A Layered, Codesign Virtual Machine Approach to Modeling Computer Systems

JoAnn M. Paul and Donald E. Thomas
Electrical and Computer Engineering Department
Carnegie Mellon University
Pittsburgh, PA 15213 USA
{jpaul, thomas}@ece.cmu.edu

Abstract

By using a macro/micro state model we show how assumptions on the resolution of logical and physical timing of computation in computer systems has resulted in design methodologies such as component-based decomposition, where they are completely coupled, and function/architecture separation, where they are completely independent. We discuss why these are inappropriate for emerging programmable, concurrent system design. By contrast, schedulers layered on hardware in concurrent systems already couple logical correctness with physical performance when they make effective resource sharing decisions. This paper lays a foundation for understanding how layered logical and physical sequencing will impact the design process, and provides insight into the problems that must be solved in such a design environment. Our layered approach is that of a virtual machine. We discuss our MESH research project in this context.

1. Introduction

A system is defined as a set of inputs, a set of outputs, and a behavioral relationship between them, where behavior consists of functional transformations over time and state. Time is what gives the system model its physical nature; without time a system model would be purely functional, merely providing a transformation on inputs to outputs. Such models can be used for many kinds of physical systems, of which computer systems are only one kind. [9]

In a computer system, models of state advancement take on additional meaning over that in models of other physical systems. State in a computer system model approximates the internal physical properties of the system — its internal state and functionality. But the model takes on additional meaning since it is also a real entity — a layer of modeling — to which a layer of software may be designed as well as modeled. This software layer of state advancement is, effectively, independent of physical time and dependent only on a self-ordered, or logical context [8]; it is the programmer's virtual machine.

In the past, assumptions about the means of resolving logical and physical sequencing have provided two powerful computer design methodologies typically associated with hardware and software. However, emergent SoC designs are fundamentally different for the combination of three primary reasons. The first is that they contain multiple, heterogeneous *clock domains*. Clock domains are an abstraction of the physical limits of synchronous design — beyond these limits, computation and communication must be partitioned among multiple interacting domains. These domains give rise to custom hardware devices, heterogeneous processor types, and communication links (busses and networks). Secondly, their complexity requires that designers consider system wide effects of anticipated hardware and anticipated software over the lifetime of the product. The designs must be considered programmable since architectures and software evolve over time; a good design plans for this evolution. The third reason is that the systems will contain a mix of real-time and untimed behaviors. The notion of the purely reactive, embedded system is disappearing. Increasingly, untimed, desktop-style functionality is being integrated on embedded computers. This functionality is characterized by unknown execution times, aperiodic input arrival times, and internal state and is scheduled via dynamic scheduling techniques. These behaviors are sometimes called untimed because the system timing is internal and not tied to an absolute, external reference. The significance of these interacting design concerns cannot be overstated since they impact the appropriateness of traditional design methodologies such as component-based design, and function/architecture separation on computer system design methodology.

We show how component-based design and function/architecture separation are derived from assumptions on the way logical and physical sequencing are resolved in a system and thus are appropriate only after many design decisions have been finalized by other higher-level approaches. In component-based design, logical and physical sequencing must ultimately be too strongly coupled to an unalterable global reference. In function/architecture separation they remain independent or orthogonal, which applies only to situations in which they may be independently optimizable. In the design of concurrent software and hardware systems, adding a concurrent hardware element or software thread can affect, positively or negatively, the overall performance of the system — across component boundaries. Thus they are not independently optimizable; function and architecture must be considered together. To show this, we develop a macro/micro state model, which illustrates how logical and physical state advancements interact in a system.

We argue that a high-level design methodology for emergent, programmable, concurrent systems requires a layered model that captures how the logical and physical sequencing of events must *interact*. We show how they are layered and resolved through schedulers which are the means by which real concurrent computer systems address both correct logical sequencing and physical performance. Our approach is a basis for allowing designers to effectively develop instances of virtual machines — software, schedulers, and architectures — and evaluate their performance through simulation techniques. The virtual machine is a common but not identical view of logical and physical sequencing in a system. We briefly discuss our modeling environment which provides a simulation capability for this layered view of concurrent computing systems.

2. Event Models

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

A key reason for describing a system with a partially ordered sequence is to allow greater flexibility in the design of the system. If we (simplistically) view system design as finding a physical order for the logically ordered events, two partially ordered events give rise to alternate, and potentially concurrent, implementations of the system.

2.2 Deriving the Orderings

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:

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

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. Generally, higher-levels of system design have greater amounts of functional complexity between events. Lower, more detailed, system models have less functional complexity between events; more events are generally required to model a more finely detailed system.

The function M, above, is used to sequence the system state advancement. We distinguish two particular instances of M,

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

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

which serve to produce event sequences for logical and physical event orderings, respectively.

A system where the event sequences generated by M_L and M_P are the same is a discrete event (DE) system. The logical ordering of system state is tied 1:1 to the physical system timing. Thus, Th_P is merely a set of intervals which are assigned to the logical system states. Such a system is totally ordered, because once intervals are assigned — the system is designed — re-sequencing of the events is not allowed. When it is possible to couple logical state advancement directly with physical system resources, as in pure hardware modeling, the DE model is a good means of capturing the system behavior. Indeed, hardware design is determining the 1:1 mapping.

When M_L alone is used to sequence a system, the system will have unknown physical execution times between the events in Th_L — physical interval sizes between events are not part of the model. For example, the system may be a thread-level scheduler in which correct sequencing of threads is ensured for a variety of architectures, including a single processor, a distributed shared memory topology, and a network of processors. In software systems, a decoupling of the logical system behavior from its physical architecture exists — the same program will run on different processor platforms exhibiting different physical event time tags for the software's logical events. Thus, DE is a poor means of reasoning about the interaction of a high-level concurrent software system view with its underlying architecture because it requires the *a priori* 1:1 mapping of logical and physical events.

For our purposes, we are most interested in systems which have interacting models of Th_L and Th_P, but for which Th_L and Th_P do not resolve to the same sequence by insertion of fixed time delays in a Th_L sequence. Thus, system designers can understand the physical ramifications of unbounded, data depended software models executing on shared physical resources. In these systems, the ramifications of scheduling and sharing, and thus the synchronization of Th_L and Th_P at certain events, are determined by platform simulation. Indeed, in software systems the mapping between Th_L and Th_P events is determined at execution time, not at design time. Thus, the designer must be able to explore these interactions at high-levels through simulation so that effective interactions between hardware and software views may be designed.

3. Design Implications on Sequence Resolution

Logical and physical sequencing can be strongly coupled, independent, or related in computer system design. In this section, we show how assumptions on how logical and physical sequence are resolved results in familiar design methodologies associated with hardware and software. Then, we show how these traditional methodologies are inadequate to capture next generation designs, because of an inability to resolve logical and physical sequencing in a way that captures the design space at hand for these systems.

3.1 Composable Containment

The least detailed (most abstract) model of a system is one for which the system can be thought of as a black box of functional processing of system state and inputs, where all system state events are also associated with the system inputs. At this level, the state update functionality F is typically large-grained as is the time base. For instance, the functionality might represent an MPEG unit and the time base is time to process a single image. The tags are imposed by the system's required response to input events, so that the functional output processing and state advancement is atomic between the presentation of inputs to the system.

When designing highly complex systems, the

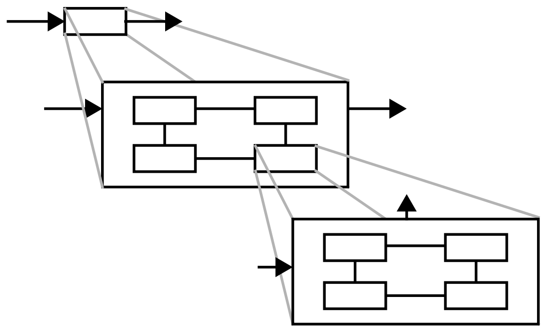

Figure 1 A Containment Hierarchy of Modeling

functionality F is typically decomposed into interconnected black boxes, as illustrated in Figure 1, permitting portions of the system to have internal inputs which are processed to internal outputs with a finer grained functionality F and time base T. This component-style decomposition for computer hardware/architecture is consistent with other levels of modeling detail such as gates, and transistors.

The physical design of a computer system requires reasoning about the actual execution time of the functionality. Assume that the thread sequence,

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

is a high level model. Because the events represent a relatively large amount of functional advancement, we term them *macro states* or *macro events*. These macro events can be decomposed, as above, 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, such as in a system in which time tags are ordered by physical intervals or are otherwise totally ordered as in a purely sequential software program, they allow for substitution on micro event sequences, allowing the sequence to be re-written as

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

Thus, each macro event, e_i, is seen to *contain* a sequence of micro events, $e_{i1},e_{i2},...e_{ij}$. Each macro event triggers a sequence of micro events which is presumed functional and atomic by the macro events. If the macro events are totally ordered, then the micro events must complete before the next macro event. Each micro event sequence in turn may contain another micro event sequence. This is physical decomposition, which ultimately results in simple functions, such as gates, separated by fine-grained events for computer hardware.

When a system is decomposed to adequate detail, the logical events specified by Th_L may be substituted by a physical sequence Th_P with physical time tags — the system is "synthesized." Alternately, Th_L may be related 1:1 to a set of physical time tags. Both of these result in a DE model of a system. Because of this strong coupling, component-style decomposition is possible at all levels of design. That is, since the macro events were hierarchically decomposed to produce the finer micro events, the physical time tags of the micro events may be combined to form a physical macro event tag.

The design of hardware systems using a tagged event model can be seen as the specification of desired system behavior to an idealized, arbitrarily extensible, spatial-component model. Each function executes independently of the others, and the functional complexity between events is independent of other system functionality. The system can be designed to have an arbitrary number of events occurring at the same time — their functional independence assures true concurrency. This is an idealized graph model of spatially interconnected components, or physical resources.

3.2 Independent Logical and Physical Ordering

When logical and physical sequencing are developed *independently*, a software design paradigm results. Clearly there must be some assumptions on the existence of a physical machine to carry out the functionality described by the software [10], but since there are no assumptions on physical interval sizes, the actual physical execution performance of the software is not specified as part of any system model. The system is literally formed only after the software, designed separately from the physical platform upon which it executes, is deployed on the platform.

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. For example, knowledge of the underlying physical platform does not significantly change the logical sequencing of a single threaded software program. This is so because the primary physical affect of a faster processor on a single threaded software program is to allow for smaller interval sizes between a logically ordered sequence of events — thus faster processing. Independently, the shorter the logical sequence of events in a software program, the more optimized the software will be.

Thus, software systems have benefited by the separation of functionality and architecture only because the logical and physical sequencing of a system may be considered independently optimizable [12]. However, this separation of design concerns does not extend to the design of concurrent systems, i.e., systems with both logical and physical concurrency.

3.3 Concurrent Hardware/Software Systems

In a concurrent, physical software system, logical and physical sequencing are related, but not identical. As a result, re-sequencing is possible at lower levels and component-style decomposition does not apply. Further, they are not independently optimizable as in a single processor software system. This is easily seen when considering the effects of adding either physical or logical concurrency to a computer system. In either case, the system may actually have better or poorer performance. For instance, the addition of a processor resource to a concurrent system may actually slow the system down for some applications for which the overhead of supporting the additional concurrency does not outweigh execution of the program on one less resource. Or, an additional resource may speed up computation. Similarly, the addition of concurrency to an otherwise sequential program by adding a thread or a process may result in an overall performance improvement — if the resources are there to support it, or decreased performance due to the need to support additional parallelism. Ultimately, in all of these situations, consideration of the interaction of the logical and physical sequencing is important to the overall design.

Consider the logical sequence of macro states in a concurrent software system as

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

Typically, concurrent software events are partially ordered. Further, the micro states implied by e_1 and e_2 events may be *interleaved* with each other on shared resources. A scheduler would be in charge of sequencing a collection of micro states of these events (and, most likely, others) as access to resources is arbitrated. A scheduler thus serves two purposes in a system. It gives access to resources in a data-dependent manner which is consistent with the required partial ordering but it also arbitrates time and so affects physical sequencing of the system.

As illustrated, the actual execution of the micro event sequences is 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 processing of all of the software being modeled,
- the scheduling method of the scheduler,
- and the shared resources (processors, busses, and networks) of the system.

Thus, the need to model resource-sharing forms of scheduling results in an inability to preserve higher-level sequencing with substitution of more detailed models — a component design hierarchy is inappropriate for concurrent, mixed hardware and software systems. Further, the approach of assigning time delays in software systems, as one would in a behavioral HDL (e.g., #delay in Verilog), is inappropriate because the timing of the microstates cannot be found through decomposition; the real delay can only be found from simulation of a real system. While some approaches effectively strongly couple a logical sequence to a physical time base [6][11][13], they give insight into the correctness of a system but they do not give insight into the *design* of systems in which the interactions of logical and physical sequencing have great impact on the performance of the resultant system.

The key point is that a truly concurrent software system requires a means of coupling logical and physical execution through schedulers that enforce proper logical execution order as well as facilitate physical system performance. Schedulers exist because computation, communication and state resources are shared in concurrent programmable systems — not specified by higher levels of design and not statically mapped. Indeed, some notion of shared state across computation components is required to effectively schedule shared resources.

4. Sequencing as a Virtual Machine

The computer system design challenge is determining a meaningful relationship between the high-level logical and physical models of a system, resulting in an effective design in terms of system performance, design time, and design extensibility. Concurrent hardware platforms affect not only physical execution interval sizes between events, but also the ordering of a logical sequence. Thus, the design of concurrent software and hardware systems requires ways of relating logical and physical system sequencing that reach beyond traditional hardware-like component-style decomposition, and separation of function and architecture. In this section we motivate a *virtual machine* relationship between logical and physical system sequencing. This can be viewed as a foundation for layering concurrent software on a hardware simulation. This approach affords flexibility as design detail in each of the hardware and software are kept independent at lower levels. Additionally, a high-level design relationship between them, resolved through schedulers, allows performance optimization to be brought to the highest levels of hardware/software codesign of multiple clock domain, concurrent systems.

4.1 Disjoint Hierarchies

Implicit relationships between the logical and physical sequencing of concurrent machines result in a virtual machine as we will describe in this section.

At any time, $t_L \in T_L$, $n_L(t_L)$ is the total number of events that have the same time tag, t_L. The maximum number of events that ever have the same time tag in the system execution is

$N_L(Th_L) = \max(n_L(t_L) \text{ for all } t_L \in T)$

which may be unbounded in a logical sequence. $n_L(t_L)$ is thus the concurrency of the system at any instant of time, so long as the functions guarded by the events are presumed to execute in zero time. The actual precision of an event model is limited by the granularity of its time base. Note that since event models have no notion of time passing between time tags, an event model is limited to consider all functions as executing in zero time, even if the output of the function is delayed to occur at another event which could be considered to model a duration. This discussion focuses on relationships implied by the granularity of logical and physical event models. This motivates the need to consider such relationships in a virtual machine.

Continuing in this way, $N_L(Th_L)$ is the maximum number of concurrent functions ever required to execute simultaneously by the system, i.e. at the same time tag. $F_L(e_{L,i})$ is the function associated with event $e_{L,i}$. Thus, if a set of events,

$E_L(t_L) = \{e_{L,i} = (t_{L,i}, v_{L,i}) \mid t_{L,i} = t_L \text{ for all i}\}$

share the same time tag, t_L, then they imply a set of F_L

$C_L(t_L) = \{F_L(E(t_L))\}$

which are a set of functions that are specified to execute concurrently in the system at time tag t_L. Each function $F_L(e_{L,i})$ has relative complexity, $w_L(F_L(e_{L,i}))$, which allows the amount of computation implied in each $F_L(e_{L,i})$ to be weighted. This complexity is a measure of the events the function implies on some finer-grained time base. The maximum complexity of any function in the system is

$W_L(Th_L) = \max(w_L(F_L(e_{L,i}) \text{ for all } e_{L,i} \in Th_L)$,

the maximum functional complexity over all events in the system. $V_L(e_{L,i})$ is the state updated by event $e_{L,i}$. The state that is advanced by the system at any instant of time is

$VT_L(t_L) = \{F_L(E(t_L))\}$

and thus the maximum state that must be advanced at any instant of time in the system is

$VM_L(Th_L) = \max(VT(t_L) \text{ for all } t_L \in T_L)$.

Th_P has similar properties, defined as $n_P(t_P)$, $N_P(Th_P)$, $E_P(t_P)$, $C_P(t_P)$, $w_P(F_P(e_{P,i}))$, $W_P(Th_P)$, $V_P(e_P)$, $VT_P(t_L)$, and $VM_P(Th_P)$.

Differences are clearly seen in the meaning of each of the properties with respect to logical and physical ordering of a system. Interval size, overall number of events over time, and maximum number of concurrent events are not specified by the logical model. Further, the maximum concurrency of each system, N_L and N_P, have different implications. N_L can only specify the logical possibility of concurrent events, but cannot, in the absence of being directly coupled to a physical model of implementation, guarantee the concurrency. By contrast, N_P specifies the actual concurrency of a system even if it is not logically utilized at any instant of time. N_P is thus finite in a physical system. Further, the weighted complexity of each logical function, $w_L(F_L(e_{L,i})$, may be considered data dependent, and thus variable in a logical system. By contrast, $w_P(F_P(e_{P,i}))$ is fixed, since it must be bound to a physical time interval.

Despite these differences, each system can be intuitively decomposed, separately, along complexity lines. Each function $F_L(e_{L,i})$ may intuitively imply thread sequences which can be directly substituted as, $F_L(e_{L,i})=Th_{L,i}$. This decomposition is similar to a hardware-style decomposition since there are no physical intervals or limits on the number of concurrent events specified in the partial ordering of a logically ordered system. As with the component-style hardware partitioning of a physical system, the maximum complexity of a given function in $Th_{L,i}$ must be less than the complexity of its higher-level function since decomposition implies that no complexity is being added to the system. Only detail is added in the form of a time base with finer granularity. Accordingly, the substitution of atomic functionality at higher levels of design with lower-level event sequences, or $Th_{L,i}$ for $F_L(e_{L,i})$, adds events to Th_L. Thus, $Th_{L,i}$ is a set of micro states to Th_L.

Thus, each of Th_L and Th_P may be decomposed separately as shown in Figure 2. However, Th_L does not decompose to a physical system because neither its maximum concurrency, $N_L(Th_L)$ nor any functional complexity weight, $w_L(F_L(e_{L,i})$ imply physical properties about the system that specify a spatial or temporal constraint on the system. Th_P, by contrast, provides two key physical properties of systems: the maximum amount of concurrency in a system $N_P(Th_P)$ as well as the interval size which is determined by the functional complexity which can take place between any two events, or $w_P(F_P(e_{P,i}))$. Thus, Th_L must interact with a physical platform in the formation of a system. The logical and physical hierarchies of a software system are thus disjoint, since concurrency and functional complexity in the software hierarchy do not imply concurrency or functional complexity in a time-

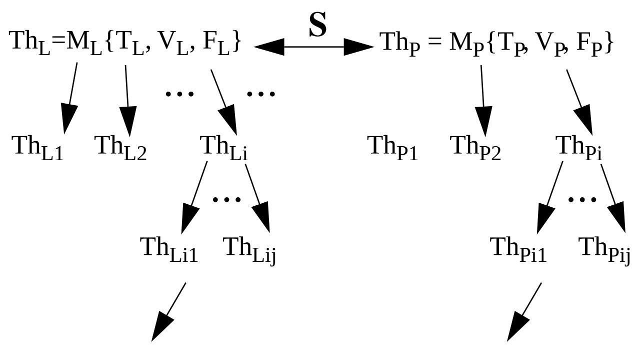

Figure 2 Resolution of two system hierarchies

ordered, physical hierarchy.

4.2 Toward Layered Virtual Machines

The central question then is, how are the two hierarchies related for optimal system design? Intuitively, the logical and physical hierarchies must be resolved on “appropriate” design levels. For instance, designers think of mapping N_L threads to N_P processor resources, not N_L threads to N_P gates. In the latter case, the logical event sequence is too far removed from the physical event sequence. Clearly there must be an implied relationship on properties such as function complexity between events and total number of concurrent events at any time tag — a virtual machine assumed allowing macro states to be effectively resolved.

Consider the implications of the groupings afforded by event sequencing of systems. Each logical or physical event in a system sequence encapsulates computation by ordered events. That is, the state advance functions, $F_L(e_{L,i})$ and $F_P(e_{P,i})$, are encapsulated functions that execute atomically, ordered by events. Thus, each event is a potential communication point in the system. However, the maximum number of concurrently executing threads in a system cannot exceed the maximum number of physical threads that can be executed at any instant (i.e., $N_L(Th_L) \leq N_P(Th_P)$). Further, the maximum amount of state communicated between events in a system cannot exceed the maximum communications capacity of the system (i.e., $VM_L(Th_L) \leq VM_P(Th_P)$). If this is not true, the logical system must adapt to the boundaries imposed by the physical system upon which it executes. Simply, the logical sequencing must compensate for the limitations of the physical machine by the addition of state and functionality that schedules logical state to resources.

Typically, the complexity of logical state advance functions, $W_L(Th_L)$, exceeds that of physical state advance functions $W_P(Th_P)$, requiring the construction of logical functions from smaller physically implemented functions. (If the functions were more complex, they couldn’t be scheduled by the logical system.) Thus, logical hierarchies reside on physical hierarchies, resulting in a layered view of systems, where logical functionality is constructed on available physical functionality through schedulers.

5. Frequency Interleaving in MESH

We have implemented a layered, virtual machine model in our Modeling Environment for Software and Hardware project (MESH). The modeled system is shown in the middle of Figure 3 as a layering of software models on scheduling models on resource (hardware architecture) models. As introduced in [2], 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. Thus the rates of the resources (e.g., processor or network performances) can be modified independently of the scheduling algorithms on each processor, which can be modified independently of the software loading. The key issue is that although the levels can be modified separately, they are related through the logical and physical sequencing allowing simulation to discover system performance.

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 well for validating the correctness of a system. However, our approach provides *performance modeling* by directly modeling, in layers, for example, a Kahn process network running on a set of independently variable processors. It further allows us to model the performance impacts of other concurrent parts of the system. Clearly, just making a system concurrent does not guarantee performance improvement; these interactions must be measured.

Our approach to simulation uses *frequency interleaved scheduling* (FI) [1][2][3][4]. All computation in FI is

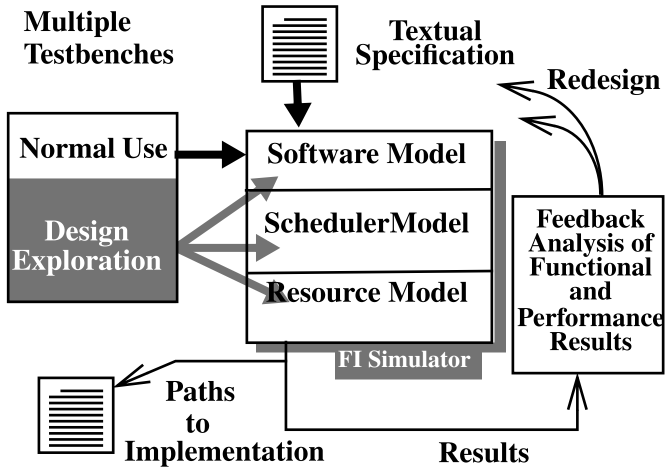

Figure 3 MESH Design Methodology

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 independently of each other. This independence allows us to independently explore the resource layer.

G(F), or *guarded functional* threads, model logical sequencing of software — Th_L threads. They:

- have *functional dependencies* — their activation depends on the state of the functional execution.
- can be eligible to execute, but *resource starved*,
- can be *dynamic in number* — they can be created and destroyed, as needed,
- need not execute atomically — atomicity must be explicitly modeled by the designer as a *critical section*,
- have flexible forms of *time resolution* — functional instrumentation of a G(F) thread's source code determines how much computational complexity was represented between successive calls back to a *scheduler*.

We also define G(C) threads which are scheduler threads; they are guarded by a C thread and schedule G(F) threads. We also define G(T) threads which are guarded by a real-time constraint. Together, C threads and G threads form the basis of mixing logical and physical system sequencing so that a virtual machine for the system can be discovered.

We have used MESH to characterize the response of the design as architectural corner cases [3] and interactions of timed and untimed system response [4]. This work is part of a larger effort to develop a Codesign Virtual Machine [1] through understanding of the design implications of programmable, concurrent systems with mixed real-time and untimed (performance-only) behaviors.

While [5] is focused on synthesis of application specific architectures, we are developing a means of relating logical and physical sequencing through development of virtual machine programmer's views. As a simulation environment, FI is not focused on analytical scheduling of real-time tasks [7]. Further, most of these modeling environments utilize ported models for all system components [14], which is not an appropriate model of the way logical sequencing is often resolved to physical sequencing through scheduling of partially ordered logical events that belong to a shared memory scope.

We utilize our simulation environment to discover the physical interactions of concurrent software and hardware. This paper lays a foundation for understanding how logical and physical sequencing will impact the design process, and provides insight into the problems that must be solved in such a design environment. We will continue to develop our formalism and design environment.

6. Conclusion

A central challenge in the development of new design methodologies for next generation programmable, concurrent systems is how to resolve the modeling abstractions traditionally associated with hardware and software into that of a system that contains both. In this paper, we show how relationships between logical and physical system timing can shed insight into the basis for existing design methodologies, as well their inadequacies to capture next generation designs. We motivate a design methodology based upon a layered virtual machine, which allows designers to consider the interactions of logical and physical sequencing as they form a system without too strongly coupling them or assuming that a system can be effectively designed when they are considered independently optimizable. Thus, we more effectively capture scheduler abstractions that resolve both correct logical sequencing and physical system performance by effectively managing a common view of shared resources.

7. Acknowledgments

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

8. References

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

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

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

[4] 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.

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

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

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

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

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

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

[11] 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, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, 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/