Mambo – A Full System Simulator for the PowerPC Architecture

Patrick Bohrer Mootaz Elnozahy Ahmed Gheith Charles Lefurgy Tarun Nakra
James Peterson Ram Rajamony Ron Rockhold Hazim Shafi Rick Simpson
Evan Speight Kartik Sudeep Eric Van Hensbergen
Lixin Zhang

IBM Austin Research Lab
Austin, TX 78758
pbohrer@us.ibm.com

Abstract

Mambo is a full-system simulator for modeling PowerPC-based systems. It provides building blocks for creating simulators that range from purely functional to timing-accurate. Functional versions support fast emulation of individual PowerPC instructions and the devices necessary for executing operating systems. Timing-accurate versions add the ability to account for device timing delays, and support the modeling of the PowerPC processor microarchitecture. We describe our experience in implementing the simulator and its uses within IBM to model future systems, support early software development, and design new system software.

1 Introduction

Full system simulators have emerged during the past decade as viable tools for low-level system software development and performance evaluation. Earlier, our team adapted the SimOS simulator platform [8] to support the PowerPC architecture [5]. While our experience was successful, it also showed the need for an industry-strength implementation that is more configurable and amenable to the rigors of the software engineering life cycle. Therefore we started Mambo, a modular full system simulator that is designed from the ground up to simulate the PowerPC line of processors [6]. The implementation supports different simulation modes, ranging from functional simulation of the PowerPC instructions, to cycle-accurate simulation of an entire system. Mambo also includes trace collection and debugging interfaces to allow detailed analysis of the simulated hardware and software. Seven processors of the PowerPC line are supported, including the 32-bit embedded 405GP [7] and the 64-bit 970 PowerPC used in Apple's new G5 system [1]. The processor support includes interrupts, debugging controls, caches, busses, and a large number of architectural features. In addition, Mambo models memory-mapped I/O devices, consoles, disks, and networks that allow the simulated operating systems to boot and run programs.

To fill our needs, our design stresses *modularity* and *configurability*. Modularity is achieved by an internal structure that features a modern, multithreaded simulation core. This in turn is enhanced with various programming constructs that support a modular and highly maintainable design. The constructs implement higher-level abstractions to express the usual characteristics of simulated systems, such as pipelined execution units, and programmers use these abstractions to quickly model different system behaviors. Due to this modularity, our team is able to experiment with simulator enhancements and performance improvement, and quickly introduce them into the simulator with minimal perturbation to the production mode operation.

The second feature stressed in the Mambo design is configurability. Mambo is designed as a collection of configuration features that can be selected to easily define a variety of processors and devices. Compile time and runtime parameters allow users to configure nearly every feature of the system being simulated. Compile time options define major features (such as 32-bit or 64-bit support), while runtime options set fine-grained parameters such as amount of memory, number of processors, cache geometry, etc. A partial list of selectable features includes:

- 32-bit or 64-bit processor design.
- Floating point registers and instructions.
- Vector Multimedia Extension (VMX) registers and instructions.
- Hardware Multi-Threading (SMT) [12].
- PCI bus.
- IDE disks.
- Network.
- Caches (L1, L2, L3, and victim).
- Bus.
- Memory.
- UART and console support.
- Hypervisor support.
- Address translation (ERAT, SLB, TLB) [6].

- Uniprocessor or multiprocessor.

The simulator runs on the x86 and PowerPC platforms running a range of operating systems including Linux, AIX, OS/X, and Windows®. It uses Tcl/Tk to provide a command language and graphical user interface and DiskSim [3] to provide timing-accurate disk models.

We have used Mambo successfully for a variety of purposes, including support of operating system development, system bringup, characterization of application performance and power consumption, performance tuning, and pre-hardware application development. In Section 2 we describe our experience with Mambo in more detail. We then describe in Section 3 the implementation of the simulation and conclude the paper in Section 4.

2 Experience with Mambo

Like other full system simulators, Mambo has proved useful in software development and application characterization. In some cases, the simulator served as a platform to enable software development before the hardware is available. As an example, a team of researchers at IBM was able to develop the software for Blue Gene/L [11] [4] [2] so that when the hardware became available, programs were running on the first day, and the system was usable within a week. Similar uses are also underway for several architectures and systems under development.

It is noteworthy that Mambo is useful for software development even if the hardware is available. For example, developing low-level system software such as operating systems on the bare hardware is time consuming. Mambo includes an interface to gdb, allowing source-level debugging from the very first instruction of the operating system. gdb attaches to Mambo so that developers can use the normal gdb interface to debug the simulated operating system. The simulator can single step through code that cannot normally be traced in this way, such as an operating system's first level interrupt handler. A team of researchers at IBM has used the simulator to support the development of the K-42 operating system [10]. In their experience, the simulator has advanced their development schedule by about a year.

Mambo also can enhance the software-hardware co-design process. For example, new hardware features such as SMT or hypervisor support can be modeled and low-level system software can be developed to examine the use of such features before they are finalized into hardware. Our experience shows that this approach has several benefits that straddle software and hardware. For example, our experience shows that using Mambo early in the hardware design process to model the new feature forces the designers to define the feature well enough to be programmed. The feedback from the model implementation and the software experience with the feature may uncover errors, missing functionality or areas that were not well understood. Traditionally, such problems are not uncovered until a detailed VHDL model of the hardware is built, or even after system software has been implemented on the finalized hardware platform. For instance, in the early design of a PowerPC processor, Mambo revealed a race condition that required changing the semantics of several bits in a control register. Also, the hardware features of a hypervisor design had to be updated based on the implementation of the operating system on the modeled hypervisor.

The second category of using Mambo is in application characterization. Mambo produces a variety of statistics, both in summary and detailed form, allowing the performance and operation of a program to be understood and evaluated for a new hardware architecture. By associating performance-affecting hardware events (e.g., cache misses, TLB shoot downs, and memory references) with the program instruction stream, it is possible to identify under-performing portions of a program and correlate the performance problems with resource usage. This may allow significant performance improvement by changing a data structure or the position of an inner loop to reflect the cache architecture. These features provide an infrastructure for characterizing application and system behavior and performance.

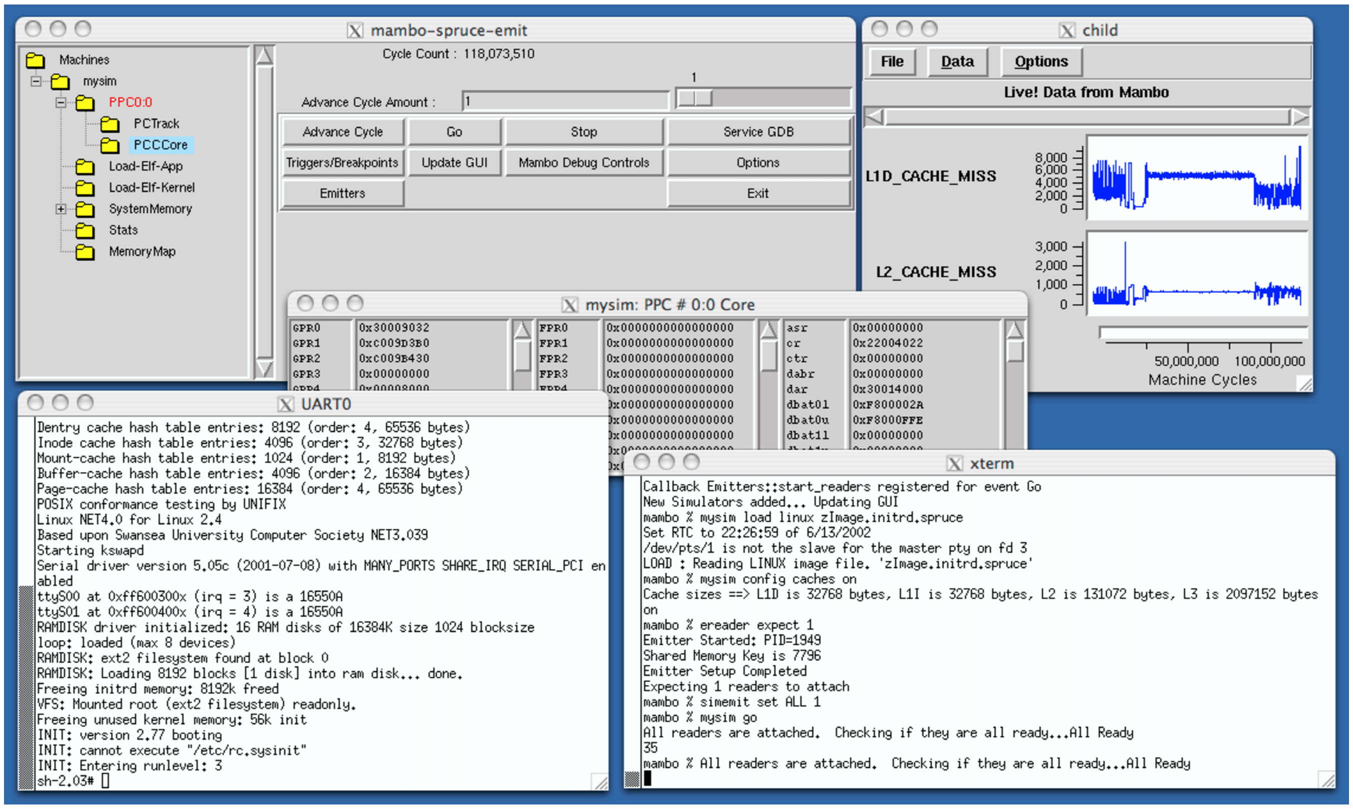
We have extended the characterization to the emerging field of power-aware computing [9]. With the help of power estimates for the various tasks associated with execution of instructions, an analysis of the total power consumed in the core and memory subsystem can be carried out. Then, one can use this information to identify opportunities for reducing processor speed (e.g., during memory-intensive instructions) or modifying the application structure to reduce power consumption.

3 Implementation

3.1 Operating System Adaptation

While Mambo is capable of booting unmodified operating systems such as Linux, detailed simulation of peripherals is time intensive to implement and slows down simulation. To improve run-time when detailed device simulation is not necessary, several changes are made to the simulated operating system to allow more direct interaction with Mambo. A direct block driver interface allows disk images on the simulation host to be used by the simulated operating system, and a virtual Ethernet interface is added that can either communicate to other simulated hosts or to real networks. Other changes include process tracking hooks, which interact with Mambo statistics gathering infrastructure.

Figure 1 shows a screenshot of Mambo booting Linux on a PowerPC 750 system. The UART0 window shows the simulated console and the xterm window shows the Mambo command line. Other windows show the GUI interface and a statistic gathering tool. The GUI ensures ease of use and



```
GPR0   0x30009032     FPR0   0x0000000000000000     asr      0x00000000
GPR1   0xC009D3B0     FPR1   0x0000000000000000     cr       0x22004022
GPR2   0xC009B430     FPR2   0x0000000000000000     ctr      0x00000000
GPR3   0x00000000     FPR3   0x0000000000000000     dabr     0x00000000
                                                    dar      0x30014000
                                                    dbat0l   0xF800002A
                                                    dbat0u   0xF8000FFE
                                                    dbat1l   0x00000000
```

```
Dentry cache hash table entries: 8192 (order: 4, 65536 bytes)
Inode cache hash table entries: 4096 (order: 3, 32768 bytes)
Mount-cache hash table entries: 1024 (order: 1, 8192 bytes)
Buffer-cache hash table entries: 4096 (order: 2, 16384 bytes)
Page-cache hash table entries: 16384 (order: 4, 65536 bytes)
POSIX conformance testing by UNIFIX
Linux NET4.0 for Linux 2.4
Based upon Swansea University Computer Society NET3.039
Starting kswapd
Serial driver version 5.05c (2001-07-08) with MANY_PORTS SHARE_IRQ SERIAL_PCI en
abled
ttyS00 at 0xff600300x (irq = 3) is a 16550A
ttyS01 at 0xff600400x (irq = 4) is a 16550A
RAMDISK driver initialized: 16 RAM disks of 16384K size 1024 blocksize
loop: loaded (max 8 devices)
RAMDISK: ext2 filesystem found at block 0
RAMDISK: Loading 8192 blocks [1 disk] into ram disk... done.
Freeing initrd memory: 8192k freed
VFS: Mounted root (ext2 filesystem) readonly.
Freeing unused kernel memory: 56k init
INIT: version 2.77 booting
INIT: cannot execute "/etc/rc.sysinit"
INIT: Entering runlevel: 3
sh-2.03# 
```

```
Callback Emitters::start_readers registered for event Go
New Simulators added... Updating GUI
mambo % mysim load linux zImage.initrd.spruce
Set RTC to 22:26:59 of 6/13/2002
/dev/pts/1 is not the slave for the master pty on fd 3
LOAD : Reading LINUX image file. 'zImage.initrd.spruce'
mambo % mysim config caches on
Cache sizes ==> L1D is 32768 bytes, L1I is 32768 bytes, L2 is 131072 bytes, L3 is 2097152 bytes
on
mambo % ereader expect 1
Emitter Started: PID=1949
Shared Memory Key is 7796
Emitter Setup Completed
Expecting 1 readers to attach
mambo % simemit set ALL 1
mambo % mysim go
All readers are attached.  Checking if they are all ready...All Ready
35
mambo % All readers are attached.  Checking if they are all ready...All Ready
```

Figure 1: Mambo Graphical User Interface during a Linux boot.

quick identification of performance bottlenecks.

3.2 Timing Models

Mambo provides a variety of timing models for software development and for hardware and software performance evaluation. The simplest timing model assumes each instruction requires one cycle to execute. Memory accesses are synchronous and instantaneous. This is a purely functional model, and is useful for software development and debugging when a precise measure of execution time is not important. Even in this mode, some system features require timing support. For example, I/O interrupts and timer interrupts are scheduled to provide at least a crude sense of the passing of time. These inaccuracies are tolerated given the intended use of the functional model. This use trades accuracy for increased processing speed. For instance, a functional model of the 405GP processor executing on a 3.2GHz, x86 system can simulate an average of 4 million PowerPC instructions per second.

For accurate performance evaluation, Mambo provides a cycle-accurate timing model. A cycle-accurate timing model requires a complete modeling of the operation of the processor including its pipeline and functional units. Each operation takes a number of cycles to complete and must consider both processing time (the time to search a cache, for example) and resource constraints (e.g., an instruction cannot be issued to an add unit if that add unit is already in use). This mode of operation provides good accuracy at the expense of longer simulation time. A cycle accurate model of the 405GP processor was validated to be within 0.6% of real hardware, but ran four times longer than the functional model, which was off by 26% against the real hardware [9]. For more complex processors, the slowdown of the cycle accurate model compared to the functional model can be 10 times or more.

A compromise between the fast, but inaccurate, functional model and the slower, but accurate, cycle-accurate model is the cycle-approximate model. This model uses probabilistic measures to improve timing estimates. For example, a memory reference may (or may not) hit in the cache. A cache hit takes a different amount of time than a cache miss. In the cycle-accurate model, it is necessary to model the cache, allowing Mambo to determine exactly if a particular reference is in the cache. The cycle-accurate model knows if there is a cache hit or miss. The cycle-approximate model does not model the cache (hence providing a faster simulation), but probabilistically determines the time for the access from user-supplied cache hit ratios as well as a predetermined time for a cache hit and cache miss. We are currently adding this model to the infrastructure.

3.3 Multithreaded Simulator Structure

To simplify the development effort while still accurately modeling hardware events, we structured Mambo as an internal thread programming model, allowing instruction execution code to simply pause in place (delay) as necessary. For instance, the main function of a cache refill request simply looks as follows:

```
Cache_Refill(MemAccessStruct *ma)
{
  DO_DELAY(cache_to_bus_delay);
  Pass_It_To_Bus(ma);
  DO_DELAY(bus_to_memory_controller_delay);
  Pass_It_To_Memory_Controllers(ma);
  DO_DELAY(memory_controller_to_dram_delay);
  Pass_It_To_Dram(ma);
  DO_DELAY(dram_delay);
  Pass_Back_To_Memory_Controller(ma);
  DO_DELAY(memory_controller_to_bus_delay);
  Pass_Back_To_Bus(ma);
  DO_DELAY(bus_to_cache_delay);
  Pass_Back_To_Cache(ma);
}
```

`DO_DELAY()` is a call to inform the thread scheduler that this thread needs to be delayed for the given number of cycles. Because Mambo's thread model allows simulators to be coded in the way that preserves hardware behavior, it can drastically reduce programming effort and the resulting code is very easy to understand.

Mambo's thread model is implemented completely at user level. Switching between Mambo threads is *almost* as efficient as switching between events, but should occur much less frequently compared to a pure event-driven simulator.

The thread programming model has also introduced a host of other programming constructs to express the constraints of, for example, a pipeline processor model or a multi-level cache. All abstractions are built on a small set of low-level (familiar) mechanisms, most notably, `gate`s, `counter`s, `aval`s, and `port`s.

A `gate` is used to express resource limitations by restricting concurrency. A gate is created with a given `width`. A thread can `enter` then `leave` a gate. The gate only allows `width` threads simultaneously (in simulated time).

A `counter` is used as an event signaling mechanism. Threads can `set`, `increment`, `decrement` the value of a counter. A thread can `block` for the counter to have a `zero` count.

An `aval` (active value) is used to implement constructs like register renaming. A thread can declare ownership of an active value. Any thread that attempts to access this value can provide a counter that gets decremented when the owner sets the final value.

A `port` is the preferred mechanism for forking concurrent activities. A port is associated with a handler function and a dynamic pool (fixed or variable size) of worker threads. Sending a message to a port schedules it for processing by a worker thread and returns asynchronously to the caller. Counters can be used to synchronize different aspects of the interaction between the caller and the worker.

Since simulation models always use those constructs to express their dynamic behavior and timing characteristics, we have the freedom to vary the implementation to achieve different objectives. Indeed, we are currently exploring different models of multi-threaded (using one way messages) and distributed (using backwards recovery techniques) implementations.

3.4 Performance Evaluation Infrastructure

The performance of a program on a PowerPC system, even one that does not yet exist, can be determined by running it on Mambo. By booting an operating system, such as Linux, the application can be executed and timed using standard timing tools running on the simulated system, *including operating system interactions*.

Alternatively, applications can be run in ”standalone” mode, where all operating system functions are supplied by Mambo, and normal OS effects, such as paging and scheduling do not occur. This provides information that is more directly a result of the intrinsic program design and implementation. This is useful for application's performance and power characterization.

Mambo also provides its own timing measures. In addition to simple cycle count information, and summary statistics, Mambo provides an ”emitter” data stream. To enhance modularity and usability, we decoupled the performance analysis toolset from the simulator implementation. Rather than build into Mambo a large set of performance analysis tools, Mambo has been designed to generate a stream of events. The specific events, such as instruction execution, memory reference addresses and contents, TLB hits and misses, cache hits and misses, and so on, are emitted into a circular queue in shared memory where they can be read by other programs, called ”emitter readers.”

The events that Mambo puts in the emitter data stream are selectable by the user at run-time. In addition, events deemed ”uninteresting” for a particular run or purpose can be ignored by emitter readers. Thus, a user can select a large set of events to be emitted and simultaneously run several emitter readers that process the emitter data stream looking for the events of interest to them. Emitter readers can compute summary information (range, average, standard deviation, and histogram of execution times), or can display the events in real time.

Another approach is to define an emitter reader that converts the Mambo emitter data stream (or some subset of it) to a pre-existing trace format. This new trace can then be fed into existing analysis tools. Multiple trace formats can be supported simply by writing new emitter readers, a relatively simple task. Mambo itself requires no changes for the additional formats.

For enhanced usability, performance analysis can be provided by the GUI based emitter readers. These provide graphs of memory access, cache misses, processor resource usage, and even power usage [9], displayed against time. Since the emitter stream includes the program counter, it is possible to trace interesting performance events, such as high cache miss rates, back to the specific instruction of the simulated program and even to the specific lines of source code.

4 Conclusions

Mambo is a full system simulator that has proved useful in supporting low-level system software development and characterizing applications' performance and power consumption. We have used the simulator successfully within IBM to support various projects, including the Blue Gene supercomputer and the K42 operating system, among several others. The simulator features a modern core based on multithreading and high-level abstractions that support a high degree of modularity, configurability, and ease of use. Users of the simulator report substantial reduction in development times, increased insight into the hardware design process, and successful characterization of application performance and power consumption.

While Mambo is not open source software, it is freely available through a special license to parties outside IBM on an as-is basis. At the time of this publication, it has been licensed to 8 companies and over 25 academic institutions.

Acknowledgements

This work was partially supported by the Defense Advanced Research Projects Agency, Department of Defense under contracts F33615-00-C-1736, F33615-03-C-4106, NBCH30390004, and NBCHC020056. Further support was provided by various IBM divisions. PowerPC is a trademark of IBM. X86 is a trademark of Intel Corp. Windows is a trademark of Microsoft Corp. We acknowledge all trademarks referenced herein to be the property of their owners.

References

[1] Apple Computer Inc. Apple Power Mac G5, 2004.

[2] L. R. Bachega, J. R. Brunheroto, L. DeRose, P. Mindlin, and J. E. Moreira. The BlueGene/L Pseudo Cycle-accurate Simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS),* March 2004.

[3] J. S. Bucy and G. R. Ganger. The disksim simulation environment version 3.0 reference manual. Technical Report CMU-CS-03-102, Carnegie Mellon University, 2003.

[4] L. Ceze, K. Strauss, G. Almasi, P. J. Bohrer, J. R. Brunheroto, C. Caşcaval, J. G. Castaños, D. Lieber, X. Martorell, J. E. Moreira, A. Sanomiya, and E. Schenfeld. Full Circle: Simulating Linux Clusters on Linux Clusters. In *Proceedings of the Fourth LCI International Conference on Linux Clusters: The HPC Revolution 2003*. Springer-Verlag, June 2003.

[5] IBM Austin Research Lab. SimOS-PowerPC web page. Available at http://www.research.ibm.com/arl/projects /SimOSppc.html, 2000.

[6] IBM Corporation. *The PowerPC Architecture: A Specification for a New Family of Processors*. Morgan Kaufmann Publishers, Inc., 1994.

[7] IBM Corporation. *PowerPC 405GP Embedded Processor User's Manual*. IBM Corporation, 2000.

[8] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete Computer Simulation: The SimOS Approach. In *IEEE Parallel and Distributed Technology,* Fall 1995.

[9] H. Shafi, P. Bohrer, J. Phelan, C. Rusu, and J. Peterson. Design and validation of a performance and power simulator for PowerPC systems. *IBM Journal of Research and Development*, 47(5/6):641–652, 2003.

[10] C. A. N. Soules, J. Appavoo, K. Hui, R. W. Wisniewski, D. Da Silva, G. R. Ganger, O. Krieger, M. Stumm, M. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis. System support for online reconfiguration. In *USENIX Annual Technical Conference,* pages 141–154, 2003.

[11] The BlueGene/L Team. An Overview of the BlueGene/L Supercomputer. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing,* Nov 2002.

[12] R. Thekkath and S. Eggers. The effectiveness of multiple hardware contexts. In *International Conference on Architectural Support for Programming Languages and Operating Systems,* 1994.