GUIs could easily be prototyped using this strategy. External users could compose custom visualization and analysis tools in fashion.

Figure 2(a) is a sample screen from a statistics collector that offers animated time-series plots of SPU stall conditions, issue and execution pipe characteristics, and DMA operations. BLT vector structures and charting widgets were used to collect and present low-level SPE performance event statistic data from instrumentation already available in the Mambo SPU and MFC models.

Figure 2(b) shows a transaction visualization tool that depicts SPU instructions as they flow through the internal pipeline and queue structures. The tool builds on elements developed for the aforementioned emitter-based transaction trace with the addition of data structures optimized for high-speed dynamic query and retrieval of tracking information.

Figure 2(c) displays an interactive SPU local store map. Similar to the SPU statistics collector of Figure 2(b), this tool relies heavily on the BLT vector and plotting functions incorporated in Mambo. Histograms of memory accesses are charted across the span of local store address space, which may be further inspected and disassembled. Instruction breakpoints may also be assigned using this dialog.

As an increasing number of Cell BE processor-based systems are becoming available, we have begun to validate our performance models against hardware. While not yet complete, this effort has already identified inaccuracies in the models that were not detected in earlier validation. Mambo also simulates many of the key performance counter registers of the Cell BE processor and supports access to them through the standard architected mechanisms. Validation efforts are planned to ensure close correlation between the simulated and actual performance counter results. The goal of this activity is to allow the same suite of performance analysis tools to be used on both the simulated and the actual systems.

Verification tool chain

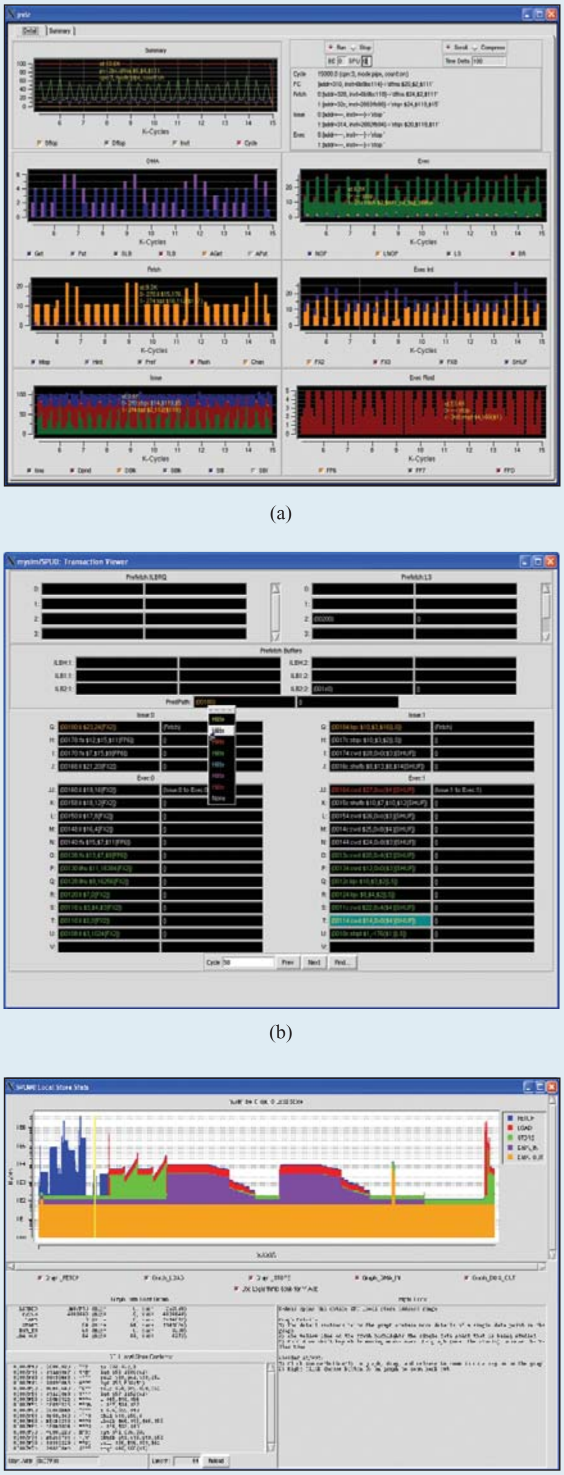
IBM has a number of methodologies that focus on verification of early hardware designs and early hardware bring-up systems. One of the tools employed for architectural verification is a random test-pattern generator, which produces thousands of tests to exercise a broad range of architectural features of a given design. These same tests are run on low-level hardware models to ensure their accuracy at the final stage of processor design, before the circuit description is sent to manufacturing. The simulation team made extensive use of these tools to validate the architectural accuracy of the core models supplied in Mambo.

In addition to these tests, the software teams developed an entire Cell BE software stack consisting of a hypervisor, Linux-based OS, libraries, and applications running on the PowerPC and SPU cores. These applications included a large number of verification tests based on the same hardware specification used by the simulation team. Any discrepancy between interpretations of system documentation made by the simulation team and those of the software team would be quickly evident as a result of these tests. The effect was to broaden the scope of testing that was available to verification. The applications exercised the runtime library that handled job scheduling, loading of state in the SPUs, and communication between the application code running on the various cores. These operations, in turn, exercised the OS support that handled translation faults from the various MFCs and caught other signals flowing between the cores.

At this point, the verification team realized that the extent of the software stack running on Mambo could actually help further verify the hardware design. The Mambo simulation team, the OS team, and the verification team collaborated on a process whereby a complete Cell BE workload was run on Mambo and captured to a system snapshot. Using these system snapshots, the verification team was able to load different system configurations into the low-level simulation environment to further test the hardware design.

System bring-up

The big test for the entire Cell BE software and simulation team came when the first hardware arrived at the bring-up laboratory. All of the simulation and software stack development was done in preparation for fast bring-up on early hardware. In general, working around early hardware errata consumes considerable time for the software bring-up teams. For the Cell BE project, we instrumented Mambo to highlight known errata conflicts in code that was run on the simulator. This helped software teams to ensure that their software was adapted appropriately to avoid early known hardware


(a)

(b)

(c)

Figure 2

Mambo visualization and analysis tools: (a) statistics collector; (b) visualization tool; (c) interactive SPU local store map.

problems. When the first systems became available, the software teams used Mambo to work around newly discovered system problems. Mambo provided a richer, more convenient debug and test environment for the software teams to complete their hardware workarounds. Once this new code worked on Mambo, it was migrated to the hardware, where it executed without further modification.

After hardware availability

Once hardware is available, the natural tendency is to shift focus to hardware execution rather than running in simulation. However, the use of simulation in the process of developing and refining the Cell BE processor system has not ended with the recent delivery of this processor. The ability of Mambo to run the exact same software stack as real hardware facilitates an easy transition both to and from hardware. The following are two realities that lead to continued simulation use, even after hardware becomes available:

- *Hardware accessibility:* It may not be practical to provide dedicated hardware access to all developers. In these situations, Mambo continues to serve as a fully functional platform for software development and testing.
- *Debug environment:* Years of software development on Mambo have led to an extensive debug environment that enables user visibility with respect to state, structures, and events not immediately accessible on the physical machine.

Application of Mambo in the PERCS project

As part of the High Productivity Computing Systems (HPCS) initiative sponsored by the Defense Advanced Research Projects Agency (DARPA) [15], IBM is researching and developing the IBM PERCS—productive, ease-to-use, reliable computing system—for implementation by the year 2010. The overall goal of the HPCS initiative is to take a broader look at high-end computing systems to encompass issues such as development time, programmer productivity, system robustness, portability, and overall system value leading to the next generation of supercomputers. PERCS will support balanced, economically viable HPCS hardware and software solutions that can be leveraged both in the national security sector and in commercial markets.

The long lead time of the PERCS project and associated ambitious plans call for examining many novel design ideas, thus precluding hardware prototyping except in the later phases of the project. In particular, the complex interactions between software and hardware must be examined closely to see whether they can be exploited to yield the desired boost in productivity. These interactions tend to be difficult to examine during regular design cycles because of budget and schedule constraints. We have decided to face the challenge partially by using the Mambo tool as a platform for prototyping new hardware features and software mechanisms.

Performance evaluation tool chain

Performance evaluation of modern systems uses one of four approaches: measurements from hardware prototypes, analytic modeling, trace-driven simulation, and execution-driven simulation. Each of these approaches has its advantages and disadvantages, and we considered all of them as alternatives for the early research phase of PERCS.

Measurements from actual hardware prototypes provide accurate performance data on the system. However, this approach requires expensive hardware prototyping, and thus is best employed in the preproduction phase after the design has been largely decided. The sheer number of ideas that we had to examine in PERCS excluded hardware prototyping as a viable approach to conduct research in the concept phase.

We decided, instead, to use a combination of analytic modeling, trace-driven simulation, and execution-driven simulation. These approaches vary in expense and accuracy but share the need for result validation. Analytic modeling does not incur much expense, but it requires the development of a matrix of application-design models, each requiring about six months of effort by a skilled developer who has a good understanding of the application and its projected behavior on the modeled platform. Trace-driven simulation does not require the laborious development of the models, but it requires the laborious and often expensive generation of application traces and may be limited in its ability to predict certain interactions, such as the memory behavior of multiprocessor systems [16]. Execution-driven simulation eliminates the need for generating and manipulating traces, and if the simulation model represents the hardware with good fidelity, the results can be as accurate as those obtained from hardware [10, 17]. An execution-driven model typically runs hundreds or thousands of times slower than hardware, with increased modeling fidelity coming at the expense of longer simulation time. The increased simulation time limits this approach to small-scale systems and may preclude the realistic modeling of detailed interactions in a processor core.

Analytic modeling

We decided to use analytic modeling for the case in which a validated application model exists on a reference design point that can be manipulated to produce answers to design alternatives quickly. One such example was for commercial applications, such as transaction-processing systems, for which we exploited existing models to test cache enhancements [18]. These applications cannot be realistically studied using a complete trace, and a simulation model would be too slow to generate useful results. Given the lack of hardware to validate the analytic models of an advanced memory hierarchy, we used Mambo to verify the results of the modified analytic model of the memory hierarchy. To do so, we changed the parameters of the analytic models for specific design alternatives and produced results for simple applications. We then used Mambo to run the same applications on an equivalent simulation model of the memory hierarchy and verified that the results of the simulation were statistically identical to those of the analytic model. It is interesting to note that this verification effort revealed bugs in both models, and our experience leads us to recommend that Mambo be used as a reality check for analytic models whenever appropriate in the future.

Trace-driven simulation environment

Trace-driven simulation is most applicable to examining ideas and research in processor cores and, in some cases, the memory hierarchy. Trace-driven simulation requires the laborious generation and maintenance of traces. Several tools exist for generating traces from actual hardware [19], but they tend to be cumbersome and expensive to operate. In some cases, these tools may require intrusive support in the processor or the memory subsystem, or they may not be able to capture the traffic at all levels in the memory hierarchy. For example, it may not be possible to capture traces of on-chip Ll cache accesses without specialized processor support.

Once a trace is generated, it typically requires some filtering—for instance, to extract the portions of the traces that may be deemed interesting for a particular study. Statistical sampling and other methods have been proposed in the literature [20], but these require a great deal of labor and they may require a validation that the sampled traces do not lose useful information. Finally, traces require prodigious storage space and bandwidth. Several trace-compression mechanisms have been implemented in previous work [21], but they add more processing overhead to the simulation for decompression and require further effort on the part of the trace maintainer.

We have used Mambo to mitigate these problems with trace-driven simulation. Our methodology is to first run the application through existing hardware and capture the characteristics of the application during its execution using hardware performance counters. For instance, it may be expedient to focus on instructions per cycle (IPC) as a metric for describing the performance of an application as it passes through various phases of program execution. A profile of IPC behavior over time is then captured, as shown in **Figure 3**.

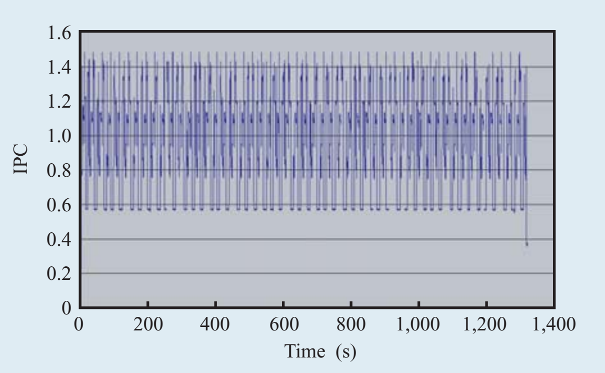


Figure 3

IPC behavior over time.

The IPC curve can be examined to identify intervals of low IPC. These intervals of the program presumably challenge the existing hardware architectures and may reveal serious bottlenecks in the hardware, software, or both. The number of executed instructions in each interval is determined using the performance counters. These intervals define the starting points for generating application traces. We feed these starting points to Mambo, which runs the application to generate the traces for the desired intervals. These intervals can also be customized depending on the requirement of the target study itself. For instance, to produce traces that are useful for core studies, it may be necessary to prepend each interval with the preceding million instructions or so to warm up the L1 cache in a trace-driven simulation.

OS interactions can be included or excluded from the trace on the basis of the mode in which the simulator is running: full-system or standalone. We have generated full traces of applications that also include the OS interactions.

Note that it is possible to do away with the need to store traces in stable storage. Using the emitter interface, the output from Mambo can be fed directly into the trace-driven simulation. The overhead of this on demand generation of a trace can be intuitively compared with the overhead of reading the large trace file from stable storage and performing the actual decompression. In contrast, however, our approach does not require the storage and manipulation of compressed traces.

Execution-driven simulation tool

Execution-driven simulation encompasses every aspect of the system in great detail. The result is a model of a system that can generate performance results close to those obtained from an actual system [10]. This form of simulation has a tradeoff between accuracy and simulation time in which accuracy increases with time, but often in a less than linear fashion. Choosing the proper point in that tradeoff remains an art.

We have used Mambo to build a cycle-accurate cache model to simulate the effects of new memory hierarchy techniques on application performance. The model includes fairly detailed simulation of a state machine that governs the cache replacement and inclusion protocols and simulates the way the cache interacts with main memory and the processor core. Rather than construct a full cycle-accurate core model, we implemented an abstract core model that may be tuned to provide different IPCs. Thus, we studied the effect of cache hierarchy innovations with aggressive or conservative core designs by selecting different parameters. The goal here was to produce a range of plausible figures within a spectrum of core designs from the most conservative to the most aggressive. We measured the effect of each innovation on cache hit ratios and memory latency on average, and the outcome was a dynamic range of values that show the effectiveness of the various innovations. This approach can be used to eliminate design choices that are likely to produce minimal performance improvements. We have used this facility to study the cache behavior of nonuniform cache architectures [22], near-memory processors, and memory-in-processor morphing of cache architectures to direct-addressable memory. It is useful here to consider the last one as an example of the way that Mambo can actually identify serious problems with an idea that is initially perceived as beneficial.

The MIP morphing of the cache architecture calls for setting aside a portion of the cache as direct-addressable memory. The idea has been tried in embedded systems, such as the PowerPC 401* processor, and in some past IBM mainframes. It has not been tried at the scale of today's high-end processors. The premise of the idea is this: Giving the application control over some portion of the cache allows the cache behavior to become more deterministic, which is good for performance tuning and programmer productivity. In addition, this arrangement can improve performance because access to the direct-addressable on-chip memory does not go through the cache directory phase, thus reducing access time and power consumption. Such memory can also be used by the OS to store some important structures, which can speed up the context switching of applications and improve overall system performance.

Initially, the MIP morphing feature was well received by internal development and HPCS customers alike. The team was aware of the need to both manage this hardware feature at the OS level and provide portable abstractions to the programmer to exploit this feature in a productive way. None of these problems appeared insurmountable, given the large body of academic research that has addressed this problem [23]. Mambo was used as a platform for exploring the performance aspects of this feature and examining potential software issues on this yet-to-be-built hardware. The implementation effort was facilitated by Mambo, allowing the OS team to prototype the MIP morph idea in a controlled development environment. Taking the prototyping effort to this level of realism uncovered many complexities in supporting the MIP morph in a virtualized manner. These complexities included partitioning the cache between cacheable memory and direct-addressable memory, bringing a process into a different processor after a context switch, and managing coherence if threads on different cores or different processor chips were to access the local memory. Other complexities—such as hypervisor support, page table construction, and compiler layout of the memory space—were confronted as well. The results showed that the conventional wisdom concerning MIP morphing did not go deep enough in examining the issues. By prototyping the software support that was needed at the OS level and exposing the usage issues at the application programmer's level, the magnitude of the problem was exposed at its fullest. Further, the improvement in performance did not show a sufficient payback for the immense effort that would be required at the software level to support the idea, and as a result it was dropped from further consideration.

Summary and future direction

Mambo has proven to be a critical part of several projects developing new PowerPC processor systems. It allows new features to be prototyped quickly and evaluated in the context of a full-system simulation. It has been used successfully to develop a complete system software stack before hardware availability. Mambo also plays a role in the performance evaluation environment as a validation tool for analytic models, as a trace generation and manipulation facility in trace-driven environments, and as an execution model with the right balance between simulation speed and result accuracy. As a result, the designer can focus on analyzing the application and on choosing the relevant parts of the execution or system that must be examined closely.

We are continuing to use Mambo for new processor and system development. IBM Research actively continues to collaborate with teams within IBM, in academia, and in various government agencies to ensure that new features and support for performance innovations are consistently built into the Mambo model. Future versions of Mambo will include enhancements to the core simulator infrastructure to provide greater usability and enrich the overall simulation environment. Also, the Austin research team has added Mambo for the PowerPC 970 and STI Cell BE processor to the alphaWorks [24] family of emerging technologies and will continue to provide additional variants to the research and development community.

At the same time, we are extending Mambo in several directions. Current research includes efforts to extend more of the Mambo model of the simulated system to be defined at runtime, allowing greater flexibility in the range of systems that can be explored. There is also a new effort to provide more accurate models of complex, out-of-order, superscalar processor designs. We are constantly working to improve its simulation speed and scalability to support its role as a platform for software development.

Acknowledgments

This work has been supported in part by the Defense Advanced Research Projects Agency (DARPA) under Contract No. NBCH30390004. Mambo is managed as an open source project within IBM and has an extensive set of contributors throughout the company. We acknowledge the Austin Research Laboratory team for its focus on continual innovation, which ensures that new features and support for new technologies are consistently built into the Mambo simulator. We also acknowledge the following individuals external to the Austin research team who have assisted in developing and extending Mambo features and functionality: Max Aguilar, Michael Day, Jonathan J. DeMent, Stan Gowen, Sid Manning, Mark Nutter, Mike Stafford, and James Xenidis.

*Trademark, service mark, or registered trademark of International Business Machines Corporation.

**Trademark, service mark, or registered trademark of Linus Torvalds or Sony Computer Entertainment America Inc. in the United States, other countries, or both.

References

1. P. Bose, “Performance Analysis and Verification of Superscalar Processors,” *Research Report RC-20094*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598, June 1995.
2. P. Bohrer, M. Elnozahy, A. Gheith, C. Lefurgy, T. Nakra, J. Peterson, R. Rajamony, R. Rockhold, H. Shafi, R. Simpson, E. Speight, K. Sudeep, E. Van Hensbergen, and L. Zhang, “Mambo—A Full System Simulator for the PowerPC Architecture,” *ACM SIGMETRICS Perform. Eval. Rev.* **31**, No. 4, 8–12 (March 2004).
3. C. May, E. Silha, R. Simpson, and H. Warren, *The PowerPC Architecture: A Specification for a New Family of RISC Processors*, Morgan Kaufmann Publishers Inc., San Francisco, 1994.
4. R. Uhlig, R. Fishtein, O. Gershon, I. Hirsh, and H. Wang, “SoftSDV: A Pre-Silicon Software Development Environment for the IA-64 Architecture,” *Intel Technol. J.* **3**, No. 4 (November 1999); see *ftp://download.intel.com/technology/itj/q41999/pdf/softSDV.pdf*.
5. M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta, “Complete Computer System Simulation: The SimOS Approach,” *IEEE Parallel & Distr. Technol. Syst. & Appl.* **3**, No. 4, 34–43 (December 1995).
6. Simics white papers, Virtutech Inc., 2005; see *http://www.virtutech.com/our-tech/white-papers.pl*.
7. IBM Corporation, Power Architecture; see *http://www-03.ibm.com/chips/power/index.html*.
8. J. K. Ousterhout, “Tcl: An Embeddable Control Language,” *Proceedings of the USENIX Winter Conference*, January 1990, pp. 133–146.
9. M. Harrison, *Tcl/Tk Tools*, O’Reilly & Associates, Sebastopol, CA, 1997.
10. S. A. Herrod, “Using Complete Machine Simulation to Understand Computer System Behavior,” Ph.D. dissertation, Stanford University, CA, February 1998.
11. VSI Alliance, System Level Design Development Working Group, “VSIA System Level Design Model Taxonomy Document, Version 2.1 (SLD 2 2.1),” July 2001.
12. H. Wang, S. Manor, D. LaFollette, N. Nesher, and K.-J. King, “Inferno: A Functional Simulation Infrastructure for Modeling Microarchitectural Data Speculations,” *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, March 2003, pp. 11–21.
13. J. Emer, P. Ahuja, E. Borch, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan, “Asim: A Performance Model Framework,” *IEEE Computer* **35**, No. 2, 68–76 (February 2002).
14. Cell Broadband Engine Architecture; see *http://www-128.ibm.com/developerworks/power/cell/*.
15. R. Graybill, “High Productivity Computing Systems (HPCS),” DARPA Information Processing Technology Office, June 2005; see *http://www.darpa.mil/ipto/Programs/hpcs/*.
16. R. A. Uhlig and T. N. Mudge, “Trace-Driven Memory Simulation: A Survey,” *ACM Computing Surv.* **29**, No. 2, 128–170 (June 1997).
17. D. Burger and T. M. Austin, “The SimpleScalar Tool Set, Version 2.0,” *Computer Arch. News* **25**, No. 3, 13–25 (June 1997).
18. E. Speight, H. Shafi, L. Zhang, and R. Rajamony, “Adaptive Mechanisms and Policies for Managing Cache Hierarchies in Chip Multiprocessors,” *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, June 2005, pp. 346–356.
19. F. E. Levine, B. C. Twichell, and E. H. Welborn, “Hardware Mechanism for Instruction/Data Address Tracing,” U.S. Patent 5,446,876, August 1995.
20. R. E. Kessler, M. D. Hill, and D. A. Wood, “A Comparison of Trace-Sampling Techniques for Multi-Megabyte Caches,” *IEEE Trans. Computers* **43**, No. 6, 664–675 (June 1994).
21. E. N. Elnozahy, “Address Trace Compression Through Loop Detection and Reduction,” *Proceedings of the ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, May 1999, pp. 214–215.
22. J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler, “A NUCA Substrate for Flexible CMP Cache Sharing,” *Proceedings of the 19th Annual ACM International Conference on Supercomputing*, June 2005, pp. 31–40.
23. B. Fraguela, J. Renau, P. Feautrier, D. Padua, and J. Torrellas, “Programming the FlexRAM Parallel Intelligent Memory System,” *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 2003, pp. 49–60.
24. IBM Corporation, “alphaWorks: Emerging Technologies,” June 2005; see *http://www.alphaworks.ibm.com/*.

Received June 28, 2005; accepted for publication August 19, 2005; Internet publication March 3, 2006

James L. Peterson *IBM Research Division, Austin Research Laboratory, 11501 Burnet Road, Austin, Texas 78758 (petersjl@us.ibm.com).* Dr. Peterson is a Research Staff Member. He received a Ph.D. degree from Stanford University in 1974 and then joined the Department of Computer Sciences of the University of Texas at Austin. Dr. Peterson joined IBM in 1989.

Patrick J. Bohrer *IBM Research Division, Austin Research Laboratory, 11501 Burnet Road, Austin, Texas 78758 (pbohrer@us.ibm.com).* Mr. Bohrer is a Research Staff Member, and manages the Performance and Tools Department. He received a B.S. degree in computer science from the University of Texas at Austin. He joined IBM in 1994.

Lei Chen *IBM Research Division, Austin Research Laboratory, 11501 Burnet Road, Austin, Texas 78758 (chenl@us.ibm.com).* Dr. Chen is a member of the Performance and Tools team at the IBM Austin Research Laboratory. He received a Ph.D. degree from the University of Rochester. Dr. Chen joined IBM in 2004.

Elmootazbellah N. Elnozahy *IBM Systems and Technology Group, 11400 Burnet Road, Austin, Texas 78758 (mootaz@us.ibm.com).* Dr. Elnozahy received a Ph.D. degree in computer science from Rice University. From 1993 to 1997 he served as an Assistant Professor at Carnegie Mellon University. In 1997 he joined the IBM Austin Research Laboratory, where he managed the System Software Department until 2005. Dr. Elnozahy currently leads PERCS.

Ahmed Gheith *IBM Research Division, Austin Research Laboratory, 11501 Burnet Road, Austin, Texas 78758 (ahmedg@us.ibm.com).* Dr. Gheith is a Research Staff Member. He received a Ph.D. degree from Georgia Institute of Technology in 1990 and then joined the IBM Advanced Workstation Division. After pursuing interests outside IBM from 1996 to 2003, he returned to IBM and joined the Austin Research Laboratory.

Richard H. Jewell *IBM Systems and Technology Group, 3039 Cornwallis Road, Research Triangle Park, North Carolina 27709 (rhjewell@us.ibm.com).* Mr. Jewell is a Senior Software Engineer working on the system simulator for the Cell processor. He received a B.S.E.E. degree from the Christian Brothers University and an M.S.E.E. degree from the University of Kentucky.

Michael D. Kistler *IBM Research Division, Austin Research Laboratory, 11501 Burnet Road, Austin, Texas 78758 (mkistler@us.ibm.com).* Mr. Kistler received a B.S. degree in computer science from Susquehanna University and an M.S. degree in computer science from Syracuse University. He joined IBM in 1982 and is currently working on simulation technologies for IBM POWER and PowerPC processors and systems.

Theodore R. (Ted) Maeurer *IBM Systems and Technology Group, STI Design Center, 11501 Burnet Road, Austin, Texas 78758 (maeurer@us.ibm.com).* Mr. Maeurer is the manager of the software organization for the Austin-based STI Design Center, where he has been responsible for the development of software technologies for the Cell processor. He received B.S. and M.S. degrees in computer science from Rensselaer Polytechnic Institute, and an M.S. degree in engineering and management from the Massachusetts Institute of Technology.

Sean A. Malone *IBM Systems and Technology Group, 3039 Cornwallis Road, Research Triangle Park, North Carolina 27709 (seaxouri@us.ibm.com).* Mr. Malone received a B.S. degree in electrical engineering from the State University of New York at Buffalo. He joined IBM in 1997.

David B. Murrell *IBM Research Division, Austin Research Laboratory, 11501 Burnet Road, Austin, Texas 78758 (dmurrell@us.ibm.com).* Mr. Murrell received his degrees from Purdue University and has focused on simulation and performance analysis since 1989. He joined the Austin Research Laboratory in 2004.

Neena Needel *IBM Research Division, Austin Research Laboratory, 11501 Burnet Road, Austin, Texas 78758 (nneedel@us.ibm.com).* Ms. Needel is a member of the Performance and Tools team at the IBM Austin Research Laboratory, where she develops technical communications. She received a B.S. degree in communications from St. Edward's University in 1996. She joined IBM in 2004.

Karthick Rajamani *IBM Research Division, Austin Research Laboratory, 11501 Burnet Road, Austin, Texas 78758 (karthick@us.ibm.com).* Dr. Rajamani is a Research Staff Member in the Power-Aware Systems Department. He received a B.S. degree in electronics and communications engineering from the Indian Institute of Technology, Madras, and M.S. and Ph.D. degrees in electrical and computer engineering from Rice University. He joined IBM in 2001 and has since been engaged in computer systems research.

Mark A. Rinaldi *IBM Systems and Technology Group, 3039 Cornwallis Road, Research Triangle Park, North Carolina 27709 (nalds@us.ibm.com).* Mr. Rinaldi is a Senior Technical Staff Member in the xSeries* Engineering Software Department. He received B.S. and M.S. degrees in electrical engineering from Rensselaer Polytechnic Institute, joining IBM in 1974.

Richard O. Simpson *IBM Research Division, Austin Research Laboratory, 11501 Burnet Road, Austin, Texas 78758 (simpson@us.ibm.com).* Dr. Simpson has worked in computer architecture, operating systems, and simulation at various IBM laboratories since 1969. He received a Ph.D. degree from the University of Texas at Austin in 1988.

Kartik Sudeep *IBM Research Division, Austin Research Laboratory, 11501 Burnet Road, Austin, Texas 78758 (kartik@us.ibm.com).* Mr. Sudeep is a Staff Engineer working on designing performance tools for next-generation systems. He received an M.S. degree in electrical and computer engineering from Duke University in 2000.

Lixin Zhang *IBM Research Division, Austin Research Laboratory, 11501 Burnet Road, Austin, Texas 78758 (zhangl@us.ibm.com).* Dr. Zhang received a B.S. degree in computer science from Fudan University, China, and a Ph.D. degree in computer science from the University of Utah. He was a postdoctoral research associate at the University of Utah from 2001 to 2003, when he left to join the IBM Austin Research Laboratory.