View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by CiteSeerX

Frequency Interleaving as a Codesign Scheduling Paradigm

JoAnn M. Paul, Simon N. Peffers, and Donald E. Thomas
Center for Electronic Design Automation
Carnegie Mellon University
Pittsburgh, PA 15213 USA
+1 412 268-3545
{jpaul, peffers, thomas}@ece.cmu.edu

ABSTRACT

Frequency interleaving is introduced as a means of conceptualizing and co-scheduling hardware and software behaviors so that software models with conceptually unbounded state and execution time are resolved with hardware resources. The novel mechanisms that result in frequency interleaving are a shared memory foundation for all system modeling (from gates to software-intensive subsystems) and de-coupled, but interrelated time- and state-interleaved scheduling domains. The result for system modeling is greater accommodation of software as a configuration paradigm that loads system resources, a greater accommodation of shared memory modeling, and a greater representation of software schedulers as a system architectural abstraction. The results for system co-simulation are a lessening of the dependence on discrete event simulation as a means of merging physical and non-physical models of computation, and a lessening of the need to partition a system as computation and communication too early in the design. We include an example demonstrating its implementation.

Keywords

Hardware/Software Codesign, Computer System Modeling, Clock Domains, Frequency Interleaved Scheduling

1. INTRODUCTION

One of the greatest challenges in capturing and co-simulating mixed hardware/software systems is the means of merging timed (resource, physical) and untimed (software, configuration) models of computation. Hardware models require time to resolve the structural interconnect of physical resources and to resolve time-based behaviors with external systems. Arbitrary structural interconnect of hardware behaviors has been conventionally resolved by time-tagged data events which activate an arbitrary number of user-specified resources simultaneously (at the same time tag). Software modeling ideally serves as a configuration paradigm in which behavior is not tightly coupled with physical architecture — rather, it configures a general architecture to execute the given behavior. Arbitrary resource management of behaviors specified in the software domain has been conventionally resolved by interleaving behavioral execution based on a higher-order "design to" paradigm, such as a fetch-decode processor (Turing Machine model), a program (function calls or access methods), or an interleaved scheduler (distributed software program [14]). When hardware and software are used to specify system-level behaviors, these two design scenarios are in opposition. Comprehensive computation system design requires that they be merged such that models of hardware and software behavioral design are not restricted.

The foundation of most physical (resource-based) models of computation are time-tagged data events [6]. Events are a model of activation and propagation of change of state, where state can be considered to be registered, wired, or in conventional memory. Previous mixed system approaches unite the hardware and software domains by introducing explicitly timed computation into the software domain for uniting models of computation in a discrete event simulator [9], by synthesizing timed and bounded (finite) software from a system specification language that does not utilize unrestricted, unbounded software models as a system level behavioral specification domain [1][7][10], or by utilizing untimed transaction handshaking for both hardware (architecture or deployment) and software models [2]. Untimed models do not capture physical (hardware) models of computation. Explicitly timed software computation requires the user to introduce time delays, processor models to reduce software to physical models [4][12], or other measurement techniques (e.g., instruction set simulators) to determine software computation time. Simulation environments such at Ptolemy focus on uniting heterogeneous models of computation (MoCs). The basis for uniting MoCs in Ptolemy is that all semantics lead to the same "bubble-and-arc" or "block-and-arrow" diagrams [5] with the constraint that all processes in the MoCs must have discrete firings [3]. These models of encapsulated computation and communication are natural for modeling embedded systems, but not natural for modeling software concurrency where resources (especially state) are shared and conceptually unbounded.

As software becomes an increasingly important behavioral specification domain for all digital computation, it must be fully included (not restricted) in any computation system model. Software paradigms are naturally hierarchical. Software can serve to specify system behaviors, or as "design to" paradigms. Dynamic memory allocation and distributed software schedulers are two examples of resource unbounded (software) MoCs which can blur the lines of distinction between software schedulers and physical architectures [15]. The challenge is to merge conceptually unbounded models of time and space with physical (timed) models of architecture and resources for trade-off analysis, co-simulation, and paths to physical synthesis.

The major difficulties posed by the strong coupling of software behavior to time-tagged data events or by reducing software to "bubble-and-arc" diagrams are in:

- modeling behaviors with conceptually unbounded state, communication, and execution times. System modeling must include a tolerance for time budgets for software functionality rather than just strictly timed (or untimed) execution. Software models must not be restricted to finite state machine-like models.
- modeling software schedulers as architectural features or trade-offs. Placing software schedulers and physically interconnected architectures on the same level of design enables design trade-offs between them — at various levels of design detail.
- allowing software to serve as a flexible, data-dependent behavioral configuration paradigm. The software paradigm includes written collections of behaviors that just execute "fast enough." Many background tasks can be considered to be completely time independent; some may be place-holders.

We present *frequency interleaving* as both a conceptual and scheduling device. It provides a basis for merging and resolving

unrestricted hardware and software models of computation accommodating the above modeling scenarios as well as the more conventional codesign scenarios (such as those that place bounds on software behaviors and time its physical execution). Frequency interleaving unites untimed, unbounded software models with finite, timed hardware models by using a shared memory foundation. All threads are resolved back to conceptual clock domains. Since the foundation is shared state, there is no need to partition a system as encapsulated computation and communication too early in the design.

2. FREQUENCY INTERLEAVING

The foundation for frequency interleaving is that all digital computation — from gates to software systems — can be viewed as an idealized shared memory paradigm, and that relative execution times of the threads modeling these systems can be based on a relative frequency of execution for each of the threads.

2.1 Thread Types

We begin by first defining the types of threads we use to represent systems. Our basis for behavioral modeling includes software functions, hardware threads, software threads, and software processes (threads with private namespace). These are all schedulable execution entities that advance system state — and are all potentially concurrent entities. We consider a thread the basic unit of modeling state advancement in any system. While our modeling techniques are aimed at high-level co-simulation and design exploration for system synthesis, we do not require libraries of components (unlike [8]), but instead focus on the semantics of co-computation. Our approach is developed from an identification of existing thread types from the hardware and software domains.

We have defined the Codesign Virtual Machine (CVM) [11], illustrated in Figure 1. Resource threads (R), such as always blocks in Verilog or process statements in VHDL, are threads that respond to a value change (combinational logic for asynchronous modeling), clock edges (global synchronization) or hardware waits (self-timed hardware synchronization of interacting finite state machines [13]). The common feature of these thread types is the one-to-one mapping of behavior to an execution resource. All of these threads may be considered to be continuously "sampling" inputs and translating (via functional computation) to outputs with some delay. All resource threads are derived from a thread type we call "type C" (for continuous translation).

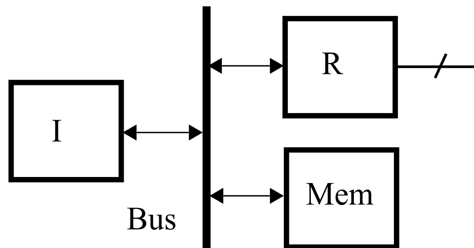

Figure 1 Codesign Virtual Machine

The interleaved (I) domain of Figure 1 models computation traditionally associated with the software domain. All I threads are derived from type G(F), where the G stands for "guarded execution." Guarded thread types are activated by the execution of some function, F. The function may be considered to schedule the threads it guards. Examples of these thread types include ones guarded by programmatic sequencing, resource multiplexed threads, dynamically scheduled threads (as a result of data-dependent programmatic sequencing), critical sections guarded by mutexes, and periodic threads mapped to a real-time operating system (where time is the guard). The common feature of these threads is that their activation is not continuous, but a function of a higher order scheduling algorithm (e.g., an operating system, program, processor, or thread scheduler). Each of the G(F) threads may be considered to be many-to-one mapped to system resources. System resources for type G(F) threads are not specified behaviorally, but are provided as architectural/scheduler features with computation power which are "loaded" by mapping behaviors to them. These will be described later.

G(F) threads may be activated by schedulers ranging from custom user-defined software to simple processor resource models. In the CVM, all G(F) threads may be considered to execute in a single I domain because of the presumption of an ideal scheduler that can resolve all G(F) threads to a single time base. (Hierarchical relationships between CVMs, introduced in [11], are not discussed in this paper.)

G(F) threads may represent behaviors ranging from a synchronous reactive "hard-time" paradigm to a task that executes correct behavior within a soft range of execution time, to a task that executes correct behavior regardless of execution time, to a task that executes only when a resource is otherwise idle and so may be considered to execute in zero time — all may be scheduled by a single, idealized scheduler. We are not designing real-time schedulers, nor making assumptions about software execution. We are accommodating the widest range of software modeling paradigms possible, including the view that software schedulers can be appropriate to think of as system architectural features.

The virtual machine of Figure 1 is literally the abstract system in the sky. It allows arbitrary connectivity to be modeled through a single idealized bus with unbounded bus width and addressability. It allows physical structure to be modeled on this bus by an infinite number of computation resources that continuously translate inputs to outputs. Finally, it allows shared memory access among all computation resources from this bus. Behaviors written to this model may be quite detailed in functionality, or may be mere place-holders. But, all behavioral threads can be simulated regardless of the underlying implementation; we just assume that there are enough computing and hardware resources to execute the behaviors. The observation that physical models of computation can be achieved with a foundation of a shared memory model of computation forms the basis for uniting hardware and software as truly unrestricted modeling domains.

2.2 Conceptual Clock Domains

Discrete event simulation is the foundation of most hardware models of computation because it models the sparse execution typical of many physical models. As execution models of physical systems become more complex, as with cycle-accurate modeling, time steps are utilized to activate the entire system — all system resources execute every cycle. This can be viewed as executing at a different frequency; one that's high enough to capture the behavior of the system (the clock cycle frequency) but lower than if we were modeling individual gates. In these models of execution, all behaviors are assumed to complete within a time step — and the entire system is designed to this single time step. We can think of this as a time budget within which the behaviors must fit. Interestingly, both synchronous digital hardware design and the modeling of analog systems utilize this paradigm.

In a cycle accurate paradigm, the time budget allows the bounds on exact propagation time of individual gates to be relaxed. Only the upper bound on a grouped execution of gates need be known while the tolerance of individual gates may vary. In addition, the clock provides an independent event that acts as a global guard function. In essence, the clock schedules atomic actions between events.

Software behavior is commonly designed without exact execution time of each line of behavior in mind, but rather with a time budget for a grouping of lines or a whole program. The time budget may be strict, as in hard real-time systems, or it may be very loose, as in systems with behaviors that execute "fast enough."

While a time budget for a synchronous hardware design paradigm is tightly coupled to the single clock, there may be a variety of time budgets for software executing in a system. These time budgets are coupled to schedulers which are ultimately resolved to physical

time, however loosely or tightly coupled. Software behaviors may be sequenced by instruction fetch, function call, mutex unlock, or server availability for example. These are software schedulers that result from higher-order abstractions. Software behaviors designed to software time budgets may be likened to hardware behaviors executing at a clock frequency — fast enough is good enough.

We consider a type C thread, which executes at a given frequency, to be a *clock domain* because it has an independent time basis. Indeed, in real systems this could correspond to a physical clock domain of a synchronous system. But it can also correspond to software schedulers since they resolve software behaviors to a physical time basis. Frequency interleaving provides for an idealized, protocol-independent, event-independent foundation by which clock domains can be resolved, and physical architectures and software schedulers that support software execution can be codesigned. Frequency interleaving allows the level of computer system design to be raised by modeling clock domains and software schedulers as equivalent architectural abstractions.

3. MIXED SYSTEM MODELING

The power of frequency interleaving is in the generalized software schedulers that resolve physical boundedness with conceptual unboundedness of time and space. Unbounded models permit software to have enhanced representation as an untimed configuration mechanism and as a model of shared memory arbitration and allocation that is not limited to deterministic, finite state machines. Frequency interleaving is a scheduling abstraction which is itself useful for both system conceptualization, and for providing paths to physical systems with schedulers. It will support system synthesis.

Frequency interleaving resolves hardware and software behavioral design to a unified state and time base, conceptualized by the CVM as an idealized shared memory architecture with state update properties. Differences in hardware and software modeling are in activation, time budget consumption, and atomicity of state update of hardware and software threads. Time budget consumption and state update atomicity are concepts that are unique to our frequency interleaved modeling foundation.

3.1 Thread Relationships

Consider the "completely connected substrate" of all state in a system with three type C threads as illustrated in Figure 2. All system state (wired, registered, conventional memory) is considered equally accessible in the same shared memory space – thus the three threads are shown as residing on a "grid" of memory to which they have ideal, equal access. (We can think of this as a flattened view of three C threads that might be in the R domain of in Figure 1.) Inter-thread state resolution is, ideally, shared with no penalty associated with information exchange between threads. Activation of the threads is based *only* on their relative frequencies. The relative frequencies are derived as an inverse function of relative computation complexity (computation inertia).

Each type C thread executes at a frequency relative to that of the other threads in the system and inversely proportional to the propagation time of the logic device it models. Further, each thread runs atomically, thus its update is the output set size of the thread. Partial updates of multiple output threads will never be seen by other threads because of this guaranteed atomicity. The only synchronization of state update between C threads is based upon the relative execution frequency of the behaviors specified by each thread.

A processor or software scheduler is a system level resource to which type G(F) threads can be mapped. That is, we can think of a G(F) software thread either as having a processor to run on, or as being scheduled along with other threads to run by some form of OS; the choice is up to the system designer and the level of detail being used. System level resource threads for these G(F) threads are modeled as type C threads with a specified frequency of execution. If the resource thread is a processor, the frequency

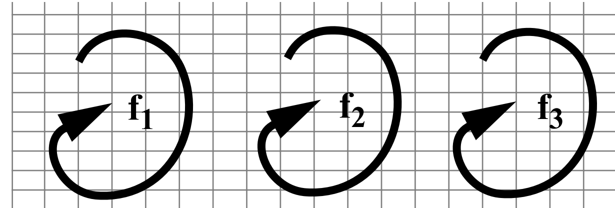

Figure 2 Frequency interleaved threads, sharing idealized state access

could be that of the instruction or bus cycle, depending on the level being modeled. If it is a software scheduler, it could be the frequency of thread scheduling. This scheduler resource thread can then execute the G(F) threads mapped to it as a function of scheduler behavior and interleaved execution frequency. The point is that the specification of time-independent software behaviors is de-coupled from the time basis that ultimately unites all behaviors.

We can combine C and G(F) threads as illustrated in Figure 3, which shows a system with three frequency interleaved clock domains (C threads) and four G(F) (software) threads. The execution model of all of the computation resources is the result of the interleaving of the frequencies of the resource threads in the system. Resource threads C1, C2 and C3 are the only threads in the system with a time budget, represented with a relative frequency of execution f_i. Threads C2 and C3 are resource threads that are completely specified behaviorally. Thread C1 is a resource thread that exports (includes and provides) a software scheduler. The scheduler directly executes configuration threads G1, G2 and G3. Configuration threads (type G(F)) are threads that can be mapped to threads that export schedulers. Thread G1 may also export a scheduler; it schedules and executes thread G4. Execution of software threads G1 – G4 can only take place during the time budget provided by frequency interleaved thread C1.

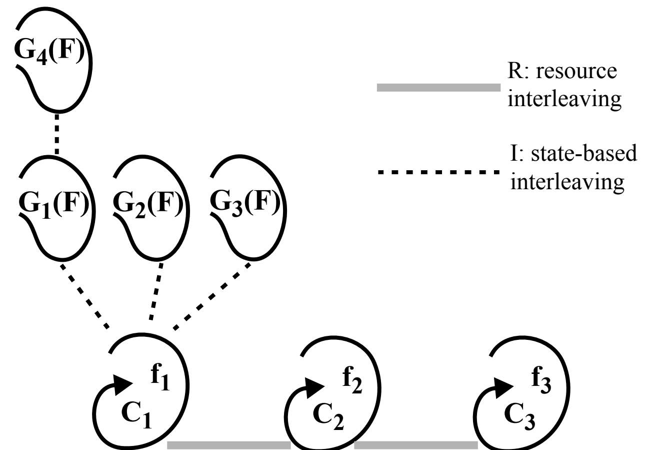

Figure 3 Frequency Interleaved Scheduling

Figure 3 depicts the two interrelated but de-coupled scheduling domains: the resource (R) domain in which computation is activated solely on the basis of time, and the I domain in which untimed software type G(F) behaviors are activated solely on the basis of state and resolved with an underlying resource time budget by OS-like simulation threads. Frequency interleaved scheduling in the R domain is represented as the solid line at the bottom of the figure; this allows the three type C threads to co-execute in a shared memory substrate such as shown in Figure 2. Scheduling in the I domain is represented as the dashed lines of the I domain.

The common feature of the state interleaved schedulers implied by the dashed lines of Figure 3 is that they sequence unbounded type G(F) behaviors and allow them to be resolved with the physical models in the system. The important modeling aspect of this

resolution is that we have made the modeling of clock domains (as found in hardware) equivalent to the modeling of the software scheduling domain (as found in software). These are the foundation of computer system modeling.

3.2 Schedulable Differences

In frequency interleaving, a resource C type thread has three characteristics that make it unique to a bounded, resource-based hardware model of computation: continuous activation, complete consumption of its time budget (a perfect match between behavior and available resources), and a one-to-one output set and state update atomicity size match. In contrast, software G(F) thread characteristics are: state scheduled activation, scheduler/context-dependent time-budget consumption (a variable match between behavior and available resources), and scheduler/context-dependent state update atomicity. The property of scheduled activation is traditionally associated with control flow modeling and is a part of our CVM modeling.

Context-dependent time budget consumption permits greater time-flexibility for certain software behaviors, thus allowing software to have a wider variety of actual execution time tolerances. Unlike hardware models of computation, rarely is software programmed to precisely match the available processing power. G(F) threads may not completely consume the time budget of the resource to which they are mapped — indeed, some G(F) threads may theoretically consume zero-time, since they are programmed to be background tasks. In addition, the execution of a G(F) type thread may produce a partial set of outputs each time it executes. Indeed it may not execute to “completion” during a given time period because its complexity exceeds that of the computation resource power to which it is mapped.

In frequency interleaving, software schedulers resolve time budgets to available resources with a wide variety of system modeling scenarios ranging from timed instructions to zero-time behaviors. The point is that the de-coupling between behavior and physical resource by a scheduler abstraction allows different software behaviors to be treated differently with respect to time. The scheduler may in turn represent a programmatic feature of a resultant system or a model used for the purposes of system simulation. Indeed the scheduler abstraction allows some sets of software behaviors to be treated differently within a single scheduler — which is typical of a priority-resolved RTOS or a custom simulator.

Scheduler-dependent update atomicity is necessary to capture the theoretically unbounded input and output set sizes of software. Unlike hardware behaviors, software behaviors contribute state update to shared memory architectures with variable atomicity. At a bus-level model of behavior, the bus width is the appropriate level of granularity for interleaving atomic update with other system resources. And yet, software behaviors written to higher order software schedulers provide interleaving with a wide variety of enforcements for critical sections, e.g., mutexes. The critical sections are a software means of enforcing atomicity. Context dependent state update atomicity permits input and output sets of software to be conceptually unbounded in size. The result is an ability to allow software to more naturally model the dynamic, unbounded model of a Turing Machine.

In summary, software behaviors can be resolved to context-dependent activation, execution frequency and atomicity by frequency interleaved schedulers. These permit greater approximation of unbounded time and state modeling. The unbounded time and state modeling are necessary to capture software as a resource configuration modeling paradigm as well as an unbounded resource modeling paradigm. By conceptually separating the activation mechanism, time consumption, and update atomicity of G(F) threads we have more accurately modeled software threads. By resolving G(F) threads to type C threads with a richer set of assumptions about software scheduling mechanisms, we allow software execution to be resolved with physical architectural models in a richer way. Further development of schedulers that resolve G(F) threads with frequency interleaved clock domains will be the subject of future research.

4. EXAMPLE

This section discusses a co-simulation implementation of frequency interleaving that resolves C and G(F) threads. Ours is but one implementation of the scheduling concepts. What is novel is the way in which threads are interrelated and interleaved on the basis of time and state sequencing. These support the frequencies and software schedulers that appear in the system. We chose to build our cosimulator from a completely custom scheduler as opposed to building it from a pre-existing simulation package.

Figure 4 shows how we co-simulate frequency interleaved systems such as depicted in the C and G(F) threads of Figure 3. In step 1, the base time of all threads in the system is initially calculated from the relative frequency requirements of all threads in our cosimulation scheduler. For the threads in Figure 3, this represents the relationships between frequencies f_1, f_2, and f_3. For example, if the frequencies are 2.0, 1.0, and 0.5, respectively, we define the frequency of a thread to be the execution rate of the frequency interleaved thread over the execution rate of a base thread. This means that every time the base thread runs, C1 runs twice, C2 once, and C3 half, relative to the execution of the base thread. The base thread executes in step 2 and is interleaved with each frequency interleaved thread — it schedules the frequency interleaved Ci threads, which execute in step 3. The Ci threads are statically scheduled, unlike discrete event lists.

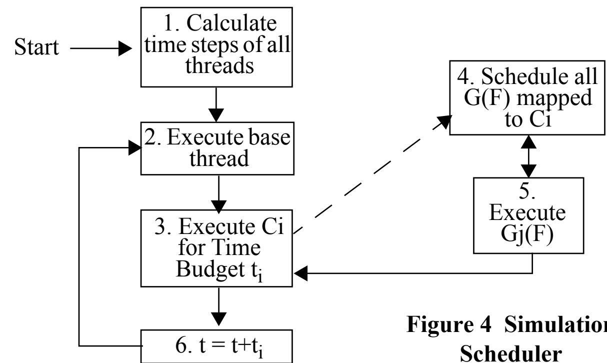

Figure 4 Simulation Scheduler

Each Ci thread may be a behavioral hardware resource thread (threads C2 and C3 in Figure 3) or a software resource thread that exports a scheduler upon which G(F) threads can execute (thread C1 in Figure 3). There is no limit to the number of Ci threads in the system, nor the number of Ci threads that export software schedulers, nor the number of G(F) threads mapped to each Ci thread that exports a scheduler. The G(F) threads are executed by the specific scheduler mapped to a Ci thread. The scheduler, if it exists, is executed in step 4. The Gj(F) threads of step 5, which map to the scheduler exported in step 4, then run a scheduling algorithm specifically defined by that scheduler until complete — where “complete” is defined by the scheduling mechanism of step 4. When the time budget of thread Ci has been exhausted, the time step of the system is advanced by the time budget allocated to the Ci thread in step 6 and control is returned to the base thread to schedule the next Ci thread. For the example frequencies, this gives us an execution history of C1, C2, C1, C1, C2, C3, C1.

Our current co-simulator is able to support exported POSIX thread (Pthread) schedulers, and can even export a discrete event scheduler from a Ci thread. Interestingly, discrete event schedulers can also be interleaved with other discrete event schedulers in the frequency interleaved paradigm. The discrete event scheduler is utilized in our example (below) to model the internal execution of portions of a system. Any hardware thread (including a single gate) can be modeled as a C thread, but for sparse execution, a discrete event scheduler may be more efficient.

To demonstrate our methodology and scheduler, we implemented a chess playing system using our cosimulator. Chess may be considered one of the earliest codesign problems in that custom hardware was utilized to speed up parts of the early machine chess algorithms. We used a public domain chess algorithm [17] as the chess playing engine and xboard [16], an X Windows package for displaying chess games, for the display. The implementation of this system took place in three phases, as shown in Figure 5.

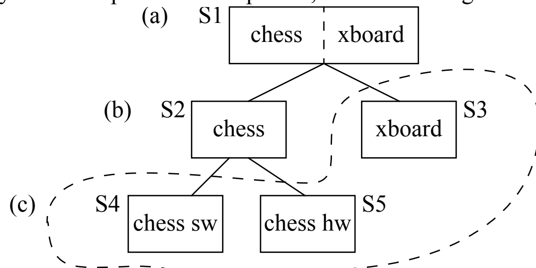

Figure 5 Chess System

Each 'S' is a scheduler. In (a), the chess algorithm was modified to support xboard and part of the chess algorithm was rewritten as hardware. The system was implemented in a single scheduler, with the software part of chess as one thread, xboard as another thread, and the hardware part of chess as several Verilog threads. (We currently use C or Verilog to specify C type threads.) In (b), the system consists of S2 and S3. Xboard was split from the chess engine and simulated in a separate scheduler. S3 was exported from a thread frequency interleaved with respect to the chess thread. The method of communication between chess and xboard, a FIFO buffer, did not change between (a) and (b), and no additional handshaking was required.

In both (a) and (b), the communication between the software and hardware portions of the chess engine took place by modeling events in a shared namespace. To send data to hardware, software would place an event on a wire. In (c), the system is composed of schedulers S3, S4, and S5. We separated the hardware and software parts of chess into two different schedulers. S5 is the hardware portion of chess and represents several C threads. S3 is unchanged and is reused to represent the display part of the system. With S2 split, S4 and S5 have different notions of time. We utilized a frequency interleaving paradigm to allow S4 and S5 to communicate by the sender writing into the shared memory space and assuming that the receiver would execute fast enough to receive it. Systems a, b, and c in Figure 5 produce correct behavior. As we move from the top of the tree to the bottom, we get closer to a physical implementation which may include physical resources and software schedulers.

5. CONCLUSION

We have introduced frequency interleaving as both a conceptual and scheduling device that provides a basis for merging and resolving unrestricted hardware and software models of computation. Conventionally, software modeling has been lowered to hardware modeling abstractions such as discrete event physical modeling or computation-communication partitionings. However, these have been ineffective in capturing the unbounded nature of software modeling as it interacts with physical architecture. This nature is essential in capturing the increasing use of software as a system-level behavioral specification domain that permits arbitrary configuration, shared memory modeling, and software schedulers as "design to" paradigms.

- By raising the physical modeling abstractions associated with hardware modeling to a software abstraction, we have raised the common basis for codesign modeling to be far more inclusive of unrestricted software modeling paradigms.
- By de-coupling a physical scheduling paradigm from an interleaved scheduling paradigm we allowed for representation of software schedulers as architectural features and trade-offs.
- By allowing software threads to be executed within a time budget, we have allowed software to take on a greater representation as an untimed configuration paradigm of physical systems.
- By resolving multiple software scheduling domains with a physical scheduling domain, we have allowed software to fully represent the unbounded, dynamic resource modeling characteristic of unbounded state machine models of computation in system level cospecification.

We have developed an initial version of our frequency interleaved scheduler and included an example which verifies its correctness. Future work will include assessing relative execution efficiencies with conventional co-simulators and further development of the frequency interleaved and CVM models for multiple level modeling and paths to system synthesis. Synthesis will include consideration of the "ideal" shared memory bus model as becoming less ideal (more restricted) at lower levels of a system hierarchy which will include topological (point-to-point) interfaces, blocking communications and memory partitions. Such system details will either be derived from the behavioral hierarchy, specified by the system designer, or both.

6. ACKNOWLEDGEMENTS

This work was supported in part by NSF Award EIA-9812939.

7. REFERENCES

[1] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, B. Tabbara. *Hardware-Software Co-Design of Embedded Systems*, Kluwer, '97.

[2] G. Booch, J. Rumbaugh, I. Jacobson. *The Unified Modeling Language User Guide*. Reading, MA: Addison-Wesley. 1999.

[3] 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.

[4] J.-M. Daveau, G. Marchioro, A. A. Jerraya. Hardware/Software Co-design of an ATM Network Interface Card: a Case Study. *Proceedings of the International Workshop on Hardware/Software Codesign*. Mar 15-18, 1998.

[5] 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.

[6] S. Edwards, L. Lavagno, E. Lee, A. Sangiovanni-Vincentelli, "Design of Embedded Systems: Formal Models, Validation, and Synthesis," *Proc. of the IEEE* 85:3 (3/97) 366-390.

[7] O. Færgemand, A. Olsen. Introduction to SDL-92. *Computer Networks and ISDN Systems* 26 1994, p. 1143-1167.

[8] D. Gajski, F. Vahid, S. Narayan, J. Gong. SpecSyn: An Environment Supporting the Specify-Explore-Refine Paradigm for Hardware/Software System Design. *IEEE Transactions on VLSI Systems*, Vol. 6, No. 1. March, 1998.

[9] K. Hines and G. Boriello. Dynamic Communication Models in Embedded System Co-Simulation. *Proc. of 34th DAC*, '97.

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

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

[12] V. Rompaey, D. Verkest, I. Bolsens, H. De Man. CoWare - A design environment for heterogeneous hardware/software systems. *Proceedings of EURO-DAC*, 1996.

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

[14] D. Skillcorn, D. Talia. "Models and Languages for Parallel Computation," *ACM Computing Surveys*. Vol. 30, No. 2,1998.

[15] D. Thomas, J.Paul, S. Peffers, S.Weber. "Peer-Based Multithreaded Executable Co-Specification" *Proc. 7th International Workshop on Hardware/Software Codesign*, 1999.

[16] www.research.digital.com/SRC/personal/mann/chess.html

[17] http://ucsu.Colorado.EDU/~kerrigat/