HARDWARE/SOFTWARE CODESIGN OF PROCESSORS: CONCEPTS AND EXAMPLES

JOHN HENNESSY
Stanford University and MIPS Technologies, Silicon Graphics, Inc.
AND
MARK HEINRICH
Stanford University

1. Introduction

Today, many VLSI designs are processors at the core. Microprocessors are one obvious example; however, other examples abound. Many special-purpose, embedded controllers consist of a microprocessor, at least at the core. Digital Signal Processors (DSPs) are special-purpose processors. Special-purpose engines for functions such as graphics and video or audio processing are essentially processors, some using microcode implemented in a ROM or PLA and others using code stored in a RAM. The key aspect of all these designs is that they require the development of both hardware and software to make a successful machine.

The movement towards RISC machines has increased the number of designs that can be implemented as processors. This has occurred for two reasons. First, by using RISC pipelining techniques it is possible to implement a programmed engine that performs competitively with a hardware design that is more specialized to the design requirements. Second, the availability of high-performance RISC cores has made it possible to build application-specific processors that achieve high performance with modest effort.

Because many VLSI designs are essentially processors, designing both the hardware and software for such systems is critical. Furthermore, completing the hardware and software on time, tuned to the needs of the application is often key to the success of the overall project. Although the hardware and software involved in a special-purpose processor may be less complicated than the those in a general-purpose processor, many of the design issues are the same. Thus, the challenges faced by complex general-purpose microprocessors striving to achieve new performance levels will shortly be faced in the design of application-specific engines.

2. Key Hardware and Software Issues in Processor Design

Irrespective of whether a processor design is intended for a general-purpose or special purpose application, there are two design requirements that are critical to success. These critical issues are the functional correctness of the design and the attainment of the performance targets for the design. Functional correctness depends on a detailed design of the machine and verifying that the processor implements this detailed design. For a definition to be unambiguous and to provide a basis for verification typically requires that the design definition be in a form that can be easily simulated. Design specifications that are written, but cannot be simulated, are simply too ambiguous and incomplete. Formal specifications are very useful for small portions of the design, but our present ability to construct and use formal specifications cannot accommodate full, large-scale designs.

Verifying that the performance of a design will attain the desired goals is a different, but equally complex, problem. To design a processor that meets its performance targets often requires evaluating many different design alternatives including both the hardware and software. The design tradeoffs can become quite complex, employing a variety of hardware or software mechanisms. Furthermore, real designs must accommodate a different set of constraints, such as power budget, limited silicon area, or pin count limitations. The challenge in many designs is to achieve the performance targets without compromising the limitations on power, silicon area, or pad counts.

2.1 DESIGN ELEMENTS OF PROCESSORS

Many of the key design aspects of processors fall into one of three areas: the instruction set, the pipeline organization, or the memory system [1]. These three aspects of the design are critical both from a correctness and a performance viewpoint.

The instruction set defines the primary boundary between hardware and software. As such, it is the highest level definition of the hardware and the basis for verifying the design. In addition, since it defines the boundary, tradeoffs between hardware and software implementation are made at this boundary. The difficulty in making such tradeoffs arises because the design space for the instruction set is very large, even if the basic instruction set is RISC-oriented (for example see Appendix E of [1]). To evaluate the instruction set choices, a compiler that can be easily retargeted to allow exploration of the value of certain tradeoffs is critical. It is unbelievable that some designers still will completely define and even implement an instruction set before they have constructed a compiler and examined the instruction set alternatives quantitatively. Although less common in general-purpose processor designs, this unscientific design approach is common in the special-purpose processor arena.

All modern processors, whether special purpose or general-purpose, make use of pipelining, since it is the key organizational technique for achieving high instruction throughput. The increasing complexity of the pipeline being used in modern processors has led to a situation where both the design and verification of the pipeline are extremely difficult. Furthermore, the design space for pipelines is extremely large and evaluating the alternatives is costly for three reasons: the effort required to create accurate performance models for the pipelines, the long run times of the simulation models used to evaluate performance, and the possible need to tune the compiler for each pipeline alternative.

All modern processors also make use of a memory hierarchy to supply the data required by the processor in a cost-effective manner. The potential design space for the memory hierarchy is also very large, since it can include both single and multilevel caches. Furthermore, the cache at each level in the hierarchy has a large variety of independent design parameters:

- the size of the cache,
- the degree of associatively,
- the block size (as well as possible subblocking),
- the write policy (e.g., write through or write back).

Fortunately, accurate and efficient simulation of memory systems is well understood and can even be made moderately fast (at least compared to pipeline simulation). The most challenging aspect of evaluating a particular memory hierarchy is often obtaining adequate traces to determine the performance. For modern processors with powerful memory hierarchies these traces must often be quite large (10’s to 100’s of millions of references) and need to reflect the complex operating environment (e.g., operating system or other large applications) that the processor will be used in. For more specialized application environments, obtaining accurate traces often requires building the application and the underlying software system to compile and simulate the application.

3. Design Verification and Performance Validation

Verifying the correctness of a processor as well as ensuring that performance goals are met requires the concurrent design of both the hardware and software for the system. Assuming the validation strategy uses simulation of the hardware, the software system must provide a complete and extensive stimulus for the simulation. In addition, to validate the performance, which is typically stated in some metric involving one or more benchmarks, we must have a software system capable of compiling the benchmarks so that hardware simulators can be used to accurately project the performance. Thus, the compiler system plays a critical role in providing stimulus both for functional simulation as well as for performance analysis of the system. The compiler system must be powerful enough to compile the key performance benchmarks and any necessary libraries or OS code, so that the benchmarks can be simulated adequately.

To be able to complete the hardware and software design within a time frame that is competitive, the hardware and software system must be designed concurrently. For the software, this leads to requirements such as the ability to retarget the compiler as instruction set changes are made. The requirement for concurrent engineering of the hardware and software has even more critical implications for the hardware and the hardware design methodology. In most hardware designs, much of the effort goes to decomposing the design and completely defining it as a series of interacting blocks. Furthermore, as the design proceeds, difficulties encountered at lower levels of the design hierarchy may force certain decisions to be changed or may have significant performance implications that were not visible at the higher levels of the design.

To address these issues and allow the design of the hardware and software to proceed at multiple levels, we must create a variety of models for the design. By building simulators for these models we can often obtain rough estimates of performance as well as develop the higher levels of the design. Because the higher levels of the design (and the accompanying simulations) are more abstract, they can be created earlier and typically simulated more efficiently. In addition, since they contain fewer details, the higher level models are easier to change allowing for more rapid exploration of design alternatives. Although such higher level models of the system may be missing details and may yield rough estimates of performance, these higher level models are critical to allow the design to proceed in a concurrent fashion and to allow the exploration of high level design tradeoffs before the effort is expended to complete the lower levels of the design. In the next two sections, we explore how these concepts were applied in the design of the MIPS R4000, a general purpose processor, and to the MAGIC chip, a special-purpose, programmable communications processor for use in multiprocessors.

4. An Example of Hardware/Software Codesign: the MIPS R4000

In this section we show how the design strategy of the MIPS R4000 [2] used concurrent engineering of the hardware and software to allow simultaneous design of hardware and software as well as to verify the design both functionally and from a performance viewpoint. The key to this process (and to the concurrent design and implementation of the software) is the use of multiple levels of simulation for both functional and performance verification.

Although a detailed design description of the R4000 is beyond the scope of this paper, a few aspects of the design are important to understanding the motivation for the design strategy that was employed. The R4000 is a general-purpose processor and its design goals included a variety of functional capabilities as well as performance goals. Among the key functional capabilities required for the R4000 are:

- the implementation of the MIPS-III instruction set architecture including 64-bit instructions,
- high performance measured using the SPEC benchmarks and other significant programs,
- a sophisticated memory hierarchy with an optional off-chip cache with a variety of "programmable" aspects, such as access time, transfer rate and block size,
- the implementation of a powerful coprocessor capable of handling memory management and other operating system support functions.

These design objectives had to be achieved within tight limitations on the silicon area, power consumption, and number of pins. Finally, the desired schedule allowed for only a small number of fabrication runs with the goal that the first silicon would be usable (possibly with software work-arounds) for debugging the chip, for testing and debugging complete systems using the R4000, and for completing operating systems development.

4.1 OVERALL SIMULATION STRATEGY

The strategy for the design and validation of the R4000 focuses on a set of simulators that model the system at increasing levels of detail. Five levels of simulation are employed in the design.

1. Instruction level simulator: this is used for performance evaluation at the instruction set level as well as for more detailed modeling of the pipeline and memory system. This level is also used to generate test vectors employed in lower-level simulators.
2. System level simulation: this simulator models the details of the system environment including such things as interrupts and memory management. It is used to verify the system-level aspects of the processor, for concurrent design and debugging of the operating system, and as a comparison basis for generating test vectors.
3. RTL level: this simulator is generated by compiling an RTL description of the design into C code. It probably consumes the most simulation cycles. It is used to simulate both portions (blocks) in the design as well as to simulate the entire design.
4. Switch level with delays: used to simulate the design mostly in sections; test vectors are generated from the RTL level.
5. Circuit simulation: Spice is used for detailed modeling of the critical paths as well as for verification of circuits under variations in temperature, power supply, etc.

In the rest of this paper, we focus on the top two levels in this design hierarchy, since they are most critical to the codesign of the hardware and software system. A key to this simulation strategy is that the top level simulators are considerably faster, allowing them to be used for more extensive performance and functional verification. Figure 1 shows the names given to each simulator (for those that can be used to simulate the entire design) and the wide variation in performance of these simulation models. The performance is given in number of clock cycles simulated per second on a 30-mips machine used for simulation.

Simulator	Level of Accuracy	Simulation Rate
Pixie	User instruction set	> 10^6 cycles/second
Sable	System level (OS instructions + interrupts)	> 10^3 cycles/second
RTL	Synchronous register transfer	> 10 cycles/second
Gate	Gate/switch level	< 1 cycle/second

Figure 1. Levels of simulation used for the entire R4000 design and the performance of each level.

4.2 INSTRUCTION LEVEL SIMULATION OF THE PROCESSOR

Using an appropriate instruction set simulator, we can analyze the performance of a processor at three levels: the instruction set effectiveness, the pipeline performance, and the memory system performance. In the design system used for the R4000, the basis for all three of these evaluations is a system called *pixie*. Pixie translates an object-code file for the MIPS architecture into an instrumented version of the object-code (called a pixified program). When the instrumented object-code file is run it can produce both profile information (for each basic block in the program) and a trace of the instruction and data references. Figure 2 shows how pixie instruments a code sequence and uses the instrumented binary to produce information.

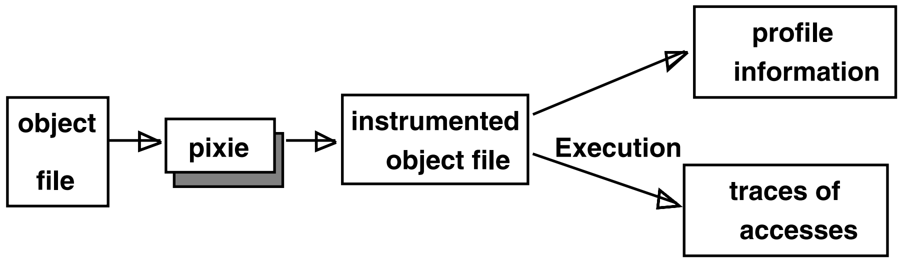

Figure 2. Pixie instrumentation process translates an object code file into an instrumented executable. When run, the executable produces both profile information and memory reference traces.

There are two key advantages to this approach:

1. The pixified program runs very fast with a typical slowdown of 2 to 10 times. This makes it possible to make many runs of this simulator.

2. Pixie translates any system calls in the pixified program into calls to the underlying OS running on the machine that executes the pixified program. The enormous advantage of this approach is that full programs can be run and evaluated even if they require OS facilities. This eliminates a major drawback of many simulation systems that can handle only kernels or small programs. This advantage allows the analysis of large benchmarks, such as the SPECmark programs.

Once the profile information has been produced from the pixified program, an analysis program, called *pixstats*, can analyze both the instruction set performance and the pipeline performance of a machine. Instruction set performance is analyzed by using the profile counts and the instruction makeup of each basic block to compute instruction counts, frequencies, and other instruction set usage properties. By simulating the pipeline behavior of each basic block and using the profile information to determine how often each basic block is executed, the performance of the pipeline can also be estimated. This approach is extremely cost-effective since each basic block is simulated only once. Figure 3 shows illustrates this process.

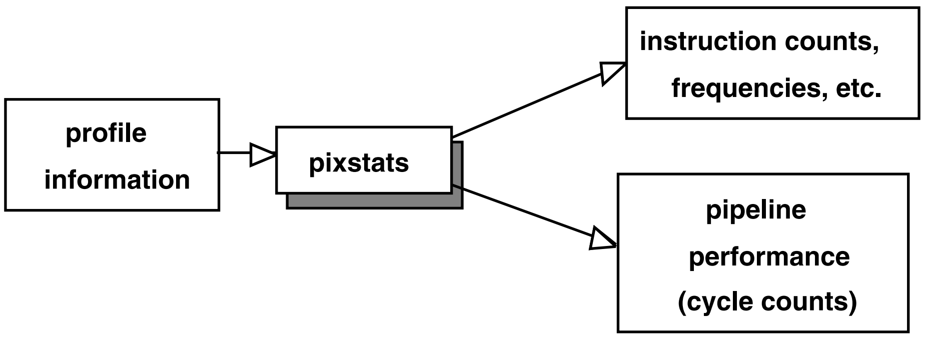
profile information
pixstats
instruction counts, frequencies, etc.
pipeline performance (cycle counts)

Figure 3. Using output from a pixified executable to estimate performance. The pixstats program contains a model of the processor that may be partially parameterized, making comparison of design changes very fast.

Pixie can also instrument a program so that the pixified program produces traces of instruction and data references. These traces can be used to evaluate the performance of the memory system using one of several cache system simulators, such a Dinero or the R4000 cache simulator (cache4000). Figure 4 shows how the memory system performance is analyzed. The overall processor performance can be determined by combining the measurements of the memory system and the pipeline. Thus, we have a fast and accurate method for evaluating the performance of the system at the level of cycle counts.

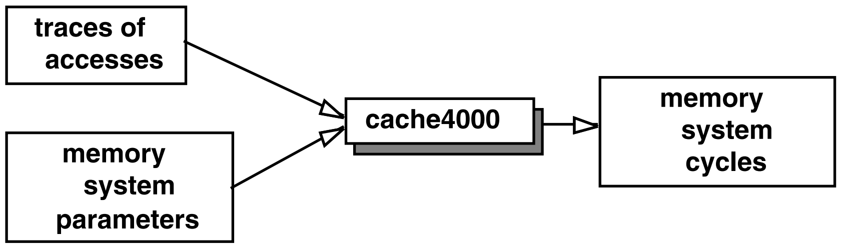

Figure 4. Evaluating memory system performance by using the traces from the pixified program.

Finally, the pixie approach allows the construction, debugging, and performance analysis of compiler systems long before hardware is available. Instruction-level simulators often do not have sufficient performance for testing and debugging compilers, since large programs that really stress compilers may run fairly slowly; pixie, however, is sufficiently fast. Thus, besides its other benefits, the pixie approach allows concurrent design and implementation of the processor and compiler system. Since processor performance depends on the compiler technology, this approach is highly beneficial.

4.3 SYSTEM LEVEL SIMULATION

Sable provides a simulation environment that implements most of the system aspects of a processor, including memory management, system instructions, internal and external exceptions, and an I/O system. Sable has two important uses:

1. It provides an environment for writing and debugging the operating system concurrently with the design of the hardware. Sable can also be used to analyze the performance of operating system mechanisms.
2. Sable provides a way to test the system aspects of the processor. Since these aspects include many of the asynchronous components of the system, it is particularly susceptible to bugs. By running the operating system and the system-oriented tests, Sable provides a way to debug these aspects of the system, as well as to generate test vectors that can be used in simulators provided at lower levels (such as RTL).

Finally, Sable is also used as the basis for testing the multiprocessor aspects of the system.

4.4 FUNCTIONAL VALIDATION STRATEGY

The complete functional validation strategy relies heavily on this hierarchy of simulators. The key steps in validating the design are:

1. Selected user programs are run at the pixie and Sable levels, as well as used to

generate RTL test vectors.

2. Randomized user programs are generated, simulated, and used to generate RTL test vectors.
3. The OS is run in the Sable environment using randomized interrupts to test the system aspects of the design and generate RTL test vectors.
4. The ultimate test is to boot the OS on the RTL model; this requires weeks of simulation time.

Overall, the functional validation of the R4000 required about 100 30-mips machines in continuous use for about 200 days, for a total of 50 x 10^{15} cycles of simulation time. Figure 5 shows the history of bugs during validation. Despite this enormous investment, a small number of significant bugs remained. These bugs did not prevent further debugging or the use of the part in prototypes; however, several key bugs needed to be repaired before the part could go into production.

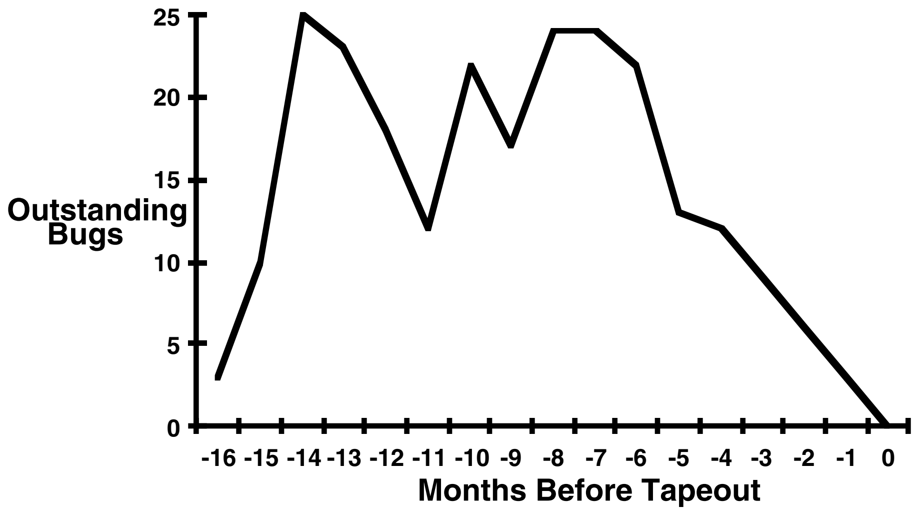

Figure 5. Number of bugs discovered as tapeout approaches.

4.5 PERFORMANCE VALIDATION STRATEGY

At the level of user programs, the pixie approach combined with pixstats and a cache simulator meant that *billions* of instructions (and memory references) could be simulated to evaluate the instruction set, the pipeline, and the memory system.

The performance of critical OS routines, such as TLB miss handling and interrupt vectoring, was evaluated with Sable. By comparing the cycle count differences among the memory system simulator, Sable, and the RTL simulator, the designers could ensure consistency of the cycle counts for key operations (such as a cache miss) across the levels of the design. Since such costs are important input parameters for the high level simulators (e.g., the memory simulator), it is critical to verify the assumed values against the actual costs that can be derived from lower-level simulators.

The simulators discussed above provide very accurate analysis of the cycle counts (as well as instruction counts) for both user and system programs (including applications or benchmarks). To completely determine the processor performance, we also need to validate that the processor will attain the desired clock rate. Clock cycle verification for the R4000 was done primarily by using Spice on the critical paths. A design methodology was used to expose such paths, together with limited use of static timing analysis.

5. MAGIC: An Application-Specific, Programmable Processor

MAGIC is the communication controller for the Stanford FLASH multiprocessor [3]. FLASH is a scalable, *convergence architecture*, meaning that it will support both shared memory and message passing programming with underlying hardware support. This support is implemented in the MAGIC chip, which thus becomes extremely performance critical. The basic architectural structure of FLASH is shown in Figure 6.

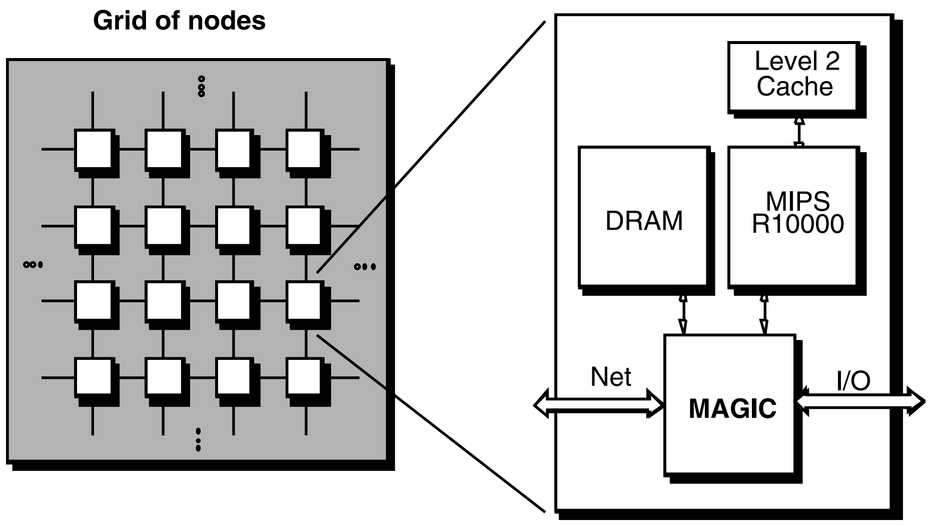

Figure 6. Stanford FLASH architecture showing the critical position of the MAGIC chip through which all data for the processor must flow.

The core of the FLASH design is MAGIC, a programmable communications and protocol engine, implemented as a semicustom ASIC. MAGIC integrates the interfaces to the main processor, memory system, I/O system, and network with a programmable, special-purpose protocol processor responsible for running the communication protocol. The combination of multiple interfaces, aggressive macro-pipelining, and the necessity of handling multiple outstanding requests from the main processor makes MAGIC inherently multithreaded, posing complex verification and performance challenges.

5.1 MAGIC ARCHITECTURE

A high-level view of the MAGIC architecture is shown in Figure 7. The key architectural feature of MAGIC is the separation of the data-transfer logic from the control logic. The data-transfer logic is implemented in hardware to achieve highly-pipelined, low latency and high bandwidth data transfers and avoid extra data copying on chip. To manage the complexity of shared memory and messaging protocols, the control logic is handled by software code (called *handlers*) running on the protocol processor.

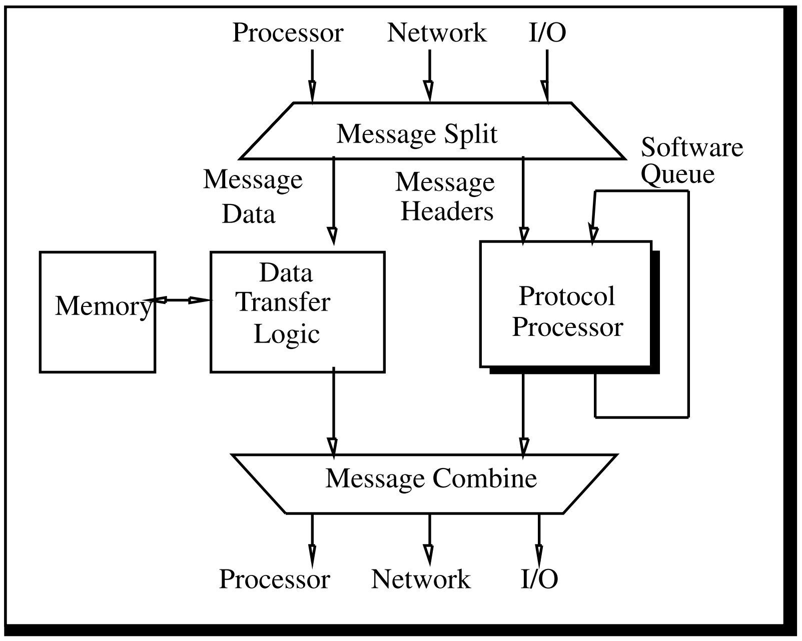

Figure 7. MAGIC Architecture showing how the programmable protocol processor handles the interpretation of messages, while data transfer and the input and output functions are handled in hardware. Separate parallel state machines are used to manage these independent functions.

5.2 MAGIC'S PROTOCOL PROCESSOR

In addition to helping with the complexity and correctness of protocols, the use of a programmable protocol processor increases flexibility. This flexibility allows different protocols and communication paradigms, the implementation of novel performance monitoring schemes, and the ability to debug or enhance protocols even after the machine is built. To minimize the performance impact of flexibility, MAGIC incorporates several features to make protocol processing more efficient.

First, the protocol processor is a simple two-issue RISC machine with no hardware interlocks, no floating-point capability, no TLB or virtual memory management, and no interrupts or exceptions. Second, the instruction set is expanded to include instructions for handling bit field operations common to directory-based and messaging protocols [4]. The handlers that are run on the protocol processor are generated from C using a port of gcc and a superscalar instruction scheduler. The compilation tools are aware of the bit field instruction set optimizations and take full advantage of them. Finally, to avoid consuming excessive memory bandwidth and to reduce protocol processor occupancy and handler latency, the protocol processor accesses all code and data through dedicated instruction and data caches, respectively.

When a request enters MAGIC and is selected for service, it is pushed through a programmable table which dispatches the starting program counter for the protocol processor. The dispatch table can also be programmed to initiate a speculative memory operation on behalf of the request to reduce latency as much as possible. The use of hardware dispatch further improves protocol processor occupancy since software dispatch is a particularly costly operation, and handler dispatch and execution can now operate concurrently in a macro-pipeline.

5.3 SIMULATION AND VERIFICATION CHALLENGES

Simulation of the MAGIC design is particularly difficult because of the high degree of interaction among design elements. Tuning the protocol processor instruction set depends on what operations it needs to do and how fast it needs to do them; determining how fast it needs to do them requires accurate simulation of a FLASH system, which depends on knowing how long protocol processor operations take.

The accurate simulation of FLASH is itself an enormous problem. The simulations are highly concurrent (potentially 100's of processors in a FLASH simulation), resulting in large memory requirements and the need to properly model contention within and among nodes to obtain accurate performance information. This detailed modelling together with the desire to simulate the machine under stress (large applications with much activity in the PP) compounds the problem by slowing down the speed of simulation.

The high degree of concurrency results in an enormous state space, complicating the verification of MAGIC as well. Not only do we need to verify all possible protocol processor code sequences, but we also need to verify the interaction between the protocol processor and the other on-chip interface units.

5.4 THREE PART SIMULATION APPROACH

To attack these simulation and verification challenges, we adopt a three part simulation approach. We start with FlashLite, our system-level simulator written in C and C++, and successively refine our approach by increasing the detail of simulation as the MAGIC design itself proceeds. FlashLite is an execution-driven system simulator which uses the Tango Lite system as a reference generator [5]. Most of the program is compiled and executed directly, resulting in reasonably fast simulation (10-100x slow down). FlashLite uses the execution-driven approach for simulation accuracy, since we want the timing of simulation to be affected by both the program and the timing of the simulated memory system.

At the top level, FlashLite models high-level protocol processor operations such as sending or receiving messages and updating directory structures. The other units on MAGIC are modelled in enough detail to implement and debug the base cache-coherence protocol. This environment, shown in Figure 8, can also be used to collect statistics about the frequency of operations done in the protocol processor and the frequency of certain protocol cases.

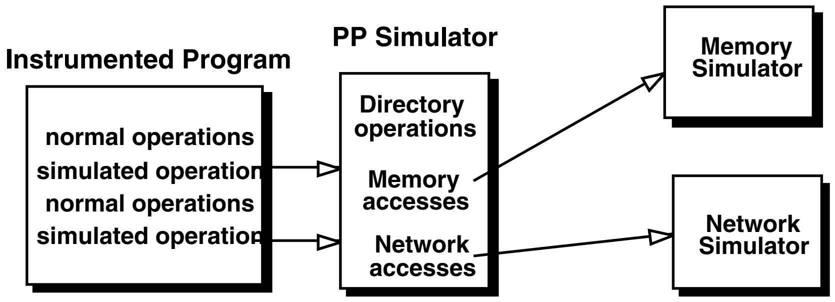

Figure 8. FlashLite Simulator: Stage I-customized simulator of the protocol processor (PP).

The next stage of refinement replaces the high-level model of the protocol processor with a simulator thread that emulates the actual compiled protocol code and produces accurate delays. This model, shown in Figure 9, allows full debugging of the protocol code (and the protocol processor compilation tools!) and can find timing-related bugs in the protocol that modelling at the higher level could not catch. In addition, it leads to detailed timing estimates for protocol processor operations and provides statistics about the caching behavior of the protocol code sequences.

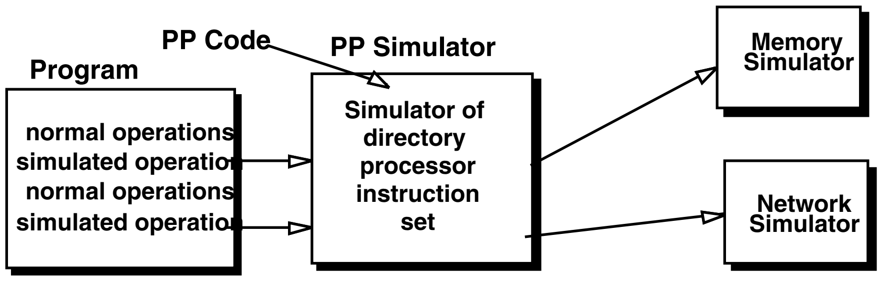

Figure 9. FlashLite Simulator: Stage II uses real protocol code and a custom instruction set simulator for the protocol processor.

The third and final level of simulation puts everything together, fully simulating the interface between the main processor and MAGIC, including the cache control of the main processor and its ability to issue multiple outstanding requests to the memory system. With its modelling of the complex timing on the external interfaces (processor bus, memory system, and network), and cycle accurate protocol processor simulation, FlashLite can now provide accurate information on overall application performance.

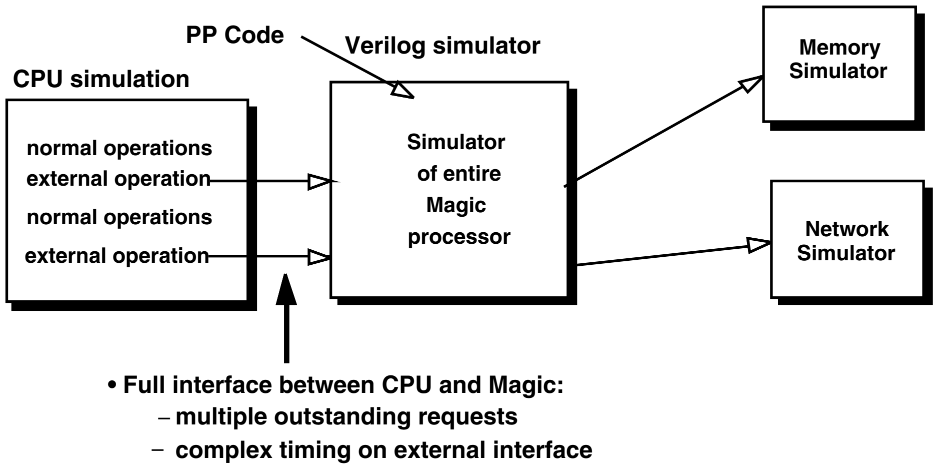
PP Code
Verilog simulator
CPU simulation
normal operations
external operation
normal operations
external operation
Simulator of entire Magic processor
Memory Simulator
Network Simulator
• Full interface between CPU and Magic:
– multiple outstanding requests
– complex timing on external interface

Figure 10. FlashLite Simulator: Stage III uses a simulator in Verilog for the Magic processor including interface simulators for the R10000 processor, the memory system, and the interconnection network.

5.5 ADVANTAGES OF THIS APPROACH

The most significant advantage of this approach is the benefit of concurrent engineering. We were able to design the protocol processor, the communication protocols, and the rest of FLASH in parallel. Having a high-level functional simulator proved invaluable in the early design stages of MAGIC, particularly in providing early performance feedback. FlashLite gave us good estimates of protocol performance before the architectural design was completed, and gave us input into the design of the protocol processor instruction set. Early simulations were also useful in fixing global resource allocation problems that might have led to MAGIC-induced deadlock.

Another often observed benefit is that our multiple levels of simulators act as documentation and provide test generation for lower levels. In fact, the reference generator portion of FlashLite has the capability to drive simulations of either the FlashLite description of the MAGIC chip or the Verilog RTL description of the chip. This allows us to debug the hardware using real parallel applications. A similar environment allows individual unit designers to run directed or random tests of their unit.

Finally, the MAGIC design approach balances hardware effort with software effort. The hardwired data-transfer logic and RISC core with support for bit field operations provide hardware performance, while the complex protocols are implemented in software.

6. Concluding Remarks and Future Directions

By using a hierarchy of design descriptions and simulations it is possible to proceed with the hardware and software design simultaneously. This is critical to achieving an improvement in the design time for the complete system. In addition, unless the design of the hardware and software can be done concurrently, it is essentially impossible to verify either the functionality or the performance of the design.

A hierarchical simulation structure also allows the designer to trade-off the cost of simulation versus the accuracy. Thus, extensive simulations can be done at the highest level to estimate performance and quickly iterate and tune the design. Furthermore, large numbers of cycles can be simulated at the higher levels and the resultant test vectors used at lower levels of the design, which are much more accurate. Overall, the approach dramatically improves the possibility of designing a hardware and software system that is both functionally correct and achieves the desired performance while maintaining a short design time.

Despite the significant advantages of this approach the complexity of a modern processor is enormous and it is critical that the design be close to perfect before tape-out. The challenge of managing this complexity, ensuring correctness, and still meeting a reasonable design schedule remains the fundamental hurdle in modern processor design. For future designs, tools that help manage the design complexity, including the complex interface between hardware and software, will be crucial. Just as important will be experienced designers that understand the relationship between design complexity, performance, and design time.

7. References

1. Hennessy, J.L. and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufman, San Francisco, 1996.
2. Killian, E, "MIPS R4000 Technical Overview–64 Bits/100 MHz or Bust," *Hot Chips III Symposium Record (August)*, Stanford University, 1.6-1.19, 1992.
3. Kuskin, Jeffrey et al., "The Stanford FLASH Multiprocessor", In *Proceedings of the 21st International Symposium on Computer Architecture*, pp. 302-313, April 1994.
4. Heinrich, Mark et al., "The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor", In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 274-285, October 1994.
5. Goldschmidt, Stephen, *Simulation of Multiprocessors: Accuracy and Performance*, Ph.D. Thesis, Stanford University, June 1993.