Modeling and Evaluation of Hardware/Software Designs

Neal K. Tibrewala, JoAnn M. Paul, and Donald E. Thomas
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213 USA
+1 412 268-3545
{zyrain,jpaul,thomas}@ece.cmu.edu

ABSTRACT

We introduce the foundation of a system modeling environment targeted at capturing the *anticipated interactions* of hardware and software behaviors — not just their co-execution. Key to our approach is the separation of external and internal design testbenches. We use a frequency interleaved scheduling foundation ideally suited to our approach because it allows unrestricted hardware and software modeling, a mix of untimed and timed software, and a layered approach using software schedulers and protocols to resolve software to resource time budgets. We illustrate our approach by discussing how architectural corner cases that arise due to interacting hardware and software behaviors can be a meaningful digital modeling concept. In addition to characterizing the response of a system when viewed as a black box, we characterize the *response of the design* to anticipated design changes. We include examples and simulation results.

Keywords

Hardware/Software Codesign, Computer System Modeling and Simulation, Digital System Design

1. Introduction

As more computer systems are being integrated into system-on-chip (SoC) designs and as the interactions of concurrent software programs with multiprocessing and distributed platforms becomes more complex, computer system designers must reason about: platform design for programmability, the modeling impacts of software schedulers, the co-execution of hardware-like and software-like system-level behaviors, and the system-level performance impacts of hardware architectures that execute software functionality. Many of the critical *system-level* design decisions are those that involve the *anticipation* of hardware/ software interactions; as hardware is loaded with software functionality, software is deployed onto a variety of hardware resources (architectures) and parts of a system (mixed hardware and software) must interact with the yet-to-be designed rest of the system (which may also include mixed hardware and software).

Codesign for system level modeling has been limited by the view that all computation should be distilled to reactive models — mathematical models of computation unified by event or token-based foundations. The resulting executable specifications are designed to respond to testbench-style inputs that model the external environment in which the system is intended to operate. The presumptions are that the computer system being designed is passive and it should be isolated from its operating environment.

Increasingly, the operating environment of a computer system is another computer system. Accordingly, next-generation computer system modeling must be based not upon the reaction of a passive computer to its operating environment, but upon active co-operation and co-ordination — sharing — across model boundaries such as resources. Computer system designers must be able to capture the sharing effects or *anticipated interactions* of concurrent software executing on multiple hardware resources over a range of design variations. More than understanding the *response of the system*, this is understanding the *response of the design.*

This paper introduces a new design methodology that allows a designer to control (vary), directly and independently, computation and communication power (modeled as rates), computation and communication loading (modeled as normal behaviors or data-dependent interference), internal performance limits (such as hard and soft time-outs) and offsets (which model non-determinism between clock domains). Essential to capturing resource sharing is the ability to vary hardware resources independent of their software loading. Our simulation semantic, *frequency interleaving* (FI), provides this ability by uniting hard-timed modeling with untimed (self-timed) modeling, and uniting data-dependent resource sharing with data-independent resource modeling at multiple levels.

Our results show how *corner cases* arise due to the effects of critical hardware/software interactions. For example, if performance degrades to unacceptable levels or resources become saturated, then any resulting bottleneck(s) may dominate the system model. Searching a complex design space for designs that satisfy performance criteria can be thought of as isolating and analyzing *prevalent performance models* that arise between corner cases in a design space. To fully analyze a computer system, designers must isolate these prevalent performance models and the range(s) over which they are valid. A designer can then understand the effects of software loading, resource variations, and resource sharing.

2. Modeling Environment

Our computer system design methodology introduces pure software models into a simulation environment, allowing for reasoning about the *whole* entity being designed — without restrictions to the software or hardware virtual machines. By viewing software modeling as a hardware resource sharing paradigm, we enable the designer to manipulate the system design space where hardware and software interact. As shown in Figure 1, a textual specification captures the model of the system as composed of software models, software scheduler/protocol models, and resource models. Distinguishing features include:

- The modeled system (middle of the diagram) is *layered*. Our layering approach allows software to be unrestricted (not lim-

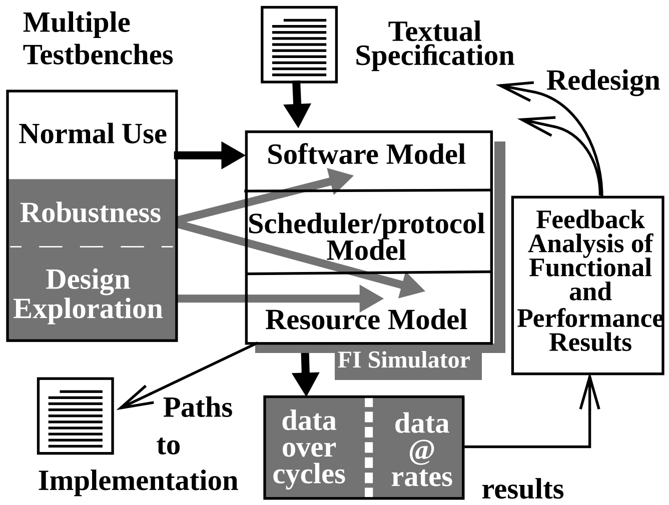

Figure 1 Computer System Design Methodology

ited to finite models of computation), allows schedulers and protocols to be included separately in the model, and enables novel means of resolving software execution to time (see later discussion on *frequency interleaving* (FI)).

- The simulation output (bottom of the figure) is viewed as mixed *rate-based* data and data that is captured over a *number of simulation cycles*. Key to our approach is that we capture *strongly timed and untimed (self-timed)* [3] software models. Real time (and structural modeling [5]) can be reasoned about in the same modeling environment with soft-time and performance-only models.
- The simulation *testbench* (TB) is split into three parts (left of figure). By separating the testbench for *normal operation* of the system from that used to *explore the design*, we provide the foundation for isolation of *anticipated* hardware/software interactions. *Robustness* captures software loading (top arrow) and resource loss or severe degradation (lower arrow). *Design exploration* varies resource models as with conceptual *tuning knobs* — adjusting computation and communication rates.
- The robustness and design exploration testbenches are shown *penetrating the system model*. By viewing parts of the system design as *incomplete*, we support an abstract level of design where hardware/software interactions are modeled as *anticipatory* behaviors — resources to be supplied for software, or software to load resources.

Our approach of modeling three independent layers allows the design methodology to support, directly, the manipulation of the resource models, the scheduler models, and the software models. Thus, any, or all, of these can contain the designable parameters while others can contain fixed or anticipated hardware or software. From this high level, a designer can converge on a point in the design space satisfying performance constraints. While some current high-level system design methodologies ignore the modeling of the hardware [6], this is clearly inadequate for system-level design. Other approaches limit software to hardware-like finite models of computation [11][12] or apply synthesis to a restricted portion of the design space [7][8][9]. Still others resolve all modeling to token-based encapsulated computation and communication [13][14][15], or gate-like discrete event scheduling [10][16]. None of these approaches allow the designer to reason about the resulting interactions and system-level characterizations of independently manipulable models of resource power, resource loading, and resource sharing.

3. Foundation: Frequency Interleaving

The modeling vision of Figure 1 is enabled by the simulation foundation, *frequency interleaving* (FI) [1][2]. Most other system-level models, most hardware-only models, and many software-only process models distill to graph-like encapsulated computation and communication, which does not directly allow for modeling the anticipated way that software will share resources. The "bubble-and-arrow" diagrams that result from graph-based models require discrete firings for information exchange [14]. By contrast, the primary concern in modeling most computer systems has become the modeling of computation and communication resources along with their sharing. These models require timed and self-timed behaviors to co-execute, and thus a flexible, comprehensive treatment of time that does not require the *strong* coupling of software to hardware (resource) time. These are all key in allowing the modeling environment of Figure 1 to become a reality — and are enabled in FI because its shared memory multi-rate, multi-threaded foundation allows independent manipulation of resource and software models.

3.1 Thread Relationships

All computation in FI is modeled from two fundamental thread types — C and G(F). The hardware foundation of system modeling (the Resource Model of Figure 1) is based upon interleaved execution rates — type C threads, for *continuous* execution. These are the only threads in the system with guaranteed, independent activation. Most significantly, they *continuously* execute based only on fixed *rates* or frequencies (f_i) — regardless of any other type of data events such as changes on inputs. Each C thread atomically samples and generates corresponding static, fixed size data sets (tuples) in a shared memory (state) space. Overlap is established by the threads' input/output shared memory.

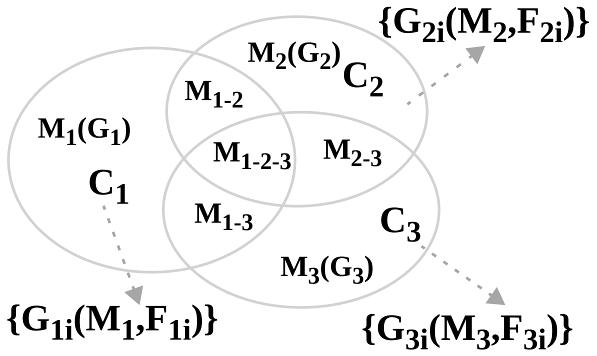

Figure 2 FI Thread Relationships

When resources are shared, software schedulers and communications protocols on networks multiplex access to the resources throughout the system in a data-dependent manner. This creates a conceptual *self-timed layer* that logically resides on top of the Resource Model layer of Figure 1. Accordingly, C threads *may* also act as a resource-like foundation for more complex forms of scheduling — G(F) threads, which have *guarded functional* execution, and state-dependent activation properties. Unlike C threads, G(F) threads have no *guaranteed* activation properties, but activate as a function of scheduler and data dependencies, need not execute atomically, operate on conceptually unbounded state, have flexible forms of time resolution, and can be thought of as behaving non-deterministically with respect to variances in the resource layer. Thus, G(F) threads are used to model the Software and Scheduler/protocol layers of Figure 1. While rate-based modeling environments [17] could conceptually be used to support the activation of FI C threads, they do not support G(F) threads. Figure 2 shows, for a system with three C threads, how G(F) threads relate to C threads, each other, and system state. Sets of G(F) threads are shown as $\{G_{ni}(M_n,F_{ni})\}$ where the *n* subscript denotes a unique C thread and the *i* subscript denotes a unique G(F) thread within a set of G(F) threads mapped to the same clock domain (C thread). The state upon which a set of G(F) threads operate is M_n, the size of which is a function of system execution as for unbounded software memory — allocation/de-allocation is possible in the shared memory foundation. Each G(F) thread within a set is activated by a function $F_{ni,}$ which is, itself, another G(F) thread within the set grouping, or a C thread. Each G(F) thread within a set must ultimately resolve back (possibly through other G(F) threads) to a foundation C thread, or the thread will never activate, shown by the dotted line emanating from each C thread on the diagram.

3.2 Resolving G(F) threads to C threads

In FI, resource models (C threads) provide time budgets that can be manipulated independently of the software models that consume them. *Conceptual* O/S (operating system) threads arbitrate access by one or more G(F) threads and account for their individual consumption of the time budget. The system designer chooses the *loading expense* of G(F) functionality. Because time budgets are themselves partitioned, different sets of G(F) threads may execute within a C thread's time budget in a highly data-dependent manner with very coarse to very fine-grained control. When a G(F) thread calls back to the scheduler at pre-determined points (which will eventually be inserted by a custom language), the scheduler makes a decision whether to continue running the thread, suspend the thread and run another thread, or suspend the thread and return control to the frequency interleaved scheduler. The latter option will end the atomicity of the C thread's execution, allow control to pass to another resource (C thread), and allow simulation time to progress.

For example, a designer may determine that each call to a particular function, such as read(), represents 50% of the resource time budget of the C-thread, allowing the remaining time to be consumed by other (parts of) G(F) threads until the C thread's time budget is consumed and control is returned to the FI resource scheduler to execute the next resource (C thread). Alternatively, a more expensive system call, memory access (via implicit callback), or library call may cost 400% of the C thread's budget. For this, the scheduler must partition the *over-budget* G(F) thread so that it relinquishes control to the simulator 4 times in order to obtain enough time to process the full call. The scheduler determines if the partitions are executed in succession or if the partitions are functionally interleaved [4] with other G(F) threads which may become eligible to execute in the meantime.

Thus OS/scheduler threads can be used to model a wide range of scenarios. They could be a simple round-robin scheduler, a more complex scheduler made to mimic the actions of a real OS or RTOS, or they could schedule the execution of an instruction-cycle accurate model — a very low level of modeling. They can represent high-level models of scheduling or portions of a design that remain in the completed system.

4. Digital System Design Corner Cases

One view of digital systems is that their models can be thought of as *invariant* over the operating space — the model of the system does not change when certain inputs or ranges of inputs are applied. This is analogous to an electrical or mechanical (analog) system for which a single set of linear equations applies for all possible operating conditions — a single linear model completely characterizes the input-output response of the system. It is convenient to think of digital systems as invariant when it is possible to use a single model to characterize the state of every register, memory location, and wire in a digital system. The system responds to sets of input data applied in succession from reset, but the system model does not change in response to the input data.

In contrast to such specification-driven modeling techniques, *prevalent models* are inferred from of a complex set of interactions. Complex systems containing software, software schedulers, communication protocols, and multi-resource interactions have made it more appropriate to consider digital system models as *variant* with respect to system inputs. These systems can be viewed as having ranges of inputs that not only exercise the system, they also modify its response, i.e. its prevalent model.

Programmability is an example of this. A program can be viewed as changing the model of the system for given data sets. Again using an analogy to electrical or mechanical systems, multiple prevalent models may be necessary to completely characterize the system, where each model may be appropriate for only portions of the range of operation. Corner cases for such systems result when there is a transition from one prevalent model to another. For a purely functional software system, the prevalent models of the system might be considered to change in such a manner as different functions are invoked — function calls can be considered *corner cases*. This singularly powerful aspect of software modeling — that of viewing the data input as being separate from the program input used to construct arbitrary models of the system — has made it possible for software systems to be characterized by a richer set of models than pure hardware systems. The challenge in digital system modeling is to provide for a similar view of systems with mixed software-programmability and hardware-resource modeling so that system designers may more effectively manipulate the design space without restricting it.

As digital systems grow in complexity, it is more useful to consider digital systems as variant with respect to *mixed models of function and architecture*. As mentioned previously, functionality is one way a pure software system model can be viewed as variant. But equally important for real-time (hard-time) and performance driven systems is the interaction of functionality with the computation platform.

Consider the system of Figure 3. It consists of four conceptual busses, or *bus domains*, labeled A-D, defined by a shared memory space, labeled M1-M4, which is equally accessible by any of the conceptual processors (P1-P6) or hardware devices (H1-H3) on the bus. Conceptual networks (N1, N2) interconnect the bus domains. The upper levels of our system model (Figure 1) are not shown on the purely structural diagram of Figure 3. These include the software schedulers implied by the multiprocessors, the protocols implied by the busses and networks, and the functionality mapped to the schedulers (or individual processors). All of these imply sharing of or across resources. Also not shown is how the schedulers may/not be tied to system clocks for real-time or self-timed behavior, and if the bus models are synchronous or asynchronous. The intended system response, i.e. the conditions under which the system is intended to operate is also missing. However, the prevalent system model is clearly defined by the interaction of all of these aspects of the design.

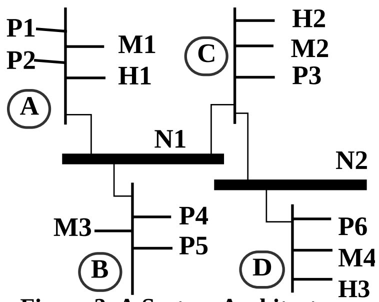

Figure 3 A System Architecture

For example, when the model shown in Figure 3 is considered invariant, the network, N2, is required to resolve communications from bus domain C to bus domain D, under all circumstances. However, when N2 is sufficiently fast *under the intended operating conditions* to resolve all data exchange between the functionality and schedulers mapped to bus domain C and the functionality and schedulers mapped to bus domain D, the network N2 may be considered to be transparent to the execution of the system — it is not a part of the model of the system so long as those conditions are met. Establishing the speed of the network where it no longer is transparent amounts to finding the corner case between two prevalent models — where N2 is/not transparent.

Corner cases can be inferred or intentionally established by the designer. Indeed, some high-level system designs are appropriately modeled and tested as a set of corner cases. This makes the design highly resistant to variations and highly predictable over a wide range or ranges of operating conditions.

5. Experiments and Results

Our modeling environment provides a basis for exploring digital system designs that contain interacting hardware and software models at a variety of levels of modeling detail, much like hardware description languages are a general model of digital hardware for a variety of levels of modeling detail [5]. No single design will illustrate all of its capabilities. The example in this paper is focused on a client/server application with multiple clients running on multiple CPUs requesting data from a single server on a single shared Ethernet link because it is rich in demonstration possibilities for high-level hardware-software interactions unique to our modeling environment. Such interactions include simulation-based computation and communication resource models, and software loading of resources varied independently as timed and self-timed software models.

As illustrated in Figure 4, the resources (C-threads) modeled in the system are 5 CPUs, 1 Ethernet, and 1 global clock reference. The clock reference provides a timebase for re-transmission of dropped packets, required in TCP/IP. The network model includes the effects of the network interface cards (NICs) on each CPU, and the actual packet exchange. Each CPU is running a copy of our conceptual OS — the G(F) thread that resolves software time to the time budget provided by its underlying C thread and which also schedules the software layer G(F) threads which are mapped to it. Initially, we have 2 clients running on each of 4 CPUs, and the server application running on its own CPU. The clients are competing against each other for both the single shared network link out of the CPU and for shared access to the CPU for execution time. A background task (BG) represents anticipated software loading which can be varied as the model is explored. The server dynamically spawns a new thread to handle each connection. All of the active server threads will be competing for their shared CPU as well as the network.

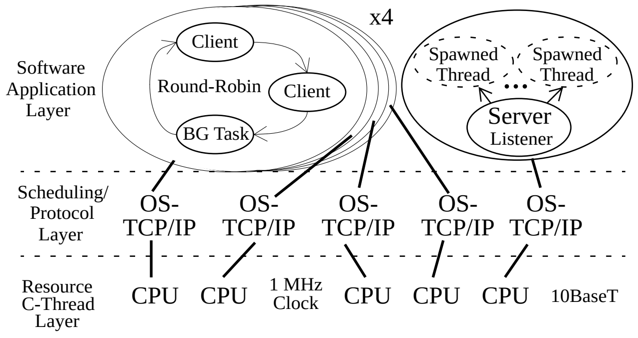

Figure 4 Thread Relationships of System Example

5.1 Packet Accurate Network

We initially created a high-level, packet-accurate model of the network. A variable amount of time is required to send a packet through an Ethernet network; smaller packets take less time to transmit than larger ones. Models with less detail can utilize average, random, or worst-case response times. We chose to do a worst-case simulation, with each packet transmission requiring 1.5ms, in a “packet accurate” cycle-based model, corresponding to the worst-case frame (packet) transmission time on 10BaseT Ethernet. The designable parameters in this initial model are the speeds of the processors and the size of the buffers on the network cards. The fixed parameters are the network speed (corresponding to 10BaseT — 1500 µs), a fast server (set to 100µs), and a single-packet buffer in each NIC.

Our first experiment, shown in Figure 5, determined the required speed of the CPUs for maximum utilization of the system resources, while minimizing the transfer time. The design exploration testbench was used to vary the power of the client CPUs. In this example, this is represented as the period of time on the processor we would expect to use to process a single packet. Therefore, slower processors are on the right, as they take a longer period of time to process a packet. The range varied from a fast 0.1ms to a slower 1.7ms time in 0.1ms increments. The same normal use testbench was used for each data point. This testbench had 8 clients simultaneously request a 10KB file from the server. Other scenarios are, of course, possible.

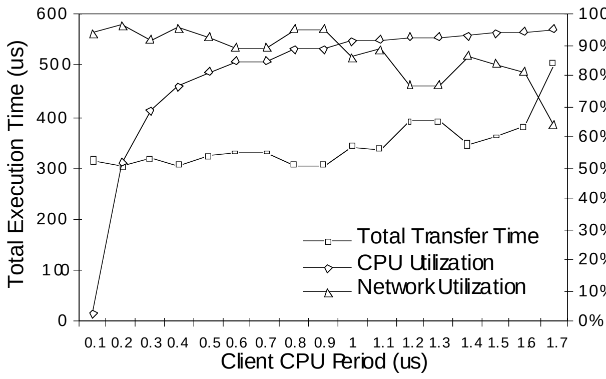

Figure 5 Design Response as a Function of CPU Period

Three metrics were used: CPU utilization, network utilization, and total execution time. The total transfer time line corresponds to the elapsed time on the system clock when the last client finished its transfer. CPU utilization is measured by the progress of the background task. A large amount of completed work by the background task means the CPU did not spend much time on network packet processing. The background task’s interference on the benchmark is minimized in this example by allowing the clients to continue in the simulation as soon as resources allow them to become unblocked. This task could also represent increased interference. The last metric, network utilization, is the percentage of time the network spent transferring packets instead of being idle.

The graph shows that as the client CPU period increases from 0.1ms (i.e., a decrease in frequency, or power), the total transfer time and network utilization remain fairly constant, while the CPU utilization increases. This is the range where the network is the bottleneck. The first corner case exists at roughly 0.9ms. This is where the CPU is just fast enough to handle packets as fast as the network can deliver them. At higher periods, the client CPUs are the bottleneck. The system, therefore, responds differently to additional decreases in CPU power; a different prevalent model dominates. At the corner case, the CPU and network utilizations are both high, indicating good use of available resources, and the benchmark is still completing in close to the fastest time possible. This tells the designer that a client CPU that can process packets in about 0.9ms is maximally efficient for the benchmark.

5.2 Byte Accurate Network

Multi-level modeling allows a designer to rewrite a portion of the model with more detail while leaving other parts at higher levels of detail. This is a strength of our modeling environment. In the previous example, our system was realizing only about 2 megabits of bandwidth (after overhead) on the 10 megabit network. A bench test on a real network would most likely yield closer to 7.5 megabits (after overhead). Several factors contribute, including the protocol model, the buffer sizes, and the network model. However, the most important contributing factor is that we were conservative in the network model. The high-level, packet-accurate model assumes the worst-case network response time. By providing more detail to the network resource model, a second, byte-accurate, model removes this worst-case assumption.

In the byte-accurate model, the client and server CPUs remain packet-accurate, but the packet buffer size on each NIC is increased to 20 packets. This corresponds, approximately, to a standard 16KB RAM buffer. This size is large enough that our benchmarks did not drop any packets due to buffer overflows, so we did not need to model, in detail, the TCP retransmission and congestion control algorithms. These could be included if we wanted to model congested networks as well.

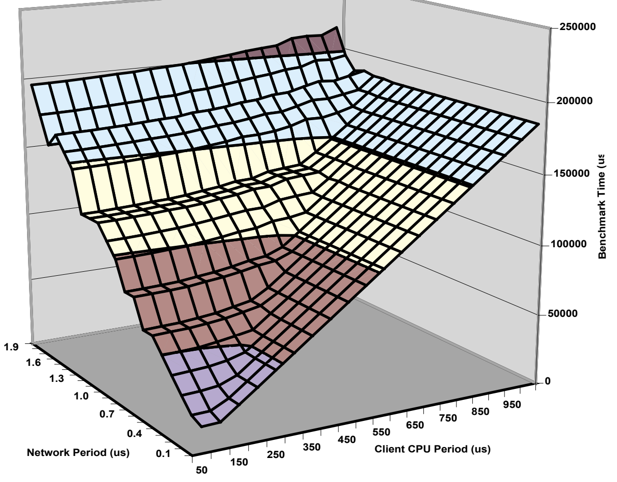

Figure 6 10KB Transfer — Byte Accurate Network Model

For this more detailed example, *both* the network and the client CPU speeds were varied. The client CPU speeds varied from 50µs per packet to 1000µs per packet in increments of 50µs. The network speed varied from 0.1µs per byte (80Mb/s) to 2.0µs per byte (4Mb/s) in increments of 0.1µs. The server CPU speed remained at 100µs per packet. Figure 6 clearly illustrates where the corner cases are. The smooth right half of the graph is the region where the CPU is the bottleneck. A decrease in network performance causes no degradation in benchmark speed. Conversely, the stepped left half of the graph is where network speed is the bottleneck. The ridges in this region represent decreasing processor speed without decreasing system performance. The corner case is where the smooth right region meets the stepped left region. The line is where the system is balanced and both the processor and network are fully utilized.

5.3 Cost Considerations

In the abstract, computation and communication speed can be varied together with no limits; the behavior shown in Figure 6 will extend ideally in the absence of additional constraints. However, physical and cost limitations can impose such constraints leading to additional corner cases. To illustrate this, we overlaid cost functions on the performance graph of Figure 6. The network cost factor penalizes high resource speeds (network periods) for a saturated resource or bandwidth allocation. Similarly, a simple CPU cost factor step function was generated by selecting the cost of the slowest CPU from a list of six that could process a packet in the period time. By increasing the penalty for exceeding the target, we amplify design effects so that corner cases are more easily seen. The factor below generates our performance/cost graph using the square of the ratio of performance to the target for a 100ms transfer time.

$$NetworkCostFactor * CPUCostFactor * (Performance/100000)^2$$

When we apply these costs to each data point along with the benefit or penalty for exceeding or missing the performance target we get the graph shown in Figure 7, illustrating the performance/cost ratio at each data point. This shows a much more intricate surface than the previous one, as there are now several equations contributing to the data. Now, it is no longer beneficial to have the fastest processor and network possible because of the increasing cost of each. The higher data points are in the center of the graph. Of course, this is expected because the previous corner case tells us that the center line is where there is a balance between network and client CPU utilization. One very illustrative region on the graph is the ridge just to the right of the three peaks. This region is one of the three where the cost function is favorable, but is too far off of the center line indicating balanced resources. It is a plateau, and not a peak, because it formed within the smooth region of Figure 6. Had this cost plateau crossed over the corner case seen in Figure 6, it would have resulted in a peak.

Table 1 Possible Implementation Points

Network Period (µs)	CPU Period (µs)	Performance (µs)	Network Cost Factor	CPU Cost Factor	Performance/ Cost Ratio
0.4	250	46749	2.5	11	.166
0.7	250	62499	1.4	11	.163
0.9	450	88199	1.1	7	.165

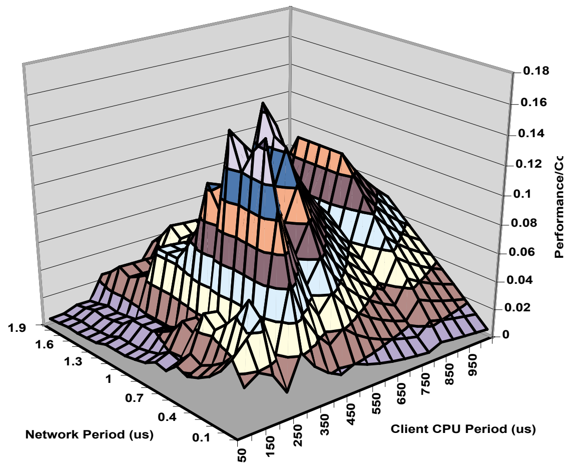

Figure 7 Cost Adjusted Graph - Byte Accurate 10K Test

The three peaks in the graph are detailed in Table 1, indicating where the Performance/Cost ratio is highest. Each point is at a different place on the network speed axis, while two of the points share the same place on the CPU speed axis. The first point (0.4,250) trades off cost for higher performance of the faster CPU and network. The last point (0.9,450) uses the drop in CPU price to justify the decreased performance, and the middle point is trying to strike the perfect balance between the cost of the resources and the benchmark performance.

5.4 Software Loading

In addition to varying resource power, our methodology allows us to understand the effects of variable and anticipated software loading of the resources. In this experiment we added a task to the *server* CPU which has the potential to interfere with the server's processing of the benchmark. The additional task loads the server CPU with an execution time ranging from 100 µs to 3200 µs as shown in Figure 8. We modeled this by changing the number of server CPU resource cycles (resource time budgets) required to process this task. The task has minimal effect on the benchmark time until the execution time reaches about 1200 µs, where the server CPU becomes 100% utilized (saturated). After that, increasingly slower performance is seen.

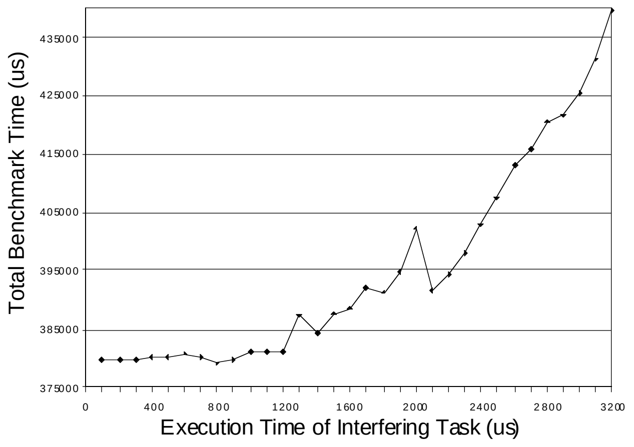

Figure 8 Performance vs. Server Loading (Interference)

5.5 Model Accuracy

To test how well our models capture real systems we ran a 100KB benchmark on a system consisting of 5 identical Pentium PCs connected via a 10BaseT Ethernet hub, measuring the completion time of each client. The simulation model has a mean client completion time of 644.0ms across 7 client threads, while the experiment showed an actual mean transfer time of 615.5ms. The 4.62% error is quite good considering the high level of modeling. For instance, Ethernet frame collisions are not being modeled. This is evident in the standard deviation of the test results. The model gives a standard deviation of 6ms compared to the real test value of 111ms. When a collision occurs, the Ethernet protocol causes the clients to use an exponential random back-off. This effectively reduces the contention on the network for some period of time, but causes a retransmission later. Of course, modeling the effects of frame collisions is something that can be included in a more detailed (lower-level) network model.

6. Conclusions

We introduced a computer system design methodology that allows a designer to model unrestricted software and hardware-resource models so that resource sharing effects can be explored for anticipated hardware/software interactions. Our methodology allows a designer to explore, directly and independently, design variations such as anticipated software loading, and computation and communication resource power, from a very abstract level through to detailed design. Critical performance limits and saturation effects conceptualized as digital system corner cases allow a designer to infer prevalent models, where a set of modeling variables dominate over an operating range. We base our scheduling foundation on frequency interleaving (FI) because it provides a multirate, multi-threaded foundation for varying resource power and resource loading, provides the basis for resolving strongly timed and self-timed behaviors, and allows for flexible means of resolving software back to hardware time. We include several sets of results for a networked model with clients, servers, software schedulers and an Ethernet hub model. Our modeling is not limited to network designs. Rather, it is aimed at modeling the general class of complex computing systems.

7. Acknowledgments

The authors would like to thank our other research team members: Henele Adams, Chris Andrews, Chris Eatedali, and Arne Suppé. This work was supported in part by the General Motors Satellite Research Lab at Carnegie Mellon University, ST Microelectronics, NSF Award EIA-9812939, and the Pittsburgh Digital Greenhouse through a grant from the Commonwealth of Pennsylvania, Department of Community and Economic Development.

8. References

[1] J.M. Paul, S.N. Peffers, D.E. Thomas. "A Codesign Virtual Machine for Hierarchical, Balanced Hardware/Software System Modeling," *37th DAC*, 2000.

[2] J.M. Paul, S.N. Peffers, D.E. Thomas. "Frequency Interleaving as a Codesign Scheduling Paradigm," *8th CODES*, 2000.

[3] C.L. Seitz. "System Timing." *Introduction to VLSI Systems.* C. Mead, L. Conway. Reading, MA: Addison-Wesley, 1980.

[4] D. Skillcorn and D. Talia. "Models and Languages for Parallel Computation," *ACM Computing Surveys.* June, 1998.

[5] D.E. Thomas and P.R. Moorby, *The Verilog Hardware Description Language, 4th Edition,* Boston: Kluwer, 1998.

[6] B. Selic. "Turning Clockwise: Using UML in the Real-Time Domain," *Comm. of the ACM,* pp. 46-54. Oct. 1999.

[7] D. Gajski, F. Vahid, S. Narayan, J. Gong. "SpecSyn: An Environment Supporting the Specify-Explore-Refine Paradigm for Hardware/Software System Design," *IEEE Trans. VLSI,* '98.

[8] Y. Li, W. Wolf, "Hardware/Software Co-Synthesis with Memory Hierarchies," *Proc. of ICCAD98,* pp. 430-436. 1998.

[9] R. Ortega, G. Borriello. "Communication Synthesis for Distributed Embedded Systems," *ICCAD98*. pp. 437-453. '98.

[10] J.-M. Daveau, G. Marchioro, A. A. Jerraya. Hardware/Software Co-design of an ATM Network Interface Card: a Case Study. *Proceedings of CODES* 1998.

[11] F. Balarin, M Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, et.al, *Hardware-Software Co-design of Embedded Systems. The Polis Approach. Boston: Kluwer.* 1997.

[12] L. Lavagno, E. Sentovich. "ECL: A Specification Environment for System-Level Design," *36th DAC,* 1999.

[13] J. Davis II, M. Goel, C. Hylands, B. Kienhuis, E. Lee, et. al, "Overview of the Ptolemy Project," ERL Technical Report UCB/ERL No. M99/37, Dept. EECS, Berkeley. July 1999.

[14] W.-T. Chang, S.-H. Ha, and E. A. Lee, ``Heterogeneous Simulation -- Mixing Discrete-Event Models with Dataflow," *Journal on VLSI Signal Processing*. Vol. 13, No. 1, Jan 1997.

[15] http://www.inmet.com/sldl/

[16] http://www.systemc.org/

[17] http://www.mathworks.com