Singapore Management University

Institutional Knowledge at Singapore Management University

Research Collection School Of Computing and Information Systems

School of Computing and Information Systems

4-2002

Multi-level modeling of software on hardware in concurrent computation

JoAnn M. PAUL

Arne SUPPE
Singapore Management University, asuppe@smu.edu.sg

Henele I. ADAMS

Donald E. THOMAS

Follow this and additional works at: https://ink.library.smu.edu.sg/sis_research

Part of the Databases and Information Systems Commons, and the Software Engineering Commons

Citation

PAUL, JoAnn M.; SUPPE, Arne; ADAMS, Henele I.; and THOMAS, Donald E.. Multi-level modeling of software on hardware in concurrent computation. (2002). *Proceedings 16th International Parallel and Distributed Processing Symposium, Ft. Lauderdale, Florida, USA, 2002 April 15-19*. 177-184.
Available at: https://ink.library.smu.edu.sg/sis_research/8282

This Conference Proceeding Article is brought to you for free and open access by the School of Computing and Information Systems at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Research Collection School Of Computing and Information Systems by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email cherylds@smu.edu.sg.

Multi-level Modeling of Software on Hardware in Concurrent Computation

JoAnn M. Paul, Arne J. Suppé, Henele I. Adams, and Donald E. Thomas,
Electrical and Computer Engineering Department
Carnegie Mellon University
Pittsburgh, PA 15213 USA
{jpaul, suppe, hia, thomas}@ece.cmu.edu

Abstract

The fundamental modeling differences between hardware and software modeling can be thought of as reasoning about connectedness vs. reasoning about interleaved (shared) access to resources. A natural design hierarchy for physical systems is component-based because of the existence of a consistent basis for interconnect between design levels. However, performance modeling and design of concurrent, programmable systems require new ways of thinking about what it means to abstract detail, add detail and partition a model of software executing on hardware. We motivate frequency interleaving (FI) as a common simulation foundation for these systems because it resolves flow and partitioning with software on hardware layering. Thus, FI provides a basis for hardware and software designs that do not simply co-execute together in fixed system views or later mappings but to truly be co-designed together from high-level conceptualizations to low-level implementable models. We include an example of a network switch within a client-server application.

1 Introduction

As programmable systems replace custom hardware and as parallel and distributed systems become more affordable and designed into smaller and smaller devices it will become increasingly important to model, simulate, design and integrate concurrent software with hardware architectures. Performance modeling and so effective design of these systems requires a consistent basis for resolving software on hardware from high levels of design through detailed designs. However, a consistent basis for modeling the timing interactions that lead to the physical performance of such systems from high levels of modeling to more detailed levels of modeling to actual systems has limited meaningful modeling and design exploration at high levels.

While hardware and software are seemingly similar in that they both advance digital state, as design abstractions they are very different. A significant difference is in regards to a design hierarchy.

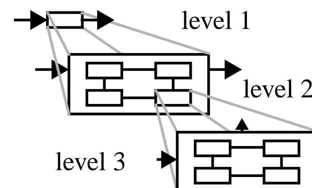
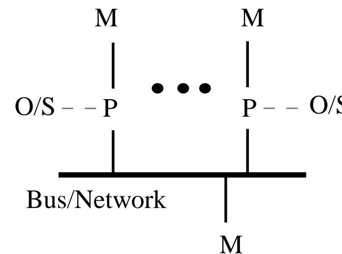


Figure 1 A Containment Hierarchy



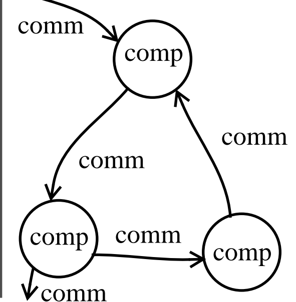


Figure 2 Contrasting Models

A design hierarchy is founded on a consistent definition of what it means to substitute more detailed models for ideal modeling assumptions at higher-levels of design. Components are a natural basis for a physical, hardware design hierarchy, where wires provide a common, consistent basis for interconnect across design levels as shown in the component hierarchy of Figure 1. Computer hardware designers utilize such hardware-structural views of systems as encapsulated computation and communication culminating in graph representations because the physical wire abstraction allows detail to be added at lower, more detailed levels of design (that include more components) without changing the timing properties at higher levels. The wired interconnection of components ensures a consistent design hierarchy with respect to timing and detail.

However, the absence of a software wire precludes a graphical containment hierarchy as being the most natural way to capture a multi-level design hierarchy when modeling software on hardware. The sharing of concurrent computation, communication and state resources transcends hardware-like 2-D structure, requiring a very different treatment of how sequencing — system timing and overall performance — is affected by design detail.

Figure 2A illustrates a *resource sharing model* where two or more processors, implicitly shared by programs (shown as the O/Ss), are connected to a common communication channel (a network or a bus), implicitly shared by protocol access. As shown, memory may also be common to the bus. Figure 2B represents the structure of *resource encapsulation models* that start with known interface boundaries. Events are used to activate or synchronize computation or tokens must be emitted and consumed for information exchange to take place across boundaries [11][14].

Figure 2B allows a designer to reason about flow through system elements in a consistent manner when the system is partitioned into more detailed elements. Although software is not explicitly captured in the structural diagram, Figure 2A motivates the importance of understanding the performance impacts of the interleaved access of the concurrent software to the *underlying, shared* system

0-7695-1573-8/02/$17.00 (C) 2002 IEEE

resources (processors, memory, and network).

A design hierarchy for layered software-on-hardware resource sharing models must provide a basis for partitioning and understanding information flow through the system in the absence of consistent wire-like interfaces.

Frequency interleaving is a simulation foundation that starts from a shared memory foundation for capturing hardware modeling as well as software that executes upon a hardware architecture. In this paper, we show how the physical modeling basis of FI can be used to capture a physical design hierarchy in the absence of a wire-like construct for connectivity. We also show that FI supports unrestricted software execution on this foundation. The combination provides the foundation for a design hierarchy for concurrent, programmable systems that is based upon atomic granularity of logical on physical state update instead of graphical decomposition.

2 Modeling Computer Systems

The basis for the differences in modeling hardware and software systems can be found in the ways in which logical and physical sequencing are related. In this section, we summarize our formal foundation from [1], which contrasts logical and physical sequencing and summarizes design implications of each.

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,

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

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 system models contain two kinds of event ordering — logical and physical [9]. 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 *physical ordering* tags represent a real time basis, establishing a physical basis for the system.

Both 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.

A sequence of events $Th = \{e_1, \dots, e_N\}$ is derived from a logical/physical time base T, the set of all state in the thread V, and the set of all state advancement functions F as shown:

$Th = M(T, V, F)$.

The calculation of a new set of values in V between two events in a sequence is functional and atomic. The functions themselves can have arbitrary complexity and are assumed to execute to completion between successive events. Thus event-based models can be used for arbitrary levels of system design. The function M is used to sequence the system state advancement. We distinguish two particular instances of M which produce event sequences for logical and physical event orderings:

$Th_L = M_L(T_L, V_L, F_L)$, and

$Th_P = M_P(T_P, V_P, F_P)$.

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 [11][13][14] or separating design concerns [12].

We first consider the design implications when logical and physical sequence are *independent*. When M_L 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.

0-7695-1573-8/02/$17.00 (C) 2002 IEEE

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 system is literally formed at runtime, when the software is deployed on the platform by a scheduler.

Next we consider the design implications when logical and physical sequence are *identical*. While it is possible to assign a time base to logically sequenced systems by assuming a fixed execution order and execution time on partially ordered events, these approaches effectively strongly couple a logical sequence to a physical time base either for hardware synthesis or for the analytical scheduling of real time tasks to a fixed, physical architecture [6][7][8][11][12][13] — the event sequences generated by M_L and M_P are the same.

For instance, assume that the thread sequence

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

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

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

Thus, each macro event, e_i, is seen to *contain* a sequence of micro events, $e_{i1}, e_{i2}, \ldots e_{ij}$, where all the micro events of macro event e_i must complete before any of macro event e_j can execute (where $i<j$). All logical state advancement may eventually be substituted 1:1 by a physical sequence Th_P — e.g., a set of gates. The physical time tags of the micro events may be combined to form a physical macro event tag; the substitutions can be made without affecting higher-level events nor events in other branches of the hierarchy.

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

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

Typically, concurrent software events are partially ordered and thus the micro states implied by e_1 and e_2 may be *interleaved* with each other on shared resources by a scheduler. As illustrated below, the actual execution of the micro event sequences are no longer substitutable and atomic, but interleaved.

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

For instance, e_{j1} above might be a hardware network event that makes data ready allowing the scheduler to schedule software e_{22}. 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.

Clearly in the performance modeling of concurrent systems, the function and architecture are not independently optimizable as in a single processor software system. When adding either physical or logical concurrency to a computer system, the new system may have better or poorer performance. Significantly, for performance modeling of concurrent software systems logical and physical sequencing must be related, not considered identical as in components or independent as in general-purpose software.

3 Logical and Physical Thread Relationships

The model of a system is shown in the middle of Figure 3 as a layering of software models on scheduling models on resource (hardware architecture) models in our Modeling Environment for Software and Hardware project (MESH). As introduced in [3], each layer provides a set of services — a virtual machine — to the next layer above. In our approach, the physical resource layer can be thought of as providing processing power to the next layer up of schedulers. The schedulers, in turn, split that power among the software threads they each schedule. Each software thread appears as a load on the scheduler; as each software thread uses its power, others are scheduled to maintain performance and fairness. Essentially, these schedulers resolve the logical sequencing with the physical sequencing through a simulation.

The layered approach supports design exploration of a concurrent computing system by allowing the models of each of the layers to be separately modified. Figure 3 illustrates this as three gray arrows emanating from the testbench. For instance, the rates of the resources (e.g., processor or network performances) can be modified separately from the scheduling algorithms on each processor, which can be modified separately from the software loading.

Some modeling environments allow certain models of computation to execute together — for instance, a discrete event system with a Kahn process network[13]. This works

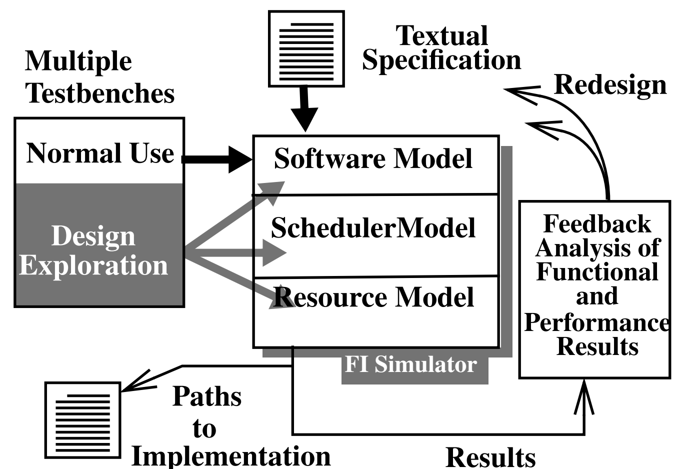


Figure 3 MESH Design Methodology

0-7695-1573-8/02/$17.00 (C) 2002 IEEE

well for validating the correctness of a system. However, our approach provides *performance modeling* by directly modeling, for example, a Kahn process network layered on a set of independently specifiable processors. It further allows us to model the performance impacts of other concurrent parts of the system.

Our approach to simulation uses *frequency interleaved scheduling* (FI) [2][3][4][5]. All computation in FI is modeled by two fundamental thread types — C and G(F). C threads model Th_P threads. They are rate-based threads that continuously sample inputs and generate outputs at fixed rates or frequencies (f_i), regardless of any other type of data events such as changes on inputs. Thus their activation is guaranteed and interleaved in time. C threads provide the resource basis for a platform style of multiple clock domain hardware design. As such, they are the foundation of all scheduling in the system, just as real hardware resources provide a foundation for all system execution. However, C threads are very high-level physical models and their rates can be varied separately from each other, allowing us to explore the resource layer.

G(F), or *guarded functional* threads, model logical sequencing of software — Th_L threads. They: have functional dependencies, can be eligible to execute (but resource starved), can be dynamic in number, need not execute atomically, and have flexible forms of time resolution. Thus, they capture layered sequencing.

While partitioning and flow are naturally captured in resource encapsulation diagrams, layering is not captured. In the next two sections we motivate FI as a common basis for capturing partitioning, flow and layering, thus forming the basis for a consistent design hierarchy.

4 A Rate-based Hardware Design Hierarchy

High-level functional models tend to be cycle-accurate models that implicitly assume ideal system properties such as perfect parallelism, zero-time computation, or unlimited bandwidth. These are our C threads. These are conceived to run at a rate which represents an overall system cycle (e.g., it might represent the cycle needed to process a packet of information). While rate-based models are intuitive foundations for system design, they are not an obvious means of capturing hierarchical detail and flow.

4.1 Modeling Flow Without Wired Interconnect

We provide the basis for reasoning about the flow of information through a system and provide a means of integrating resource/performance modeling with a self-timed layer [9] of computation. This is an increasingly important system design paradigm, as a basis for reasoning about the flow of information though a shared memory model *before* the underlying architecture has been defined.

Figure 4 motivates how we mix self-timed protocols in frequency interleaving. The loop-shaped arrows depict C threads, representing concurrency through multiple rates of activation, interleaved in a common memory space. The

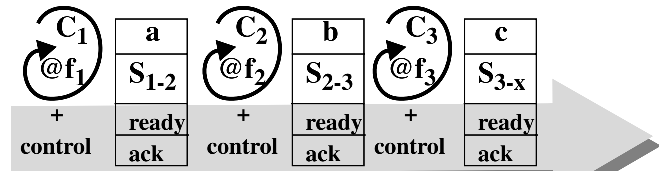


Figure 4 Layered self-timed logic on rate-based, physical resourcs

boxes represent state shared among (in this case) three C threads. Thus the boxes are labeled as $S_{i\text{-}j}$ denoting the overlapping (shared) state. Data is presumed to flow from left to right, so that location *a* as generated by thread C_1 flows through thread C_2 and emerges as location *b*. It then flows through thread C_3 and finally emerges as location *c*. Clearly, with execution based only upon relative frequency and a frequency of thread C_1 much greater than that of thread C_2, data generated by C_1 would be lost without some additional control superimposed on the C threads. At least a simple self-timed protocol with ready and acknowledge signals is needed.

Consider the mix of the C threads with the self-timed layer. The self-timed layer sits conceptually on top of the underlying resource model (visualize the gray arrow as being above the underlying C-thread resources). Thus, layer 1 consists of the frequency activated C threads, their internal state (not shown), and their shared state, shown as $S_{i\text{-}j}$ (with locations *a*, *b*, and *c* shown separately). Layer 2 consists of the ready/ack protocols, added to the shared state $S_{i\text{-}j}$ spaces and additional control in the C threads to support the self-timed protocols.

Clearly the flow of data from *a* to *c* is determined *both* by the control interdependencies established for each individual C thread and by the C threads themselves. The diagram is analogous to software modeling which ideally separates the software functionality from the underlying resources to execute it. But this approach adds resource modeling so that the performance of the interaction of software with an architecture may be considered.

Self-timed protocols are far from new. However, we provide a means of understanding the *interaction* of self-timed models with resource models when modeling mixed hardware/software systems. Consider a situation in which C_2 executes at a frequency much higher than either C_1 or C_3. The flow of information from *a* to *c* would then result in many executions of C_2 in which C_2 would simply find there was no data available to process. While C_2 would potentially have more computation capacity per execution cycle, it would not be able to produce more data, because of the dependency. Frequency interleaving allows us to adjust the rate of the C_2 thread to find architectural corner cases leading to design inferences.

For instance, past a certain point, a higher execution rate for the C_2 thread does not result in additional performance. C_2 can be isolated as a resource which either has the potential to be reduced in terms of relative rate (power) so

0-7695-1573-8/02/$17.00 (C) 2002 IEEE

that it only handles the given transfer of data, or it might be considered for additional functionality (loading or sharing of the resource).

4.2 Rate-Based Hierarchical System Partitioning

Intuitively, time granularity affects a system designer's interpretation of structure at varying levels of design detail. If a system were to be modeled as a single rate-based thread, all state inside the thread could be modeled as being perfectly accessible during each cycle of the single cycle-based model. As more rates are hierarchically substituted, providing more detail to the design, information access becomes less ideal as it must be exchanged between threads. Each finer-grained thread may also conceptually execute at a relatively higher frequency, since it is likely executing far less functionality. When the rates become closer to actual clock rates and the models become more detailed, a separation into actual devices emerges and the individual rates in the system begin to correspond to system components. Thus the hierarchy can be viewed as a few rate-based threads at a relatively slow system-wide execution rate implicitly representing a larger number of higher rate threads that specify the design detail.

Again, consider a situation in Figure 4 in which C_2 executes at a frequency much higher than either C_1 or C_3. The rate of processing of C_2 is high enough so that shared variable *b* is effectively available to C_3 at the rate at which C_1 produces *a* — the processing of *b* by C_2 is effectively transparent to the performance of the system. Further assume that C_2 did no functional processing in producing *b* from *a* — that is, the C_2 thread effectively acted as a wire, used to route the value of *a* from space $S_{1\text{-}2}$ to $S_{2\text{-}3}$. In this case, the variables *a* and *b* are effectively collapsed — state spaces $S_{1\text{-}2}$ and $S_{2\text{-}3}$ are conceptually shared and access can be modeled as such. At high levels of modeling the bus is ideal — it executes at a much higher clock rate than the level of model being considered.

However, if additional detail is added to the system such that the frequency at which thread C_2 operates is reduced — for instance the threads end up on different processors in a system — a flow channel emerges from the modeling space. This multi-level protocol view is naturally handled in C threads in FI, because the effects of each update are naturally viewed with respect to both the rate foundation, and a self-timed layer of information flow.

A further transformation might suggest that C_2 handles not only the communication between C_1 and C_3 but other (un-shown) communication as well — C_2 would then be evolving into a network that requires a level of protocols. The protocols would be captured by additional threads. It is interesting to note that combining C_2 with other communication threads breaks across the traditional (hardware) 2-dimensional, hierarchical view that computation and communication are physically distinct when they are, in fact, computationally related.

The point is that C threads afford reasoning about adding detail to a system and partitioning it into finer-grained elements without relying upon component-based structure.

4.3 Example

We designed and simulated three different network switch architectures, implementing multi-layer switching (Ethernet (IEEE 802.3) for OSI layer 2, Internet Protocol version 6 (Ipv6) OSI layer 3), Unicast, Anycast, and Multicast support, and Round Robin and Weighted Round Robin Quality of Service (QoS) mechanisms. Parameterization of the models includes the frequency of the C threads, look-up table sizes, number of ethernet ports, FIFO buffer limits, time threshold for stale entries, and enabling/disabling Layer 2 and/or Layer 3 switching. A common testbench interface exists across all models.

Our least detailed model, Model #1, is shown in Figure 5 and consists of several C threads: a single thread for the packet routing, a timer thread, a stale-entry thread, and a port service thread for each port modeled on the switch. Model #1 contains the least structural detail and is therefore the highest-level model of the switch. The routing fabric thread routes packets through the switch to the switch ports. Fixed frequencies are associated with each type of thread except for the routing thread, with f=0.1 assigned to the port service threads. Two primary variables are required for our models: work per iteration (the computation load of the loop), and frequency (the rate of recurrence — the resource power). Any frequency interleaved loop by itself is purely an abstract function — there is no limit to the amount of computation modeled in a given loop and so implied in a physical implementation.

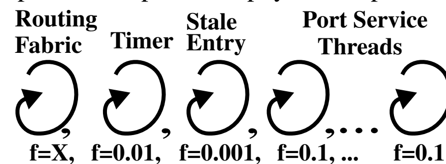


Figure 5 Network Switch Model #1

The frequency for the routing thread has been left as variable *X* so comparisons may be made within the system under different parameters and against other more detailed versions of this baseline model. The key assumption in switch Model #1 is that communications overhead is nonexistent within the switch fabric — it is ideal. Common networking issues such as switching fabric contention or buffer overflows are not realized in this model.

Note that the model includes several frequency interleaved threads. Interestingly, higher-level single loop models exist for the entire simulation. These packet-accurate models are used to explore high-level trade-offs associated with complex forms of scheduling but include less architectural insight. These cycle-based models are completely consistent with our simulation environment. However, we chose to include additional architectural insight; in this case the primary focus was to decouple the rates of execution between several general functions.

In switch Model #2, implementation detail is added by replacing the ideal routing fabric of Model #1 with a frequency interleaved model of a Banyan network. In Model #1, the routing thread handled any number of packets per cycle. In Model #2, the routing work is spread over a set of threads of bounded functionality. Notice the implied flow of data in the set of resource threads shown by the arrows in Figure 6. These arrows do not imply wire-like communication channels — the switch is completely modeled as a set of C type threads sampling inputs and producing outputs at regular intervals for which the flow of packets through the switch can be inferred.

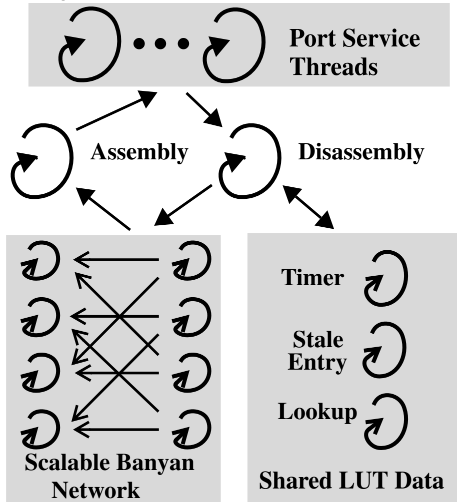


Figure 6 Network Switch Model #2

In making the design transformation to a more realizable model, work done per thread execution was re-defined to include the functional complexity of state advancement and the amount of state read and written in each execution. This implies more structure through data locality, and increased overhead when accessing non-local data.

Model #3 (not shown) represents a switching hierarchy. The switching hierarchy is designed to route local traffic in smaller Banyan networks, while non-local traffic must be routed through a central Banyan network. Thus, we used smaller Banyans in this model plus a new central Banyan for the hierarchical routing and a master lookup table. The design captures a trade-off that local routing would not leave the smaller Banyan network and access the central Banyan, thus, in theory, improving throughput.

Our simulation results are shown in Figure 7. For each model, we varied the routing fabric frequencies (the compressed X-axis) while all other frequencies representing system resources remained constant. In all cases we assumed unbounded buffers and we used a traffic model that should have benefited Model #3. The top line in Figure 7 represents the number of packets presented to the switch models by the testbench in a given period of time. We varied the frequency of the switching fabric (a single thread in Model #1 and multiple, identical threads in Models #2 and #3) and show the resulting bytes that are processed by the switch in the given time. The second line from the top represents switch Model #1. Note that the switching fabric is not the bottleneck in the design at high frequencies. Rather, the loss in bytes is due to other structural losses within the model. However, as the switching fabric frequency is lowered below .02, it becomes the bottleneck.

From Model #1 to Model #2, C threads are added to capture the added detail, the Banyan network threads and their interactions. Now a packet must traverse several threads to propagate from an input to output port, and there is contention for resources as multiple streams of data vie for a limited number of communication pathways. The lower level models require higher frequencies for the fabric, and may require scaling to attempt to achieve the higher level model functionality. In the case of Model #2, the frequencies of the fabric and the associated supporting subsystems may be increased to minimize the effects of the change in the design. Optimally, if the frequencies are high enough, the overhead and resource constraints added may disappear as is reflected in Figure 7 by the throughput approaching that of Model #1.

In Figure 7, switch Models #2 and #3 (the bottom curve) give identical performance — the two lines essentially overlap. Each curve shows that low frequencies in the Banyan networks interfere with switch throughput until approximately a frequency of 0.2. This is an intuitive result, since the more fine-grained Models #2 and #3 should require higher frequencies for the C threads, but more cycles required to pass information through their implied structure. The non-ideal nature of the switching fabric is captured as these fine-grained C threads must exchange information only on the basis of execution rates. Most interesting is that Models #2 and #3 exhibit identical behavior. This result is seemingly counter-intuitive, since the traffic model should have benefited Model #3. However, under this traffic model a hierarchical Banyan switch model produces little or no additional benefits. This shows that our interleaved rate-based model correctly captured the structural detail of the network.

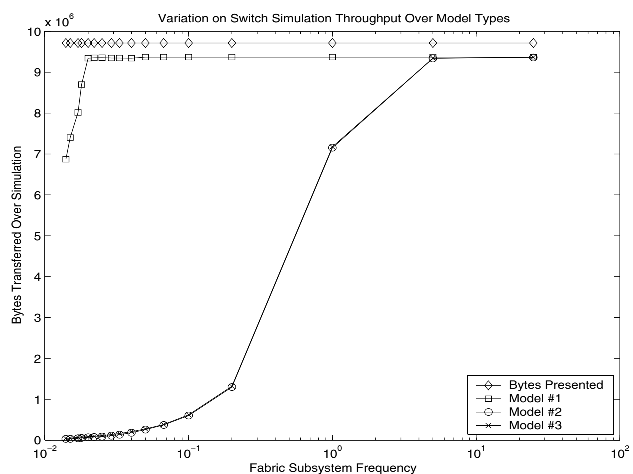


Figure 7 Throughput of Three Switch Models

5 Layered Foundation for Software

Clearly, performance modeling of data-dependent software can not be modeled as simple rates. However, as we resolve the logical sequencing of software to the physical sequencing provided by our rate-based models, we form the basis for a common model that allows for consistency across a design hierarchy. While FI allows for hardware-like structure to be captured in the absence of a wire model, significantly, it also supports a layered view of software, allowing for a common basis for the two design abstractions to be merged. In this section, we discuss how FI resolves software timing to hardware models. Since we do not require full resource models, we establish the basis for high-level models as well as more detailed ones that include full processor and communication models.

We modeled streaming MPEG-1 audio decompression in a client server model distributed over our network Model #1 of Figure 5 with a routing fabric frequency of 1 and 16 port service threads. Each of the 16 switch ports has a computational resource attached to it; 5 random traffic generators of two types, and eight clients accessing two servers. The first stochastic traffic generator creates full size packets at random probabilities and the other generates streams of packets with the likelihood of variable length packets at the beginning and end of the streams. This behavior models typical network traffic.

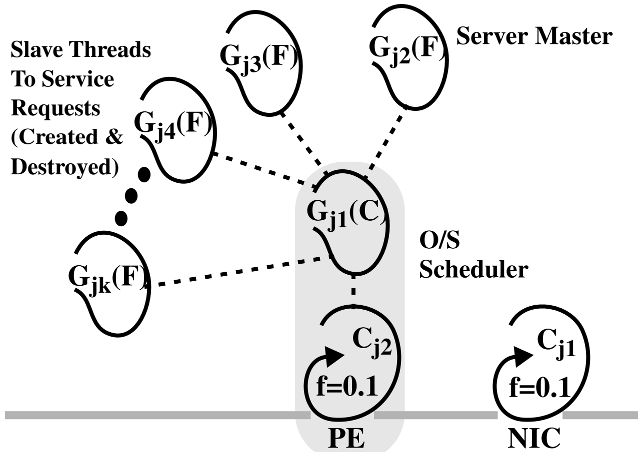


Figure 8 Layered Thread Detail of Server

Figure 8 shows the FI thread relationships for a thread-on-accept file server. A network buffer buffers both incoming and outgoing traffic. It is modeled as a C thread because it represents the network interface card (NIC) and can be considered an intermediary channel. The outgoing buffer is limited in size; threads wanting to transmit data are stalled until there is buffer room. The incoming data buffer is unbounded in order to prevent packets from being dropped before entering the system model. We are interested in the normal operation of the network; monitoring the backup at the buffer is an indicator of network performance. At reset, the server has only one server master G(F) thread which listens for clients requesting a file. This thread then spawns a child which sends the data to the client over the network interface. When finished, the slave is terminated — we can create and destroy threads that all compete for network resources.

The client shares the same code with the server thread. The two differentiate only at run time. The client has an additional piece of hardware (a C-thread) in the form of a buffered digital to analog converter (DAC). Its purpose is to play the decoded sound at the proper rate (e.g. 44 kilosamples per second). The client also only ever has exactly two threads mapped to its OS scheduler, an MPEG and a background thread. Further, the client has a variable frequency assigned to its processor (PE).

The MPEG-1 Layer III audio decoder [15] provides an interesting, computationally complex example with real time demands. All G(F) threads supported by the O/S G(F)-C threads are scheduled using a highest-priority-first round-robin scheduler. Normally, the MPEG decoder is the highest priority thread in the system with all other tasks executing in the background only when the decoder is blocked. However, by creating other threads of equal or higher priority, we degrade the decoder's performance through competition for the CPU. By changing the frequency of the CPU, we can compensate for changes in the performance of the software through changes in the performance of the hardware.

A 128 kbps MPEG audio file translates to 16 kilobytes per second. In the steady state, the network transmits one packet every 50 cycles, with respect to the rest of the system. If the packet payload is 512 bytes, each cycle of the simulation accounts for .0064 seconds. Assuming an infinitely fast processor and therefore no bottlenecks between the network buffer and the DAC buffer, and that the DAC buffer is able to empty itself each time it runs, the throughput is equal to the buffer size in bytes divided by two audio channels each 2 bytes wide, divided by 10 times the elapsed time of .0064. The factor of 10 is the period of a loop of frequency .1.

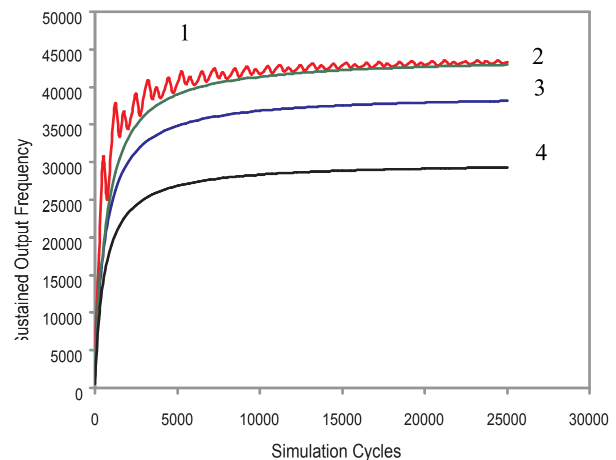


Figure 9 Client Node, Steady State Throughput

The actual throughput is dependent upon the algorithm that fills the buffer. For an MPEG file playing audio at 44.1 kHz, the buffer size must be at least 1129 bytes, given a frequency of 0.1 and the above computation of .0064

seconds for each cycle. Figure 9 demonstrates the steady state behavior of the system with a packet size of: 1) infinity, 2) 1250 bytes, 3) 1129 bytes, 4) 1000 bytes. Note that the curve is predictable, and that with the proper packet size, it converges on the throughput we would expect from a 128 kbps MP3 file playing at 44.1 kHz. Notice the roughness of curve 1 as compared to curves 2-4. Intuitively a buffer acts as a filter.

Of course the CPU is actually not infinitely fast. By assigning a cost to each of MPEG's discrete cosine transform (DCT) executions equal to some percentage of the CPU's total computational capacity, Figure 10 shows how throughput drops as computational cost increases. Curve 1 results when the CPU is not shared. The cost at the knee of the curve is .015, or 66 mDCTs (modified DCTs) per CPU cycle. A CPU capable of about 1000 mDCTs per second is required.

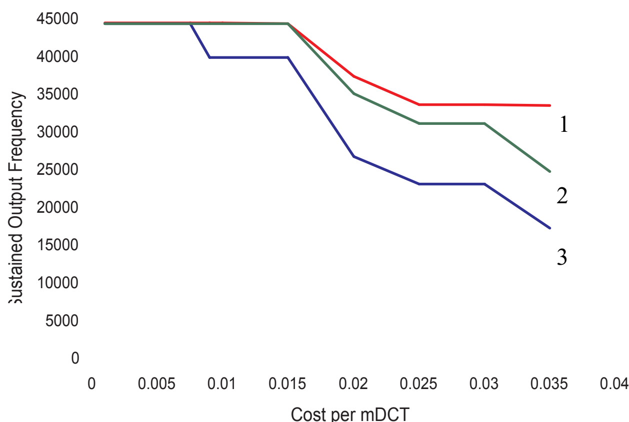


Figure 10 CPU Performance Vs. Throughput

Curve 3 results when the CPU is shared with a data dependent process (a random number search) which has priority over the decoder. On average, this process takes 23 percent of the CPU capacity before completing. As expected, curve 3 is about 75 percent of curve 1. The match is not exact because the system is actually somewhat more complex. Curve two results from changing the frequency of the CPU from .1 to .14, recovering most of the performance loss. However, since the input and output buffers have a frequency of .1 each (Figure 10), the CPU is out of sync with them, causing faster fall-off of the curve.

6 Conclusion

The modeling and design of concurrent, programmable systems that result in embedded and other mobile devices will transcend traditional approaches including those for parallel and distributed processing and the Computer-Aided Design (CAD) of custom hardware devices. As in CAD, the design process will become as important as the ultimate performance of the devices; design time is an important factor. As in parallel and distributed processing, understanding the impacts of concurrent software executing on concurrent software poses a performance modeling challenge for these programmable systems. One of the most important aspects of a design methodology around which tools and techniques can be based is that of a design hierarchy. Frequency Interleaving (FI) provides both a basis for hardware modeling as well as a basis for layered models of software systems. We have shown how FI provides the basis for a novel design hierarchy that is not based upon physical components, but which still includes the physical modeling of software on hardware. This will provide designers with a consistent basis for understanding the design implications of concurrent software executing on concurrent hardware for high-level designs as well as more finely detailed designs. A supporting modeling environment for these systems is being developed as part of the MESH research project.

7 Acknowledgments

This work was supported in part by NSF Awards EIA-0103706 and EIA-9812939, the General Motors Collaborative Research Lab at Carnegie Mellon, ST Microelectronics, and the Pittsburgh Digital Greenhouse. We also thank the other MESH research team members.

8 References

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

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

[3] J.M. Paul, S.N. Peffers, D.E. Thomas. "Frequency Interleaving as a Codesign Scheduling Paradigm," *International Workshop on Hardware/Software Codesign,* 2000.

[4] N.K. Tibrewala, J.M. Paul, D.E. Thomas. "Modeling and Evaluation of Hardware/Software Designs," *International Workshop on Hardware/Software Codesign,* 2001.

[5] J.M. Paul, A.J. Suppe, D.E. Thomas. "Modeling and Simulation of Steady State andTransient Behaviors for Emergent SoCs," *International Symposium on System Synthesis,* 2001.

[6] D. Lyonnard, Y. Sungjoo, A. Baghdadi, A. A. Jerraya. "Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip," *DAC01*.

[7] D. Desmet, D. Verkest, H. De Man. "Operating System Based Software Generation for Systems-on-chip," *DAC*00.

[8] P. Pop, P. Eles, Z. Peng. "Schedulability Analysis for Systems with Data and Control Dependencies," *EURO-DAC*00.

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

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

[11] 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.

[12] F. Balarin, M Chiodo, P. Giusto, H. Hsieh, et.al, *Hardware-Software Co-design of Embedded Systems. The Polis Approach.* Boston: Kluwer. 1997.

[13] E. Lee, A. Sangiovanni-Vincentelli. "A Framework for Comparing Models of Computation," *IEEE Trans. on CAD*. Vol. 17, pp. 1217-1229. December 1998.

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

[15] http://www.iis.fhg.de/

View publication stats