The Design Context of Concurrent Computation Systems

JoAnn M. Paul, Christopher M. Eatedali, Donald E. Thomas

Electrical and Computer Engineering Department

Carnegie Mellon University

Pittsburgh, PA 15213 USA

{jpaul, eatedali, thomas} @ ece.cmu.edu

Abstract

Design for performance-optimization of programmable, semi-custom SoCs requires the ability to model and optimize the behavior of the system as a whole. Neither the hardware-testbench style nor the software-benchmark style is adequate to capture completely the design interactions required in concurrent software-on-hardware systems. We use a formal relationship between a computer system design content and its external context to motivate the need to consider a more effective modeling framework to which concurrent software-on-hardware computer systems are designed.

Keywords

Hardware/Software Codesign, Modeling, Simulation, Concurrent Computation, Digital System Design

1. Introduction

Next generation SoCs will be semi-custom platform designs that must be both programmable and performance-effective across a range of anticipated applications. The platform will include concurrency in the software and hardware and the software scheduling will involve both time and data-multiplexed resource sharing. These designs will only be as good as the model of the *context* to which they are being designed and evaluated.

Traditionally, two dominant means of design evaluation exist. The first is a hardware-testbench style used for both pure hardware and reactive systems in which inputs are presented at physically timed intervals and outputs are evaluated for coupled logical and physical correctness. The second is a software-benchmark style in which a set of representative programs — a benchmark suite — executes on a physical platform or its model. By measuring the start-to-finish execution of the benchmark suite, relative cross-platform performance is determined.

Neither the hardware-testbench style nor the software-benchmark style is adequate to capture completely the design interactions required in concurrent software-on-hardware systems. The logical and physical sequencing of hardware-testbench systems is too tightly coupled in both the model of the design and its external context. The software-benchmark style does not allow the system formed by software executing on hardware to be evaluated for sustained interaction with another system.

The downfall of these approaches, when used for evaluating concurrent systems, is that they model and analyze portions of a system in isolation. SoC designs will consist of a programmable computer system interacting with another, where each system could contain concurrent software and hardware — for the SoC as a whole, as well as for portions of its design. At high levels of design, one computer system can be considered the design, and the other can be considered its *context* — our term for a view beyond testbenches and benchmarks. Modeling the overall performance of these systems requires consideration of a system's ability not merely to react to, but to *interact with* its context. Modeling must be based on a holistic view that captures effective interactions between partitions, not on isolation and reaction.

Novel ways of conceptualizing the design space of programmable, concurrent digital systems are required at high levels of design — beyond a box-style component. For instance, in SoCs, design elements are more sharable than for other multi-computer designs — the penalty for interconnect is significantly less between portions of a chip as compared to portions of designs interconnected between chips. This affects not only the way a system is designed and evaluated, but also partitioned. A design's environment is strongly related to its basis for partitioning; reactive approaches define partitions *a priori*.

System models that formally advocate one or more models of computation while only indirectly accommodating other key design concepts that designers require are inadequate. Key design concepts, such as shared state and layering, must be acknowledged as part of the performance design space that designers manipulate. Such concepts lead to a larger view of how systems are partitioned, optimized, and extended by additional programming or physical hardware. We should not be looking for ways to restrict the SoC design space. With this paper, we illustrate the restrictions when only hardware-testbench or software-benchmark styles are used, motivating the need to capture the interactions between concurrent software-on-hardware systems and their context.

2. Modeling Computer Systems

We begin by summarizing our formal foundation [1] which models systems using logical and physical sequencing of events. The definition of an event model as a pair of data and time values is not new and the notions that digital systems include both logical and physical sequencing [8] as well as partially and totally ordered sequences [9] are both well established. We adopt the nomenclature in [12] referring to a time value in an event tuple as a tag. We observe that the relationship between the logical and physical events of a system model imply the type of design. We then motivate the need to consider a programmable concurrent system as a layered model of logical-on-physical sequencing and show how interactions among concurrent elements must be captured.

An *event* in a system model has a tag and a value $e = (t, v)$. The *value* represents an operand $v \in V$, the set of all operands in the system, which is the result of a calculation. The *tag* indicates a point in a sequence of events in which the operand is calculated.

Threads are an ordered set of N events,

$$Th = \{e_1, \ldots , e_N\}$$

where the ordering is specified by the tags of the events and N may be considered infinite. Thus event $e_i < e_j$ iff $T(e_i) < T(e_j)$, where

$T(e_x)$ represents the tag of event e_x.

In addition to a specific logical or physical ordering of tags, there are separate *data precedence* constraints to consider in a thread. These often arise from sequential language specifications where the resultant operand from line i of the specification is used in another calculation on line j, where $i < j$. That is, making the single change of moving line i in the language specification to be after j would make the results of line j's computation invalid. Thus a basic assumption is that reordering the events of a thread (i.e., reordering the time tags) is allowable as long as the data precedences are not violated.

2.1 Logical and Physical Ordering

Computer systems contain two kinds of event ordering — logical and physical [8]. The tags used in *logical ordering* specify a sequence which is not physically based. There is no physical meaning to the interval between any two events; we only know that one precedes the other or that the tags are the same. Logical ordering often arises from functional language specifications at a high level of design. The tags used in *physical ordering* represent a real time basis, establishing a physical basis for the system.

Separately, logical and physical event ordering can be characterized by the maximum total amount of state that is advanced by any event, the maximum complexity of the functional state advancement between any two events, and the number of events that can be considered to occur simultaneously or at the same time tag. The latter allows for a determination of the number of functions that can be considered to execute simultaneously in the system. The ordered sequence

$$Th = \{e_1, \dots , e_N\}$$

is ordered based on the tags. Clearly, a physically ordered system is totally ordered. A *partially ordered* system has at least two logical tags t and t' for which we do not know if t < t' or t' < t. Thus, assuming events e_a and e_b are partially ordered, one mapping to a physical order is the sequence

$$Th = \{\dots, e_a, e_b, \dots\},$$

and another correct mapping of events is

$$Th = \{\dots, e_b, e_a, \dots\}.$$

It is also possible that the two events are concurrent and have the same tag. Thus a key reason for describing a system with a partially ordered sequence is to allow greater flexibility in the design of the system; partially ordered events give rise to alternate, and potentially concurrent, implementations of the system.

Computer system models potentially contain two distinct types of thread sequences, Th_L and Th_P, which are sequenced to logical and physical time bases respectively. Designers do not design event sequences, rather they design to models that generate them. An implementation of a system results in a resolution of the logical events to physical events. We discuss how implicit assumptions on the means of resolution impact the implementation.

2.2 Concurrent Hardware/Software Systems

For the effective design of concurrent systems, interactions between Th_L and Th_P must be explored prior to partitioning into components [12] or separating design concerns [11]. We first consider the design implications when logical and physical sequences are *independent*. When logical sequencing alone is used to sequence a system — software design — the system will have unknown physical execution times between the events in Th_L. The key to this view is the assumption that the *design* of the logical ordering of the system (writing the software), is not significantly affected by the actual physical system (the processor, or system architecture) upon which it will ultimately execute. This is the case

because the software and hardware are largely independently optimizable. Function and architecture can be separated. Clearly there must be some assumptions on the existence of a physical machine to execute the software [10], but the actual physical execution performance of the software is not part of the system model. The system is literally formed at runtime, when the software is deployed on the platform by a scheduler. At this time, the logical events are resolved to physical events.

When logical and physical sequence are *identical*, a hardware design scenario results. While it is possible to assign a time base to logically sequenced systems by assuming a fixed execution order and time on partially ordered events for resolution to a common time base [12], these approaches strongly couple a logical sequence to a physical time base [6][13]. The event sequences Th_L and Th_P are identical because they are both sequenced by a physical time base. While they give insight into system correctness, they do not give insight into the *design* of systems in which the interactions of logical and physical sequencing have a great impact on performance.

For instance, assume that the thread sequence,

$$Th_L = \{e_1, \dots , e_i ,\dots\}$$

is a high level model. Because the events represent a relatively large amount of functional advancement, we term them *macro states* or *macro events*. These macro events can be decomposed into several states or events which have relatively less functional advancement — we term these *micro states* or *events*. If the macro states are totally ordered, they allow for substitution on micro event sequences, allowing the sequence to be re-written as

$$Th_L = \{(e_{11}, e_{12}, \dots , e_{1i} ,\dots), (e_{21}, e_{22}, \dots , e_{2i} ,\dots), \dots\}$$

Thus, each macro event, e_i, is seen to *contain* a sequence of micro events, $e_{i1}, e_{i2}, \dots e_{ik}$, where all the micro events of macro event e_i must complete before any of macro event e_j can execute (where i<j). This physical decomposition ultimately results in simple functions, such as gates. In this component-based, hardware design view, all logical state advancement may eventually be substituted 1:1 by a physical sequence Th_P, the physical time tags of the micro events may be combined to form a physical macro event tag, and the substitutions can be made without affecting the sequencing of higher-level events nor events in other branches of the hierarchy.

Finally, we observe that neither independent specification of sequences (software design), nor component-like containment captures the way logical and physical sequencing are related in concurrent hardware and software systems. Consider such a logical sequence of macro states

$$Th_L = \{e_1, e_2, \dots, e_i ,\dots\}.$$

Typically, concurrent software events are partially ordered. Further, the micro states implied by events e_1 and e_2 may be *interleaved* with each other on shared resources by a scheduler. That is, schedulers serve to resolve logical-on-physical sequencing — they arbitrate both logical (data-multiplexed) and physical (time-multiplexed) interleaving of a resource.

As illustrated below, the actual execution of the micro event sequences are no longer substitutable and atomic, but interleaved.

$$Th_L = \{e_{21}, e_{11}, e_{12}, e_{j1}, e_{22}, \dots \}$$

For instance, e_{j1} above might be a hardware network event that makes data ready based upon a number of factors. The time at which e_{j1} occurs is dependent on other data dependent dynamic software and e_j's time base. Indeed, this is only one ordering of the system's events as the true order depends on the data dependent and dynamic unbounded software, the scheduling method of the scheduler, and the shared resources (processors, busses, and

networks).

Clearly in the performance modeling of concurrent systems, the function and architecture are not independently optimizable as in a single processor software system. This is easily seen when considering the effects of adding either physical or logical concurrency to a computer system — the new system may have better or poorer performance. The key point is that in a truly concurrent software system, logical and physical sequencing are related — not identical as in system-level components or independent as when concerns are orthogonalized. Approaches that assign time delays to system-level software [13], as does a behavioral HDL, do not capture logical-*on*-physical sequencing.

3. Model Relationships

In this section, we formally illustrate the fundamental modeling differences between a hardware-testbench style of design and a software-benchmark style of design, starting from a general model of an I/O system. We observe that not only must the content of the concurrent software system be modeled as a logical-on-physical layering, but so also must the context within which it is being designed. We formally describe the design interactions in computer systems that must be facilitated by models, simulators and methodologies that capture the design space at hand.

3.1 Physical Testbenches

We begin with a very general model of an input/output system S [9] as illustrated in Figure 1 as:

$S = (T, I, O, \Omega, Q, \Delta, \Lambda)$,
where

T is a time base

Figure 1   I/O System

I is a set of input values
O is a set of output values
Ω is a set of input segments, or allowable (value, tag) pairs
Q is the set of states
Δ is the global state transition function
Λ is the output function

As a very general model, the I/O system can be used to model designable entities, like digital computers, or things we can largely only observe, like the weather. Thus, as a general model of a physical entity, an I/O system can be used to model a system being designed, as well as the environment with which it interacts as in Figure 2. A physical model of the computer system being designed, or the computer system content, S_N, can be related to the model of the system's environmental context, S_X, in a relationship on I/O ports. This is a formal way of thinking of a conventional hardware testbench, or even a model of the physical behavior of the environment of an embedded computer. The overall wired, ported model takes on the form of:

$S_W = (T_X,T_N,I_X=O_N,O_N=I_X,\Omega_X,\Omega_N,Q_X,Q_N,\Delta_X,\Delta_N,\Lambda_X,\Lambda_N)$.

The elements of the closed form S_W system tuple are related. For example, one system must not produce outputs which violate the input segments (value and time) of the

Figure 2   Context Relationship

other system. This implies that the physical time granularity (interval size) of the systems must correspond — typically one system must not produce outputs faster than another system can accept them, even if it is only to store them in its internal state for later processing.

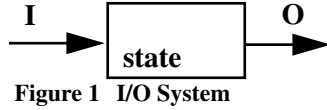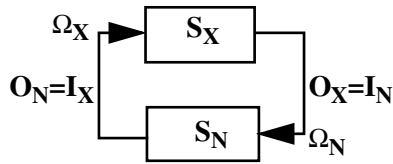For physical system models, the time base of both the testbench and the simulation are often both identical and physical, such as for discrete event (DE) simulations. Here the time base is totally ordered by physical interval sizes as well as for fixed interval simulations, such as cycle accurate simulations. Thus, $T_X=T_N$. The wired, ported system model thus simplifies to:

$S_W = (T_W, C_W, Q_X, Q_N, \Delta_X, \Delta_N, \Lambda_X, \Lambda_N)$

where T_W implies a common time base and C_W defines a common coupling basis, such as a wire or other interface model. So long as a common coupling basis can be found, the model can be extended to include more arbitrary forms of ported interconnect. This is a generalized I/O port model, with private state spaces in which entities can only exchange information across I/O ports. Interestingly, it has been difficult to achieve standardization for software components because of the absence of a software wire, or means of interconnecting on boundaries that behave like physical interconnect [5].

Analytical techniques for real-time scheduling of hardware-in-the-loop computation have focused on how multiple, largely independent inputs may be processed by a single or *conceptually single* processing element so that the real-time physical demands of the external physical system are met. Thus, given a computation platform, the challenge is in understanding if and how the computation can be *time-multiplexed* so that the computation demands presented by the external system are met over time [7]. Note that, regardless of whether the analysis views the inputs to the computer system as rate-based, for real-time demands to be met, the inputs must be considered as occurring over some fixed interval of time, with processing completing within some interval of time.

3.2 Logical-on-Physical Benchmarking

By contrast, systems modeled as a software program executing on a processor are represented as mixed logical and physical sequencing; these require the modeling of the resolution of two (or more) time bases. That is,
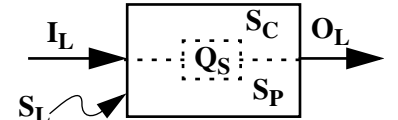

Figure 3   I/O System Formed by Shared State Between Two Layered Models

a software program is typically not thought of as being resolved to the processor through an I/O wire-like coupling of ported models, but by the sharing of a portion of the state vectors between two models that resolve to a single I/O system. This results in a layered partitioning of an I/O system as illustrated in Figure 3. S_P is the Platform and S_C represents the Completion of the model, with software. We say that software completes the system by resolution to the platform. S_P can be thought of as a model that generates one or more physical event sequences, Th_P, while S_C can be thought of as a model that generates one or more logical event sequences, Th_L. The resultant layered system, S_L, is:

$S_L = (T_C, T_P, I_L, O_L, \Omega_L, Q_C, Q_P, Q_S, \Delta_C, \Delta_P, \Lambda_C, \Lambda_P)$

where Q_S is a subset of state common to both Q_C and Q_P. Significantly, the state of the programmatic completion in the layered model, Q_C, need not be considered finite. Some portions of a software program may be written to allow for conceptually unbounded state which is only resolved to finite state when the program is resolved to a physical platform, i.e, at runtime. This also allows for runtime, data-dependent resource sharing of the hardware by the software.

S_L is characterized as a system formed by the resolution of two time bases through this common state and a scheduling contract between the layers. This results in a physical I/O system so long as one of the time bases is physical, i.e., it assigns physical time

intervals to the system outputs. For single clock domain [8] processors, this can be thought of as the state of the model of the software program, resolved to the processor state — the register file, program counter, and control register, through the implied scheduling contract (the instruction set architecture). Both the processor and software contain local state that does not overlap with the other. For instance, when the processor is modeled on a more finely detailed time base, it contains state on wires that interconnect registers. Similarly, the software program contains state in memory which may not overlap with the hardware view of the processor.

Q_S and either implied or implicit scheduling contracts between the layers form the means of resolving the sequencing between logical and physical layers in a programmable system, ultimately resulting in a physical system which can be analyzed and designed for effective sequencing — overall performance. For concurrent programs executing on concurrent platforms, a challenge is defining a general way to model state shared between the layers which will adequately capture high-level resource sharing decisions, but still in the absence of fully detailed models.

If the lower platform layer is physically sequenced, the higher, logical layer can be seen as a programmatic completion of the platform into an I/O system that interacts with some external environment S_X as in Figure 4. In the figure, the layered model, S_L, has been substituted for the system content, S_N, in the closed form relationship between context and content of S_W of Figure 2.

When designing an instance or a programmable platform, S_{Pi}, such as a general-purpose single processor or parallel processor, S_X can be thought of as triggering a program from a benchmark suite, $B_C = \{S_{C1}, S_{C2}, ..., S_{Ci}, ...\}$, and measuring latency of the functional state advancement in S_C. Performance evaluation is considered very differently from that of the hardware-testbench

Figure 4   Context, Content & Platform Modeling

style of design. One platform may be evaluated against another from a set of platforms, $B_P = \{S_{P1}, S_{P2}, ..., S_{Pi}, ...\}$, for a given set of benchmarks, B_C, by weighting performance of one benchmark against another, or even by considering throughput of successive benchmarks from the set B_C. From this, performance of anticipated actual programs is predicted. Or, B_C can be used to predict the performance of certain programming constructs on a given platform.
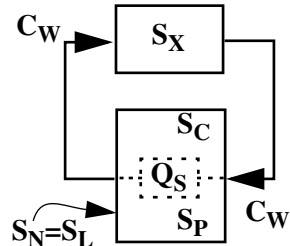
4. Design for Interaction

Neither the hardware-testbench style nor the software-benchmark style is adequate to capture completely the design interactions required in concurrent software-on-hardware systems. The hardware-testbench style of design specifies system throughput by presentation of time-bounded inputs to a physical model — these must respond within a fixed amount of time. This is one system reacting to another, where sequencing is ultimately done by physical time. The actual physical sequencing of the system is thus specified and not inferred, as in a programmatic sense. The software-testbench style of design characterizes the latency of logical-on-physical sequencing, but does not provide the ability to optimize the interaction of a programmable computer system with its environment or design context. Particularly significant is that the design context is another computer system.

Based on Figure 3, Figure 5 shows a more general model, a layered/ported model, in which two layered computer systems communicate via a common wire interface. Subscripts, N and X, refer to the computer system content and context, respectively.

The closed form system of Figure 5 now includes a richer set of possibilities for design and evaluation for overall performance of the interacting systems as a whole than hardware-testbenches or software-benchmarks alone. For instance, since the design context is now a logical-on-physical computer system, it is possible to optimize the design content to the state and scheduling decisions in the context — its

Figure 5   Layered/ Ported Model

data-dependent resource sharing decisions. The scheduling of the system as a whole may be considered.
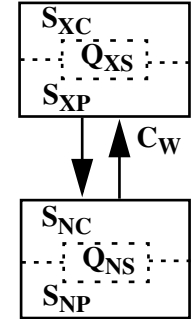
4.1 Conventional Performance Measures

The actual performance of the closed form system of Figure 5 can be seen when related to more standard techniques for measuring performance. A hardware-testbench style is often used to maximize throughput on a physical portion of the system, such as the wired interface, C_w. This can be considered the gross throughput of the hardware portion being evaluated, or its physical, *gross capacity*. Since gross capacity is often measured as a constant value over time (instructions/sec, bytes/sec, frames/sec), or in rates, we use the symbol R_G. Or, when a software-testbench style is used, benchmarks can be considered to characterize the *net capacity* of software-on-hardware execution for representative programs of the programmable computer system.

The net capacity, R_N, can be considered as a rate also, but for far more complex forms of computation or communication that characterize a wider range of data-dependent execution — for example, average numbers of DCTs, list insertions, or sorts per second, are often found in a benchmark. In general, R_N is a physical measure of logical execution performance of a program on a platform. Unlike R_G which tends to be a measure of constant throughput, R_N, tends to be averaged over numbers of datasets, or even a number of benchmark programs.

Significantly, in general, $R_N \neq k*R_G$. That is, R_N is not generally proportional to R_G. This captures the need to execute software benchmarks on platforms in the first place — execution time is determined by logical-on-physical sequencing which is, in general, even less proportional for concurrent software on concurrent hardware than for single processors.

4.2 Interactive Performance

In addition to R_N and R_G, overall performance of one computer system interacting with another can be thought of as the *interactive performance*, R_I, of the overall system. Again, significantly, $R_I \neq k*R_N$. The overall performance of one computer system interacting with another is not directly proportional to either the gross hardware performance or the programmed performance of any single portion of the computer system. This observation is consistent with the notion that adding a unit of logical or physical concurrency to an already concurrent system does not predict if the actual performance of the system will be improved or made worse.

Maximization of R_I for anticipated programming scenarios can be thought of as the art of computer architecture design for programmable computer systems. This acknowledges that the

system is not merely reacting to a hardware-testbench model, nor is it characterized only by the performance of a program resolved to execute on a physical architecture.

For instance, portions of the behavior of the system being designed might be improved by considering more effective relationships between the common state between each system, which runs counter to the notion of the design of the SoC as a reactive device. One system is not designed to respond or react to the state emitted by another system. Rather, the global context permits the two systems to be designed more effectively together.

This is shown in Figure 6 where common state, or a global context, Q_G, between the models of the computer system content and context allows for optimization of the net behavior of a computer system interacting with another computer system. As a physical entity, Q_G may either reside in a single copy sharable by both systems, or be resolved by regular interaction between systems. As a separate modeling entity, Q_G also often utilizes particular, custom portions of the system architecture. Unlike hierarchically composed finite state machines, Q_G is not a single state machine composed from many. Rather, it is a *portion* of the overall system state which is considered common across multiple clock domains — it models state at the same design level of detail as the content and context of the closed form system. Further, none of the state in the model of Figure 6 need be considered finite.

Figure 6 represents our final model of interacting computer systems, as introduced in this paper. We consider these systems to not be synthesizable. That is, we do not expect to be able to synthesize this level of system architecture from some abstract description. Rather, a creative design process is required. The purpose of the figure is to motivate, in a formal way, the implicit relationship between a design and its context when a creative design process is required. Clearly, novel design features result when designers are able to exploit knowledge about how a design is likely to be utilized. A designer creates a working model of the significant features of the way a system will be used — the design context — and reasons about how a design might most effectively interact with it. This implies two significant things. The first is that the model of the design context is as important as the model of the system — a highly manipulable model of each must match for effective exploitation of the design space. The second is that in order for creative designs to be possible, the design space as a whole must not be overly restricted; flexible manipulation is the key to effective designs. Overall, the designer must have system models *in mind* that allow for effective manipulation of the design space at hand — what a computer system is, as opposed to a highly idealized view of what it is not.

5. Illustration

No single example which would completely capture the broad design space advocated by the paper could be summarized in the space allowed. We introduce a simple example — discussed in terms of a design scenario — to illustrate some of our points. We modeled and simulated an SoC interacting with a server through a network. The SoC in Figure 7 consists of two hardware resources, modeled as two processors with concurrent threads. Processor P_1, models the main application of the SoC, which consists of n threads, Th_{1x}, that are clients to the server with which the SoC

interacts. P_2 models a software cache which utilizes three threads, Th_{2x}. The SoC interacts with a server thread through a high-level model of a network, shown as thread N_1. Processor P_3 models the server, which supports a dynamic number of client connections through its thread-on-accept operation. The system resources, the three processors and the network, are all physically sequenced. They have independent, continuous activation but also support logical sequencing in a layered manner. (In our layered, logical-on-physical modeling environment [1][2][3][4], physical resources would be modeled using C threads.) The threads mapped to processor resources, Th_{ij}, are logically sequenced. They are activated by functional dependency and resolved by a scheduler to the physical sequencing of the resources. (In our modeling environment, these would be modeled as G(F) threads — scheduler threads are not shown here in order to simplify the presentation, but are included in the simulation.)
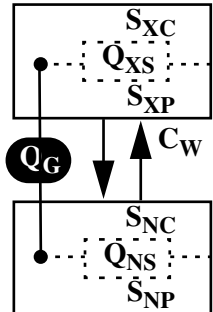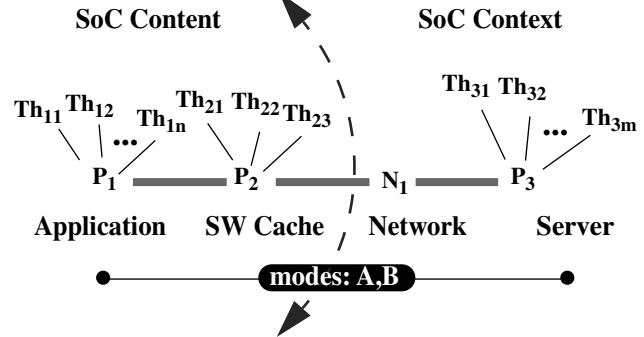

Figure 7 SoC Content/Context Relationships

The initial design is considered without P_2, nor any of the logical threads mapped to it. We utilized three threads in the client, where each attempts to request 10 files per second from the server for a total of 30 requests per second from P_1. The client and server threads communicate by exchanging fixed sized 1016 byte packets across resource N_1, which models a packet accurate network. A client initiates a connection by sending a single request packet to the server including the name of the file to be retrieved. The server spawns a child thread to fulfill the request and terminates the connection once the file has been sent. P_1 and P_3 provide computational power (physical sequencing) for a client or server thread to process a single packet in 10μs. A working set of 10 files was drawn from with a distribution of 60% under 10KB, 20% between 10KB-100KB, and 20% between 100KB-150KB. Each client request was then chosen uniformly over this distribution.

Simulation results over 10 seconds of simulation time are given in Figure 8. The performance of this initial system is given in the lower line (cache disabled). Here, client performance scales linearly as the bandwidth of the network increases, as anticipated. Since the network is the only variable impacting overall performance, the gross, hardware-like performance of the system can be captured as a simple rate, R_G.

Now suppose the designer would like to explore ways of obtaining increased performance of the client-server system. With limited network bandwidth, making the client-SoC or even the server (if it were possible) faster does not help. The only way to increase performance of the system as a whole is to decrease demand for the network — performance improvement must be obtained by considering the system interactions as a whole, not any part of the system in a hardware-testbench, reactive sense.

After further analysis, the designer determines that there are two distinct broad categories of data in the system, set A and set B. For set A, a cache on the server might effectively decrease the network
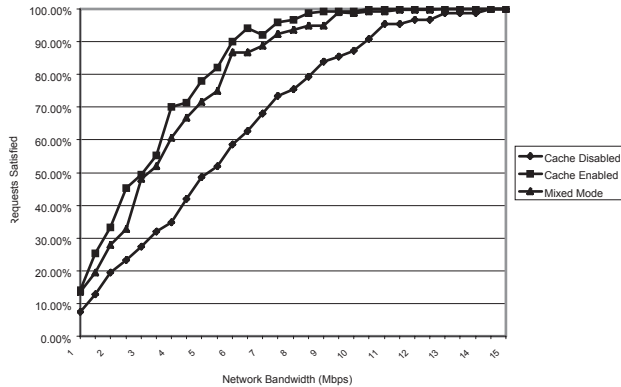

Figure 6 Interacting Computer Systems

Figure 8 System Performance Under Three Design Scenarios

utilization, because entries become stale relatively infrequently. However, this particular dataset is not amenable to compression, such as might be the case for audio and video that are already compressed. For set B, requests are more likely to become stale, as when updating more general data, however, this data is far more amenable to compression. Therefore, the designer proposes two design features be included in the system that operate under two different modes of operation, mode A and mode B.

In mode A, a software cache which utilizes a separate processor is included on the SoC. Note that, while the performance of the logical-on-physical sequencing of the system is included in the modeling, utilizing only the software-benchmark style of design would not include the flexibility to allow the designer to add the software cache, since it is modeled on a separate processing resource. The cache consists of three concurrent threads: one for managing requests for data and two for servicing an outstanding read or write to the cache, respectively. This allows for at most a single outstanding read and a single write to be serviced at a time. In this example, the cache is limited to storing five files and is ideal in that each of the files can be of arbitrary size. The replacement policy is random. Figure 8 shows the performance of the system in mode A in the upper-most line. Performance gains of almost 25% are observed over the cacheless system. Again, these results could be approximated in a hardware-testbench environment by limiting the content partition to the client processor and approximating the cache by assuming a fixed hit ratio for all client requests — approximating a fixed R_N for the cache. In so doing, modeling accuracy is sacrificed, such as the effect of the request distribution on the performance of the cache. More importantly, the simpler statistical/reactive cache model does not allow for more complex interaction of the SoC with its context when it is inappropriate to utilize the cache, as discussed next.

While the cache improves performance during mode A, the performance of the system is still relatively poor when it is inappropriate to cache, i.e., for set B. Thus, the designer introduces compression into mode B operation of the system. Mode B assumes that files under 10K are compressed to 50% of their size, between 10K-100K 20%, and 100K-150K 10%. Also, the server latency per packet is tripled and the client latency per packet is doubled, accounting for additional compression and decompression operations, respectively. Note that compression on both the client and server is required. Compression could have either been available on the server already, or it could be added, since the system context can often be considered another portion of the design of a partitioned system.

Figure 8 now shows the performance of the interactive system,

R_I, operating in both modes A and B, as the middle line. The server begins serving files in mode A, switches to mode B at 3 seconds, and then resumes serving files in mode A at 7 seconds. The switching between nodes is arbitrary — we included timing information only for interpretation of the results. This performance is obtained by modeling logical-on-physical sequencing in both the content and context, as well as common state, Q_G, which ultimately leads to two modes of operation for the system and the two distinct design features that support them. While Q_G is only two states in this simple example, in general it can be highly complex and need not be considered conceptually finite. The point is that system-level features of concurrent, software-on-hardware systems require modeling and understanding of design interactions in non-traditional forms of design space partitioning.

6. Conclusion

By formally considering the design implications of relationships between a computer system design (or content) and the context to which it is designed, we motivate the need to consider more complete forms of system modeling, design, and evaluation than afforded by design to reactive, hardware-style testbenches or software-style benchmark suites. Designers must be able to reason about the overall interactions of concurrent systems in novel ways that directly support shared state and layering concepts. This ultimately allows designers to more effectively balance computation systems towards optimization of performance and to reason about design partitioning and extensibility. We included a simple example to illustrate our concepts.

7. Acknowledgments

8. References

[1] J.M. Paul, D.E. Thomas. "A Layered, Codesign Virtual Machine Approach to Modeling Computer Systems," *DATE* 2002.

[2] N.K. Tibrewala, J.M. Paul, D.E. Thomas. "Modeling and Evaluation of Hardware/Software Designs," *CODES* 2001.

[3] J. M. Paul, A. J. Suppe, D.E. Thomas. "Modeling and Simulation of Steady State and Transient Behaviors for Emergent SoCs," *ISSS* 2001.

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

[5] D. Lyonnard, Y. Sungjoo, A. Baghdadi, A. A. Jerraya. "Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip," *DAC* 2001.

[6] D. Desmet, D. Verkest, H. De Man. "Operating System Based Software Generation for Systems-on-chip," *DAC* 2000.

[7] P. Pop, P. Eles, Z. Peng. "Schedulability Analysis for Systems with Data and Control Dependencies," *EURO-DAC* 2000.

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

[9] B. Zeigler, H. Praehofer, T. Kim. *Theory of Modeling and Simulation 2nd Edition*. San Diego: Adademic Press. 2000.

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

[11] K. Keutzer, S. Malik, A. R. Newton, et. al. "System-Level Design: Orthogonalization of Concerns and Platform-Based Design," *IEEE Trans. CAD*, pp. 1523-1543, Dec. 2000.

[12] E. Lee, A. Sangiovanni-Vincentelli. "A Framework for Comparing Models of Computation," *IEEE Trans. CAD,* Dec. '98.

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