A Codesign Virtual Machine for Hierarchical, Balanced Hardware/Software System Modeling

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

The Codesign Virtual Machine (CVM) is introduced as a next generation system modeling semantic. The CVM permits unrestricted system-wide software and hardware behaviors to be designed to a single scheduling semantic by resolving time-based (resource) and time-independent (state-interleaved) models of computation. CVM hierarchical relationships of bus and clock state domains provide a means of exploring hardware/software scheduling trade-offs to a consistent semantic model using top-down, bottom-up and iterative design approaches from a high system level to the machine implementation. State domain partitionings permit run-time software schedulers to be resolved with design time physical scheduling as peer- and hierarchically-related architectural abstractions which cut across functional boundaries. The resultant abstraction provides "component-less" paths to physical design with greater accommodation of shared resource modeling. A simulation example is included.

1. INTRODUCTION

Digital computation is characterized by behavioral modeling to two fundamentally different virtual machines, or scheduling paradigms, commonly known as hardware and software. The software domain has been described as non-timed, interleaved, configurable, dynamic, data-intensive, unbounded, and resource-independent. The hardware domain has been described as timed, structural, physical, dataflow, synchronous reactive, and resource-based. Previous hardware-software codesign approaches do not fully represent both of these domains in arbitrary system views (level of modeling detail) from highly abstract to physical implementation. Instead, they have attempted to extend existing hardware or software virtual machines so that the "other" domain's behavior can be accommodated for a restricted system view. We assert that a mixed system modeling semantic must include unrestricted software and hardware virtual machines in unrestricted system views.

Software centric approaches such as UML [3] do not have an inherent semantic model of and path to physical computer hardware design — time-based scheduling models of state advancement are being considered by a working group which has been established to backfit or extend real-time analysis and design into the UML language [10]. Using C to describe hardware [11] does not allow software models to make effective use of system resources, nor provide a means of resolving hardware/software interactions. The CFSMs of Polis [1] are limited to finite models of computation. The SLDL effort [16] has targeted only reactive models. The basis for uniting models of computation in Ptolemy is that all semantics lead to the same "bubble-and-arc" or "block-and-arrow" diagrams [2]. These models of encapsulated computation and communication force the designer to partition too early in the design process and are not natural for modeling resource sharing. System C [17] extends an existing software language (C++) to include hardware-like modules, eliminating the need to describe hardware and software in separate languages. Encapsulated components both partition the system and provide processor-like models to which software is bound. Such partitionings do not naturally capture shared system computation, communications, or state resources because resource sharing requires global scheduling and protocols to cut across modular resource boundaries.

Untimed, data-dependent software models of computation such as shared state (stacks and dynamic memory allocation), shared computation (software schedulers), and shared communications (bus and network protocols) are increasingly prevalent as "design-to" paradigms in computation systems ranging from System on a Chip (SoC) cell phones to complete vehicle level electronics. These must be considered as design trade-offs with physical, structural models of computation in computer system architecture design at various levels of system modeling detail and at various points in the system design process. A codesign hierarchy must be based upon novel mappings of functionality (behavior) to both software (run-time) and hardware (design time) paradigms transcending traditional component boundaries. Further the single resulting semantic must be appropriate for multiple levels of system modeling from highly abstract, where parts of the system are considered ideal, to physically implementable, resulting in a common hierarchy that naturally provides paths to physical design.

Thus, for codesign to advance to the computer systems level of modeling and design, a codesign virtual machine must be developed. There are three attributes that such a virtual machine must have. First, the virtual machine must be abstract, providing a common, unrestricted idealized semantic toward which teams of hardware and software designers can design to. A programmer, for instance, has a software virtual machine which provides a common semantic for design (programming) without regard for the underlying physical execution of the software.

While the virtual machine is abstract at some point, a codesign methodology must result in a physical realization — this must be possible without changing modeling abstractions. Finely detailed descriptions must be describable to the same virtual machine as a highly abstract system description. Thus, the second attribute that a codesign virtual machine must have is that it be self-consistently partitionable (top-down synthesis) and composable (bottom-up integration).

DAC 2000, Los Angeles, California

Finally, a third attribute that the codesign virtual machine must have is a de-coupling of functional design (including functional encapsulation) from system scheduling, and yet a resolution of both in hierarchical system views. Functional encapsulation (modules, functions, objects, threads, processes, etc.) does not always naturally map to system scheduler partitions; system scheduling (especially software-like scheduling) can cut across and subset behavioral encapsulation (functional bounds) and may be designed before, after, or in concert with functional design. A functional description must be describable to a single codesign virtual machine as well as to sets of codesign virtual machines representing more detailed design time and run-time scheduling partitionings of the same functionality.

In this paper, we develop a Codesign Virtual Machine (CVM) which unites hardware and software models of computation on the basis of hierarchically related bus and clock state domains and inter-domain scheduling properties. Interestingly, partitioning these virtual machines can be viewed as defining new scheduling domains which result in software scheduling and hardware clock domains in the architecture. Thus, the virtual machine contract allows for mixed-system design-time (software scheduler) and run-time (hardware scheduler) designers to work on the same system design together.

First we discuss behavioral modeling that includes both the hardware and software domains, determining the basic semantics needed in CVM to model complex hardware/software systems. We then show how these models are partitionable and how these partitions imply major architectural features such as software schedulers, shared memory, interconnection networks, and hardware clock domains. We introduce a scheduling mechanism that combines timed and untimed systems and show simulation results from a custom simulator for the CVM model.

2. THE CODESIGN VIRTUAL MACHINE

CVM models hardware and software behaviorally. In this section we present the underlying semantics and their role in system modeling. The basis for modeling in CVM includes software functions, hardware threads, software threads, and software processes (which may be considered 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; the development of our CVM focuses on the identification of existing thread types from the hardware and software domains. We then consider novel hierarchical system integration methodologies and system scheduling strategies.

2.1 Resource-Based Execution Models

Hardware models advance computation state by a tight coupling of state and resources. This close coupling allows for arbitrary structural interconnect, synchronous (simultaneous) state advancement, and inertial propagation of state changes. The time-based modeling of the hardware domain resolves physical systems ranging from electro-mechanical continuous time systems to other computer systems for which protocols have not yet been designed.

We define resource-based computation (the R domain) as computation that advances state as a function of system inertia, or F(inertia). Table 1 lists models of state update that closely couple behavior and the underlying physical models that carry out the behavior. When new behavioral threads are added in the resource domain, new resources are added to carry out the behavior. These models of concurrency require, in general, a simulation to correctly advance state specified by the behavior. Simulators typically capture these models of concurrency by controlling the advancement of simulation time and by allowing designers to specify inertial properties of computation.

type	when they occur	design scenarios
C	continuous update	timed “sampling/generating”
D	any value change	async portions of HW
G	global sync	clock edges in HW
WH	hardware wait: change to level	HW synchronization, processor interrupt

Table 1 Resource-based Models of Concurrency

The last three entries of Table 1 are classically associated with a hardware semantic or virtual machine. One use of the D type is to model combinational logic which responds to any change in a wired data type with some inertial propagation of state to other portions of the system. One use of the G type is to model registered state advancement on a clock cycle (cycle accurate). The WH type is a level-sensitive wait which can be used to model hardware protocols or interrupts of a software execution. The D, G, and WH models of computation can also be used to model sub-system inter-relationships in a more general manner when time granularity is abstracted above the gate or register transfer level of modeling.

The first entry of Table 1, type C, is not typically considered to be a part of the hardware virtual machine. The C model of concurrency can be thought of as loops of execution that continuously translate inputs to outputs in a completely unsynchronized way with some propagation time, which can be thought of as computation inertia.

One way of viewing these loops is shown in Figure 1, where each processing element has only one loop assigned to it; it translates inputs to outputs in the same shared memory space by time-interleaving access on an idealized bus. Translation occurs at the inertial processing rate of each processing element. Private memories permit program/data interactions, unlike pure hardware models for which inertial propagation is (ideally) static.

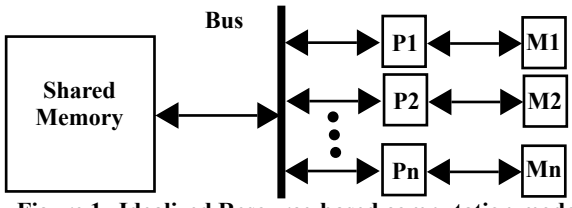

Figure 1 Idealized Resource-based computation model

Interestingly all models of computation in Table 1 can be considered to be derived from type C. Consider “always” blocks in Verilog [13] which can be used to model the D, G, and WH types. always blocks contribute resources to the system which can be considered to continuously be available to translate inputs to outputs with (potentially) some static inertial propagation delay. Discrete event simulations require scheduler events to conceptually “activate” these blocks for the purpose of efficiently modeling sparse execution. But the discrete event simulation semantic could also be conceptualized by an infinite number of loops that translate inputs to outputs at the delay (inertia) specified by the hardware designer — which is a closer analogy to the actual hardware synthesized by the hardware description. Thus, the last three rows of Table 1 are a modeling convenience for discrete event simulators but are all based on the underlying activity modeled by type C in the first row.

The “structural scheduling” implied by a hardware description

can thus be thought of as a shared memory that is continuously sampled and updated by the threads. Type C threads are a more general model of resource-based computation than a hardware model, and thus they form the basis of the thread type of the resource modeling domain of the CVM.

2.2 Interleaved Execution Models

The software designer's virtual machine advances computation state by state-interleaved programmatic sequencing, permitting time-independent behavioral modeling. The interleaving of program and data, function and program, task and scheduler, or computation and communication in the software domain allows for system resources to be interleaved in a data-dependent manner and conceptually unbounded models of computation without which dynamic (run-time) memory, stack management, and recursion would not be possible. The emphasis is on resource-sharing. State interleaved computation ideally ensures safe, atomic access to (shared) memory and CPU resources, even for multithreaded scheduling.

We define state-interleaved computation (the I domain) as computation that advances state as a function of system state, alone, or G(F(state)), where the "G" stands for "guarded execution." The threads in Table 2 list models of state update that are designed to be activated by a state-based, functional global scheduling paradigm, independent of resources except for an assumed single implied CPU and shared memory space.

type	when they occur	design scenarios
G(F)	functionally guarded	programmed activation
G(R)	resource multiplexed	resource sharing
G(S)	dynamic threads, function calls, mutexes, software waits, semaphores	data-dependent scheduling, guards for resource sharing
G(T)	periodic threads, mapped to O/S	resource-sharing RTOS

Table 2 State-Interleaved Models of Concurrency

The threads in Table 2 are all designed to the presumption of a global interleaved scheduling paradigm. (Even distributed scheduling paradigms — across a network with private memory space — are conceptually interleaved to a global scheduling paradigm [12].) Interleaved models of concurrency may execute in parallel, should resources in the system permit it. Additional processing elements made available to the global scheduler are intended to provide greater computation throughput, however, this "ideal" can be defeated by the synchronization overhead requirements of interleaved scheduling.

All threads in Table 2 are derived from the G(F) type, which is simply a thread of computation designed to be activated by "higher-order" interleaved state and functionality (modeled as separation of program and data). The G(R) type is a thread that can continually execute, but may have to share access to a resource. (If a G(R) type always has access to a resource, which is a possible physical implementation, it becomes a C type resource thread in Table 1.) The G(S) thread is any function (concurrent or otherwise) that is dependent on outputs of other G(S) threads for activation — G(S) threads model data-dependent concurrency such as function calls and user-modeled critical sections. The G(T) type is a time-based thread which is periodically activated by a real-time operating system. Even though the G(T) threads contain a model of time, G(T) threads are still designed to a functional scheduler

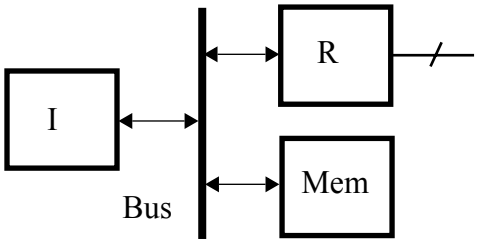

Figure 2 Codesign Virtual Machine

for which time is a global run-time state variable, not an entity used to simulate structural interconnect. All of these thread types have more complex activation than can be modeled by continuous functional translation found in hardware models. The resource sharing they capture is implied to be resolved by a global scheduling paradigm.

2.3 Combining Execution Models

Figure 2 portrays state-interleaved (untimed) and resource-based (timed) models of computation in an idealized architectural view of our CVM. The domains co-execute as peer-based modeling domains as in [14] but with novel scheduling abstractions and hierarchical relationships (described later) allowing the timed and untimed domains to be de-coupled, and yet resolvable to the same scheduling semantic. Each thread in the R domain implies added resources to support its execution. Resource threads (R) are derived from type C threads, continuously translating state in an unsynchronized fashion that merges inertial modeling with shared state modeling. Each thread in the I domain is derived from type G(F) threads, activated by a state-interleaved scheduler, which resolves to one or more continuous loop resources.

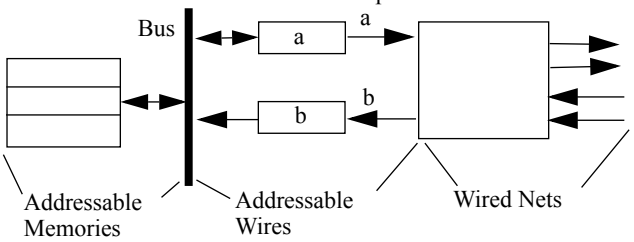

Figure 3 Connecting CVM State Domains

The CVM bus provides shared random-access addressability to some of the CVM state (Mem). CVM has three fundamental state types: address, wire, and addressable-wire as shown in Figure 3. addr is the I domain fundamental state type, wire is the R domain fundamental state type and addr-wire is the intra-CVM I-R domain interface resolution fundamental state type. Memory location *a (b)* resolves to wire *a (b)* in the R domain. Memory location *a* can be written by normal bus access from both the I and the R domain. Unlike typical memory access, when location *a* is written, an event is generated on wire *a* in the R domain, causing the possible immediate structural propagation of wired system state through the R domain. (Unlike location *a*, location *b* can only be read by either the I or the R domain as they access it as an address on the virtual machine bus.) Additional, hierarchical state relationships, based upon state domain partitionings are described in section 3.2.

Virtual machines provide a common semantic toward which designers specify systems. CVM has been defined to capture, on a peer basis, the models of thread activation and state advancement necessary to the design of physical systems containing resource-based and state-interleaved threads.

3. SUPPORT FOR SYSTEM SYNTHESIS

System designers ask two fundamental questions: "What

architecture is best suited for a given or projected set of behaviors?" and "For what types of behaviors is a given architecture and scheduler platform best suited?" Answering these questions requires a virtual machine for system behavioral design with paths to physical machines.

The virtual machine of Figure 2 is literally the abstract system in the sky; there are no limits on processing power (I), hardware (R), memory, or bus bandwidth. Behaviors written to this model may be quite detailed in functionality, or may be mere placeholders, but all behavioral threads can be simulated regardless of the underlying implementation. As a "black box," a single CVM has physical wired, and memory interfaces. CVM diverges from the typical software virtual machine by supporting self-consistent, hierarchical physical realization. For such system synthesis, any CVM must be able to accept any previously designed CVM threads (composition), or be partitioned (de-composition).

Other system codesign approaches focus on fixed architectures with pre-partitioned behavioral domains at relatively low (physically realizable) levels of system modeling for: co-verification of behaviors executing on pre-defined architectures [5][15], for sub-system interface resolution [7][9], or for mapping timed software threads to pre-specified architectures [6]. Top-down, iterative approaches for synthesis such as [4] require system functionality to be specified to a unique "top-level semantic" prior to architecture synthesis from pre-existing components, and are limited to channel-based functional descriptions that do not include resource modeling or accommodate shared memory and state-interleaved scheduling trade-offs. By allowing behavior to be designed to idealized, highly abstracted physical resource models and state-interleaved schedulers, we can represent hardware and software models in multiple levels of hierarchical system modeling from highly abstract to physically realizable.

3.1 State domains

Scheduling may be thought of as the advancement of system state. In hardware modeling all state may be considered to exist on wires with zero propagation time. In software modeling, all state may be considered to be equally accessible by name (namespace). These are implicit, ideal properties of the hardware and software designers' virtual machines. Natural bounds for partitioning and integration are where these assumptions may no longer hold. Beyond these bounds, or *state domains*, implicit assumptions of idealized state access may no longer be met. Interaction between these domains requires explicit scheduling — an interface which may or may not be part of the behavioral functionality of a system view. A natural wired state boundary is a *clock domain*. All state within a clock domain is advanced by a design-time scheduling paradigm, such as used by a hardware synthesis tool. A natural namespace boundary is a *bus domain.* All state within a bus domain can be considered to be advanced by a run time scheduling paradigm, such as a program or distributed OS.

Systems naturally contain multiple bus and clock domains that are hierarchically related. The system of Figure 4 might be considered to contain, at the highest level, three clock domains (bus1, bus2, and the network) and two bus domains (bus1 and bus2) resolved by the network interface. Within a sub-system such as the one defined by bus2, four additional clock domains might be appropriate to model (FGPA, ASIC, and two CPUs), as either a single bus domain resolved by a single software scheduler or as three bus domains — one to resolve the software scheduling on the two CPUs (across two clock domains) and one each to resolve the scheduling on the FPGA and the ASIC. These may alternatively be resolved by a physical (socket-like) interface that happens to be implemented on a bus. Other relationships are possible.

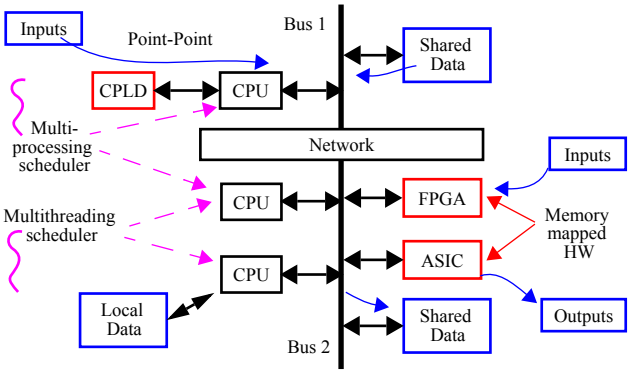

Figure 4 A System Architecture

Bus and clock domains form the basis for hierarchically related design-time and run-time scheduling. These permit more arbitrary scheduling relationships than component-based partitionings that are tightly coupled with functionality. Bus and clock domains are interrelated; they are not mutually exclusive. Therefore while they represent fundamental partitionings of the system, it is inappropriate to consider them as components. This permits hardware and software scheduling paradigms to co-execute as peers and raises the level of computer system modeling above that of traditional component-oriented approaches.

3.2 State Domains as Hierarchical Partitioning

Each CVM is a set of threads with idealized scheduling properties for their state types of "address", "wire" and "addressable-wire." These form the basis of system analysis and partitioning with respect to bus domains, clock domains and the interfaces between them. All addresses within a CVM bus domain can be considered "level 0 memory," and all wires within a CVM clock domain can be considered to have zero delay. Addressable-wires resolve interfaces between bus domains and clock domains and so have properties of both. By viewing a system as hierarchically related bus and clock domains, the functionality of the system may be designed to fundamental state types (addr, wire, and addr-wire) which are independently describable and resolvable as state domains in a separate system hierarchy. Figure 5 shows one possible hierarchy which may be intended to capture a high-level partitioning of Figure 4.

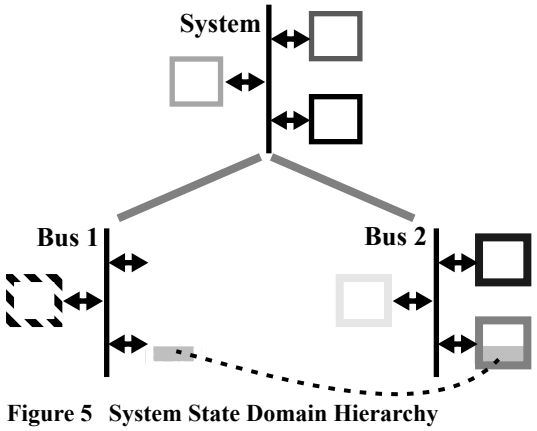

Figure 5 System State Domain Hierarchy

In Figure 5, all functionality (behavior) in the system can be described to the "System" CVM. Because all CVM models include simulation semantics (along with software-scheduling semantics) CVMs may or may not be directly physically realizable — all addressable state in the single System CVM is considered to share the single idealized bus, for instance. After considering multiple

state groupings, a system architect may determine that an appropriate sub-system relationship of bus domains and clock domains emerges such that the "System" can be thought of as containing subsystems "Bus1" and "Bus2." Bus1 and Bus2 are still not necessarily physically realizable — they are but one level of the system partitioning. Each CVM in turn may have multiple sub-system relationships. However the system depicted in the lower hierarchy is closer to physical realization — it is partitioned in a scheduling sense.

In the figure, two addressable state domains (addr1 of subsystem bus1 and addr2 of subsystem bus2) are now implied, along with two wired state domains (wire1 of bus1 and wire2 of bus2). Each of these may be considered to reside in the shared memory "local" to each CVM. Resolution between these now-partitioned state domains is implied by the dashed line that interconnects a subset of the global (to a single) CVM-domain state (which consists of types addr, wire, and addr-wire). The resolution of these state domains may, itself, take on multiple views. For instance, the dashed line may be considered (at one level of modeling) as ideal as a zero-penalty shared memory interface, or as detailed as a gate-level model of a network supporting a full software protocol.

We consider this type of partitioning analogous to a "coloring" of state domains. Consider, for instance, the line of functionality

```
c = a + b;
```

The functionality is written independently of the system upon which it will be realized. In a partitioned architectural system model, each of {a,b,c} may be considered to reside in different state (bus, clock) domains. State domains in turn may be resolved by a variety of software-programmatic and hardware architectural scheduling techniques which we consider analogous to *state paths*.

State paths are communication mechanisms which can range from "ideal" (as intra-CVM) to wireless (physical, continuous) mechanisms to simple wires, to hardware protocols (busses, networks), to software protocols. When CVM domains are partitioned there can no longer be the presumption that a simple level 0 sharing of addressable state or zero delay wired activation is in effect (although it may, if specified that way). Interestingly, the network can be viewed as its own subsystem or as a shared memory means of resolving the state domains of Bus1 and Bus2.

Inter-CVM state domain resolution allows for the development of "architectural protocols" to be independent of the development of behaviors. But because behaviors can be designed, developed, simulated, and analyzed without the presumption or development of full protocols or architectures, system analysis can be made in a far more general way than previously possible. Such "pre-protocol physical scheduling" can occur well in advance of well-established architectures and communications protocols.

For instance, if one domain is intended to operate an order of magnitude faster than another, a single state advancement in one CVM domain might represent the advancement of (on the order of) 10 state advancements with respect to the other CVM domain. If the worst-case inertial propagation time of the "wired net" domain in a given CVM partition were less than the atomic access time of the bus in the "addressable memory" domain in Figure 3, the propagation time of the "wired net" (R) domain could be considered to be transparent to the "addressable memory" (I) domain. However, if the R domain has inertial propagation time similar to the bus access inertia, and the software in the I domain is not finely timed for synchronous interaction, an asynchronous protocol is likely needed for domain interactions.

Hierarchical analysis to a unified scheduling semantic also affords the possibility of considering the implementation domain of partitioned CVM domains — moving the "hardware/software" slider switch. For instance, while it may be reasonable to convert a thread of type G(R) to type C simply by adding a resource, re-scheduling a thread (or thread grouping) of type G(S) to be physically scheduled instead of functionally interleaved requires isolation of its interaction with other threads in the system.

Note also the contrast to an object oriented paradigm in which system architectural effects (if they can be captured at all) must be considered to follow the partitionings (objects) of the behavior of the system. Like our approach, groupings of object-oriented state are made without consideration of the underlying physical architecture and scheduling mechanisms. But OO addresses only functional composition and de-composition, not physical design or physical/software scheduler trade-offs. The hierarchical encapsulation of OO discourages any design which cuts across object boundaries — but this is required for system level design. By contrast, our CVM hierarchy supports scheduler partitions based on a subsetting of all system state.

3.3 Frequency Interleaved Scheduling

The fundamental problem in current system modeling methodologies is limited flexibility with respect to treatment of time (and inertia) in uniting hardware and software models of computation. Intuitively, time must be the basis for uniting any model of computation in which it is present for any part of the model of computation. And yet, software models demand as much time independence as possible. Frequency interleaving is a simulation scheduling strategy novel to our research approach [8]. It is a means of resolving multiple clock domains and software schedulers in hierarchical system views; it unites untimed, unbounded software models with finite, timed hardware models.

Frequency interleaving is based on the type C thread that executes at its own frequency independent of other threads in a model. In the hardware view (R domain), these can be thought of as clock domains in a system that model system resources as relative time budgets tightly coupled to the behaviors that consume them. In the software domain, behaviors do not consume resources in a 1-1 fashion — the point of software modeling is capturing shared resources. Thus, C threads in our frequency interleaved scheduling semantic can also be thought of as providing a clock (time) budget to which multiple software schedulers can be mapped. Since all type C threads are activated only as a function of relative frequency (continuously interleaved) and not as a function of system state, the software schedulers are physically interleaved with conventional hardware threads in the system. And yet, software schedulers mapped to type C threads can activate an arbitrary number of type G(F) state-interleaved threads typical of the I modeling domain of the CVM. Thus type G(F) threads typical of the I domain are designed only to a state-interleaved model of activation, and resolved to a common time basis using a rich set of software-scheduler (bus domain) and time-budget (clock domain) resolution strategies. The system model may consist of an arbitrary number of C type threads representing multiple clock domains and software schedulers.

4. SIMULATION EXAMPLE

We have developed a cosimulation environment to verify the CVM modeling hierarchy. Since we are focusing on the system-level modeling semantic prior to the development of a language syntax, we currently describe and co-simulate multiple system models to be consistent to the CVM modeling semantic without the benefit of automated state domain partitioning and interface resolution — thus our hierarchy must currently be manually co-described with our functionality. However, the basis for our hierarchical state domain relationships remains consistent with our vision.

One example we have modeled in our CVM hierarchy and

frequency interleaved scheduling semantic is a floppy disk and floppy disk controller system. We chose this example because of the mixed time-based (hardware) and untimed (software) scheduling. The controller PLL alters its frequency to produce a derived clock signal so that the buffer can sample the data at the correct time [13]. The R threads are a PLL thread and a buffer thread. The I threads are the ls (unix directory listing) thread and the driver thread. The ls thread represents a class of programs that are often modeled to directly read from the disk — without modeling system architectural details. Since the driver could potentially be accessed by multiple threads, its interface is mutex protected — i.e., state interleaved.

We have modeled and cosimulated two system views, such as shown in Figure 5. In the top-level system view, all of the functional behavior in the system consisting of separate threads for a directory listing (ls), driver, buffer, PLL, and disk were initially described to a single CVM model and cosimulated. All state was considered to be shared, and relative execution was functionally interleaved in a single scheduling semantic. In this model, the disk produces a signal CLK/Data which resides in shared memory and so can be sampled by the PLL. The PLL C-type thread has a frequency 128x (on average) that of the disk C-type thread so that is has the physical resolution necessary to track the CLK/Data state location. The floppy disk can have jitter in its rate of rotation of up to 5%. Since the clock/data frequency may shift as the disk changes speed the disk must be modeled with dynamic inertia — this is naturally captured in a frequency interleaved scheduling semantic. To model the disk jitter, the disk thread changes interleave frequency (inertia), which also changes the frequency of CLK/Data. The clock and data have a 1/8th duty cycle, with the data occurring 1/2 cycle after the clock pulse.

In an alternate system view ("Bus1" and "Bus2" of Figure 5), the disk was placed in a separate bus domain. Thus the CLK/Data state is no longer ideally shared with the rest of the system. Instead the CLK/Data state, PLL and driver model a state domain interface between one subsystem consisting of the directory listing and buffer (bus1) and another for the disk (bus2). The CLK/Data state, PLL, and driver thus resolve scheduling between two CVM state domains. Figure 6 is a graph of CLK/Data period vs. cycle count for the partitioned system. The graph shows the PLL (dashed line) keeping track of the CLK/Data signal frequency as expected.

5. CONCLUSIONS

In contrast to more traditional component-based approaches, CVM unites hardware and software models of computation on the basis of hierarchically related bus and clock state domains and inter-domain scheduling properties. We extend the system scheduling paradigm from the existing h/s domains rather than restrict the modeling inherent in each domain. Partitioning these virtual machines can be viewed as defining new scheduling domains, resulting in systems with software schedulers and hardware architectures. Thus, the virtual machine contract allows for mixed-system design-time (software scheduler) and run-time (hardware scheduler) designers to work on the same system design together. State-domain partitionings in the CVM semantic provide the foundation for a next-generation scheduling semantic for arbitrary system views, thereby eliminating the need for separate gate, RTL, thread, process, architecture, transaction, etc. modeling domains. The single semantic accommodates idealized abstract views as well as paths to physical design. Results from our simulator show multiple CVM system views resolved to a frequency interleaved scheduling semantic. We are continuing to develop system modeling based upon the CVM.

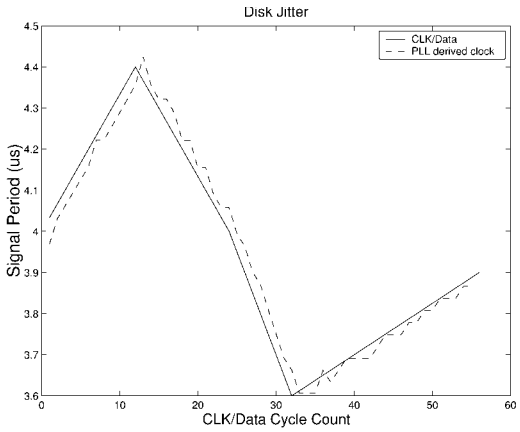

Figure 6 CLK/Data Period and PLL Derived Clock

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, et.al, *Hardware-Software Co-design of Embedded Systems. The Polis Approach. Boston: Kluwer.* 1997.

[2] 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.

[3] B. Douglass. *Doing Hard Time. Developing Real-Time Systems with UML, Objects, Frameworks and Patterns.* Reading, MA: Addison-Wesley. 1999.

[4] 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.

[5] K. Hines, G. Borriello. "A Geographically Distributed Framework for Embedded System Design and Validation," *Proc. of 35th DAC*. 1998.

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

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

[8] J. Paul, S. Peffers, D. Thomas. "Frequency Interleaving as a Codesign Scheduling Paradigm," *8th International Workshop on Hardware/Software Codesign*, 2000.

[9] K. Rompaey, D. Verkest, I. Bolsens, H. De Man. "CoWare - A design environment for heterogeneous hardware/software systems," P*roceedings of EURO-DAC, 1996*. 1996.

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

[11] L. Semeria and G. DeMicheli, "SpC: Synthesis of Pointers in C. Application of Pointer Analysis to the Behavioral Synthesis from C," *Proceedings of ICCAD98*. pp. 340-346. 1998.

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

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

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

[15] S. Yoo and K. Choi, "Optimistic Distributed Timed Cosimulation Based on Thread Simulation Model," *Proc. 6th Int'l Workshop on Hardware/Software Codesign.* pp. 71-75., 1998.

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

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