High-Level Modeling and Simulation of Single-Chip Programmable Heterogeneous Multiprocessors

JOANN M. PAUL, DONALD E. THOMAS, and ANDREW S. CASSIDY
Carnegie Mellon University

Heterogeneous multiprocessing is the future of chip design with the potential for tens to hundreds of programmable elements on single chips within the next several years. These chips will have heterogeneous, programmable hardware elements that lead to different execution times for the same software executing on different resources as well as a mix of desktop-style and embedded-style software. They will also have a layer of programming across multiple programmable elements forming the basis of a new kind of programmable system which we refer to as a Programmable Heterogeneous Multiprocessor (PHM). Current modeling approaches use instruction set simulation for performance modeling, but this will become far too prohibitive in terms of simulation time for these larger designs. The fundamental question is what the next higher level of design will be. The high-level modeling, simulation and design required for these programmable systems poses unique challenges, representing a break from traditional hardware design. Programmable systems, including layered concurrent software executing via schedulers on concurrent hardware, are not characterizable with traditional component-based hierarchical composition approaches, including discrete event simulation. We describe the foundations of our layered approach to modeling and performance simulation of PHMs, showing an example design space of a network processor explored using our simulation approach.

Categories and Subject Descriptors: C.4 [**Performance of Systems**]: *Modeling techniques, performance attributes*; I.6.5 [**Simulation and Modeling**]: Model Development—*Modeling methodologies*

General Terms: Performance, Design

Additional Key Words and Phrases: Computer-aided design, performance modeling, system modeling, schedulers, heterogeneous multiprocessors

1. INTRODUCTION

Virtually everyone agrees that heterogeneous multiprocessing is the future of chip design with the potential for tens to hundreds of programmable elements

This work was supported in part by ST Microelectronics and the National Science Foundation under Grants 0103706 and CNS-0406384. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.
Authors' address: ECE Department, Carnegie Mellon University, Pittsburgh, PA 15213; email: {jpaul,thomas,acassidy}@ece.cmu.edu.

on single chips within the next several years [IEEE Round Table Discussion 2001] . These chips will have heterogeneous hardware elements (the processors and interconnects) that lead to different execution times for the same software tasks that execute on them as well as heterogeneity with respect to the types of software. Some software will have fixed and limited performance most typical of embedded or real-time systems, while other software will be constrained only by the capacity of the underlying processing element(s) and communications channels most typical of desktop-style computing and servers. The processing elements will be coordinated by several layers of schedulers and will communicate with busses and even networks on chip [Benini and De Micheli 2002].

The complexity of both the new hardware capabilities and software applications will require designers to develop higher levels of modeling for efficient design. All successful modeling strategies eliminate excessive low-level design complexity while not over-limiting the design space. Cycle accurate register-transfer level (RTL) design, instruction set architectures (ISAs), and software objects, all captured the essence of what designers had already been doing in an elegant way. They identified *design elements*—features of a level of design abstraction that represented what could be changed about a design and what was assumed to be fixed. Lower-level details were eliminated until after many high-level, high-impact design decisions were explored and made.

We believe there are two fundamental challenges that need to be solved if this new level of design is to be enabled. First, new *design elements* will need to be defined that include new programmer's views, scheduling decisions, task mappings, communication strategies, and configurations of heterogeneous processing elements (PEs). Second, the resolution of these new design elements as *design layers* must permit meaningful performance modeling to take place in the absence of full model detail.

Performance modeling for programmable systems has traditionally been done at the ISS-level or below because it is the only accepted way of allowing the state update of the system-level program to be resolved to simulation time. However, not only is the execution time of ISS-level simulation far too slow for effective design exploration of future multiprocessor systems, but the level of detail required limits early design exploration; full software programs and processor models are currently required. For single chip systems, with on the order of ten processor resources, the simulation of full software models executing on ISS-level processor models is analogous to booting an O/S on a cycle-accurate simulation of a single processor—it can take many days to produce a single result. For single chip designs, there will be many design trade-offs to explore. Days must be turned into hours or minutes while still preserving meaningful performance-based simulation.

In achieving this, the move to levels above ISS is not straightforward. At this higher level of single chip design, complexity, design decisions about hardware resources, scheduling decisions, and software programs are all interrelated. Small design decisions about any of these can affect the state trajectory of the concurrent system and thus the accuracy of high-level performance models. The development of new simulation foundations that can resolve the key design concerns of software sequence to hardware sequence in the absence of

ISS-level detail while still preserving meaningful accuracy for highly concurrent heterogeneous multiprocessor systems is challenging but essential.

We refer to this higher level as Programmable Heterogeneous Multiprocessors (PHMs), emphasizing that not only will individual PEs be considered programmable, but the chip will also be considered programmable as a whole. The level includes layers of programming and thus layers of sequencing, where intra and interlayer sequencing all contribute to the overall performance of the programmable system. In this article, we describe the PHM level of design and the modeling foundation needed for it. We start by motivating the conceptual difference in high-level modeling when design elements are viewed as encapsulated and interconnected components vs. layers of scheduling where there is some globalization of elements in each layer. We argue that programming is sequencing *within* individual layers where layers are based on the globalization of access to collections rather than encapsulation. Sequencing between layers must then be resolved (eventually to hardware) for meaningful performance modeling to take place. Then we describe why existing simulation foundations fail to capture this layering above the ISS-level. We describe our work in a high-level, layered performance modeling environment, the modeling environment for software and hardware (MESH), and show how we modeled a network processor so that meaningful performance-motivated design decisions take place in the absence of full (ISS-level) detail.

While MESH is still in its early stages, this article contributes the foundation for MESH, motivating how a new level of performance modeling tools for programmable systems must be based upon layering instead of components if efficient designs will be discoverable in reasonable design time.

2. COMPONENT HIERARCHY VS. FUNCTIONAL LAYERING

Components are a natural way to model the two-dimensional spatial composition of physical systems where design elements are isolated and both encapsulated and interconnected by wire-like interfaces. Design elements for physical component-based computation systems include transistors, gates, and registers. However, software functionality does not naturally follow these rules of hierarchical containment and wire-like interfaces. Software directs how a collection of resources is scheduled in a global way, in contrast to the local scheduling of encapsulated components. This globalization permits data-dependent scheduling of shared access to system resources. Intuitively, sharing is the opposite of encapsulation, implying overlap and contention resolution rather than component isolation. In this section, we compare models that use a component-containment hierarchy with those that capture a layered design approach. Thus we motivate why it is important to capture high-level abstractions for layering in programmable system design.

2.1 Component-Containment Hierarchies

A component-containment hierarchy is shown in Figure 1 for three different levels. While a simple diagram, it illustrates several important concepts associated with component-based design. The components and interconnect contained

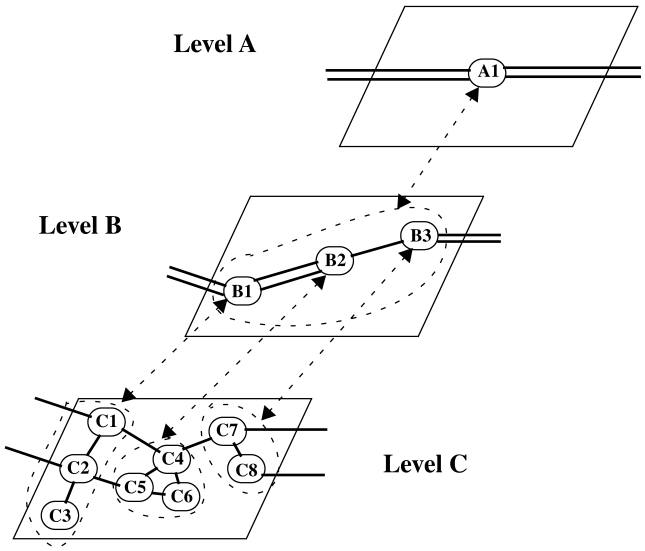

Fig. 1. Component-containment hierarchy.

in a level represent the *topological* organization of the design at that level of detail—level C in the figure might be Boolean logic gates while Level B represents three parts of the next state logic. When all of the components of a level match implementable elements, for example, components from a cell library, the logic design is complete. The higher levels in Figure 1 are abstractions to aid the designer; they provide the designer with less detailed views of the *entire* system. Thus, they are not a *part* of the implementation, but rather separate views of the system. The views between levels are related by rules for detail reduction and interface resolution, the most common of which is for physical, electrical systems where wires serve to bundle elements into components.

Components and their interconnections are commonly represented mathematically by graph models with vertices and edges. In the middle level (B) of the figure are three components, B1-B3. A component at any level is completely independent of the other components at that level. Its behavior is *contained* completely within the component. The only effects one component has on another are those communicated explicitly along the wires that connect the components. There are no side effects among the components. The encapsulation of functionality is one of the primary advantages of the use of components as an abstraction for physical design where wired interconnect is the basis for integration.

The second advantage of components as a basis for abstraction is the simplification afforded by using more-abstract (thus fewer) components than in the detailed levels. In physical models, the interconnect between the levels is preserved as depicted in the figure by the numbers of wires that are used for information exchange at the different levels. Each of the components of level B is composed of a nonoverlapping subset of the components in Level C. The

hierarchy is strict. A component on a lower level is a part of only one composition at the higher level. Because the common wire interface is preserved across levels in physical models, substitution of identical functionality using a different implementation is permitted and containment is preserved.

2.2 Functional Grouping by Layers

In this section, we contrast layered design with component-based design. Programmable systems such as uniprocessors that execute programs, networks which have programmable protocols at the top of a stack, and multiprocessors which support concurrent processes and threads are organized into design layers [Dijkstra 1968]. Each layer is a *logical* collection of functions (sometimes called services) and data objects that *support* the higher layers of the system. The services are globalized, that is, made available to the whole next higher design layer for scheduling as a collection. *Globalization* in the formation of a *logical collection* is the key difference between layered design and component-based design. Each layer may also contain private (encapsulated) state and may have the ability to schedule its services among multiple requestors at higher levels, but the formation of a collection of services is what distinguishes layers from components. Put another way, rather than integrate components with a prespecified interface abstraction (a wire), layers provide novel means of scheduling collections, even dynamically with decisions determined at runtime, as a part of the design of the system. In contrast to a strict hierarchy, more than one function in a higher layer may use a service of a lower layer in the same design; a scheduler must be provided to arbitrate how the service is accessed. Thus resources may be shared with sharing decisions determined at runtime. This layered view results in many features most commonly associated with programming including dynamic memory management, the sharing of registers, the ability to execute m tasks on n resources, and packet-based communications. By providing services, lower layers support the upper layers of design where schedulers become important design elements that resolve one layer to the next.

Consider a software program that requires floating point arithmetic. It may execute unchanged on a processor without a floating point unit implemented directly in hardware. The calculation can still be made by the fixed point processor but at a cost in overall system performance. The model is based on layering of software on hardware. Both program and processor combine to lead to overall system performance.

In contrast to hierarchical containment [Eker et al. 2003] where each level represents a different view of the entire system, the system implementation of a layered model is not complete without some representation from *all* of the design layers (i.e., their functions, state, and schedulers) working together. Full detail may not be required at each layer, high-level models may suffice—this is the view of the research described in Section 4. The important point is that all layers contribute to the overall physical system that results.

Although layering might appear to complicate a design by needing to represent all layers, it actually simplifies system design through functional *grouping*

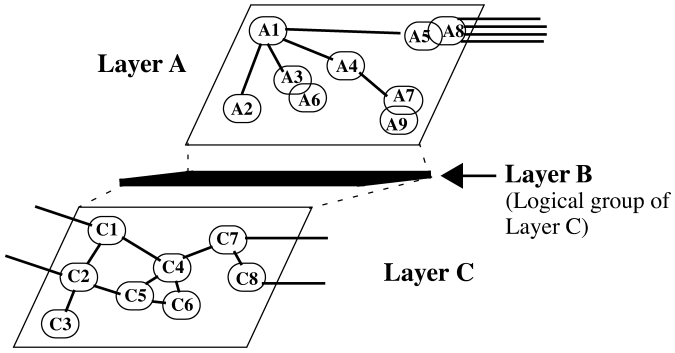

Fig. 2. Layered design—software on hardware.

into layers. Further, it permits more flexible use of resources because schedulers become design elements. For example, by grouping functionality into software function calls that encapsulate a service, that service may need to be provided only once in a lower layer. If the system can be scheduled to reuse this service without hampering overall required system performance, considerable savings can result. This ability to share resources, taken one step further, results in a unique form of design flexibility afforded by software programs; the actual scheduling of system resources can be determined at system runtime. Further, the more the higher layers are independent of the implementation of the lower design layers, the more the layers can be reused in many systems. However, this independence results in systems for which hard deadlines are more difficult to guarantee.

2.3 Forming Layers of Software on Hardware

A major feature of layered systems is in the flexibility of the scheduling they provide, including the ability to model data-dependent execution and to accommodate nondeterminism. Component-based, hierarchical hardware systems are scheduled when they are designed, meaning that for a given input vector, the sequence of logical operations and value propagations on wires are known. However, in layered software systems, the actual scheduling of entities in the upper layers is determined at system runtime. In these systems, there is scheduling state in the layers that selects what to execute next in response to the data that is presented to the system and nondeterministic coordination of system elements prevalent in multiprocessor systems such as bus access times and packet arrival times. A software program is a layered portion of an overall computation system, relying on hardware to carry out its functionality and affording the possibility that overall system behavior can be more flexible through runtime scheduling.

Consider a concurrent system in which M logical tasks are mapped to R physical resources ($M > R$). This is illustrated in Figure 2, where Layer A includes tasks A1-A9 which execute on resources C1-C8. Despite the fact that the lower layer of Figure 2 is the same as the lower level of Figure 1 several factors distinguish Figure 2 from Figure 1 . First, the design elements on Layer A can overlap. This captures the resource sharing modeled by software. For

example, the same function, service, or memory location may be utilized by many software tasks, resulting in the conceptual overlap of these tasks. Tasks that share services are not isolated from each other but may have side effects on execution sequence by the overlapping resources. These side effects may include actual functional calculations, overall system performance effects, or both. Second, the number of design elements in Layer A actually exceeds the number of design elements in the lower layer (C). This is common in many programmable systems where the overall functionality of a programmed system exceeds the functionality of the individual units in the processor(s) on which the software executes. Third, Layer A is not a more abstract view of Layer C but another portion of the overall system design. While Layer C can represent the physical hardware of a system such as a multiprocessor that has yet to be programmed, the overall programmed system can not be resolved to a single component-containment hierarchy.

Finally, there is an explicit contract between the design layers. Layer A is resolved to Layer C via a logical grouping of all of the resources provided by Layer C. The way design elements are grouped and provided to the upper layers is in the form of a contract such as a specification for how the lower layer is accessed by a software program, that is, an Instruction Set Architecture (ISA). This is shown on the diagram as Layer B. The relationship between layers is formed by a logical grouping of the design elements in the lower layer and the way they are scheduled. For single issue microarchitectures, Layer C is all of the resources provided by the microarchitecture including all support for computation, communication, and control. Layer A is the program. Layer B is the ISA that resolves the two layers. However, for concurrent systems, Layer B is another design layer which, in itself, can include state, computation, and control. In this case, layer B is a layer of scheduling and protocols that resolve logically concurrent computation entities to physically-concurrent computation entities. As such, it is an important part of a programmable, concurrent design often designed in concert with some knowledge of the end-application. In general, this resolution can be thought of as another layer such as a layer of scheduling which can serve as a bridge between the end-application and the underlying hardware which is key to performance-based design.

Note the contrast to Figure 1, where the contract between design levels is implicit—the wires preserve scheduling through each level of abstraction. By contrast, the explicit scheduling between design layers is shown on the figure as Layer B. Conceptually Layer A executes on Layer C as resolved by Layer B. Layer B provides the possibility for grouping the resources of Layer C in new and potentially dynamic ways. By contrast to the component-containment hierarchy of Figure 1, the layers of Figure 2 show that the higher layer of design is not constrained by the topology of the lower layer; this supports the observation that there may be more (and an even unbounded number of) software elements in Layer A than there are design elements in Layer C. This permits the numbers, types, and sequence of software design elements to be determined after the hardware is designed and even, in some cases, after the system is programmed. This flexibility results in a system that is actually more powerful than one described in a pure hardware design, graph-style with encapsulated components.

3. PROBLEMS WITH CURRENT SIMULATION FOUNDATIONS

Because they are based only on physical sequencing, continuous Time (CT), Discrete Time (DT), and Discrete Event (DE) [Zeigler et al. 2000] do not capture programmatic design elements that logically group resources. Programs are inputs to these models when they are used to capture low-level hardware and not a part of the model. In this section, we show how a layered execution semantic is fundamentally different from CT, DT, and DE by developing it from a fundamental event model and discuss how it resolves software to hardware timing without relying upon Instruction Set Simulators.

3.1 Foundations Based on Physical Time

Simulators of models based on physical time use *events* to maintain a time-ordered list of state updates for a system where events are coupled to physical time. More generally, an *event* has a tag and a value e = (t, v). The *value* $v \in V$, the set of all values in the system, is the result of a calculation. The *tag* indicates a point in a sequence of events where the value is calculated. Thus a tag may represent physical time or a less restricted ordering. The definition of an event model as a pair of data and time values is not new, and the notions that digital systems include both logical and physical sequencing [Seitz 1980] as well as partially- and totally-ordered sequences [Zeigler et al. 2000] are both well established. We adopt the nomenclature in [Lee et al. 1998] referring to a time value in an event tuple as a tag.

Threads are an ordered set of N events,

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

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

Computer systems contain two kinds of event ordering, logical and physical. The tags used in *physical ordering* represent a physical time basis; there is a real, physical interval of time between tags i and j when i *.neq.* j. Clearly, a physically-ordered system is totally ordered. The tags used in *logical ordering* specify a sequence which is not physically based. The basic difference between logical and physical ordering is that in logical ordering, the intervals, or difference between two events, do not relate back to physical intervals or global time. The magnitude of interval sizes between any two events in a logically-ordered system is undefined.

At a high level of system design, logical ordering often arises from functional modeling of the desired system behavior. The logical events are ordered according to a basic system time tick, but the tick has no meaning that relates to a physical design. The functional model merely prescribes an order to system events. For data-dependent, programmable systems, a basic assumption is that reordering the logical events of a thread (i.e., reordering the time tags) is allowable as long as data precedences are not violated. Thus, logical ordering can imply some design flexibility.

Assume that the thread event sequence,

$$\mathrm{Th} = \{e_1, e_2, \ldots, e_i, \ldots\},$$

is a high-level model. As yet, we will not denote this thread event sequence as being either logically or physically ordered, we will only describe it as a thread event sequence so that we can use it to compare and contrast a basic modeling difference between systems designed with logical vs. physical ordering.

Because the events of a high-level model typically represent a relatively large amount of functional advancement; we term them *macro states* or *macro events*. These macro events imply several other 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 rewritten as

$$\mathrm{Th} = \{(e_{11}, e_{12}, \ldots, e_{1j}, \ldots), (e_{21}, e_{22}, \ldots, e_{2j}, \ldots), \ldots\}.$$

Thus, each macro event, e_i, is seen to *contain* a sequence of micro events, e_{i1},e_{i2}, $\ldots$ 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—at the micro-level, the events used to model the gates within a register must complete before the register is considered to be updated at the macro level. Each micro event sequence in turn may contain another micro event sequence. This is the physical decomposition of a component-containment hierarchy which ultimately results in simple design elements that can be modeled with relatively simple functions.

The hardware design process resolves logical events to physical events by coupling or binding them at *design time* either directly in the design components that the designer manipulates or in the synthesis process. A logic synthesis tool binds Boolean algebraic functions which are logical events to gates which, when specified as library cell components, can be considered to be physical events. Synthesis tools make this design-time specification. Conceptually, the physical events have been scheduled at design time.

3.2 Performance Modeling

Software differs from hardware because, in general, it captures runtime decision making about how system resources are scheduled from the instruction to the thread level. System resources can include datapath elements, memory, processor resources, bus and network bandwidths. The runtime decisions can be affected by a variety of factors, including internal system data, system input, the speed and size of system resources, and the sharing of system resources by factors outside the control of the system designer. The dynamic, runtime sharing of the system resources leads to the need for modeling the performance of programmable designs.

The performance modeling of programmable systems is distinctly different from the modeling of systems for which performance is specified. One way to see this is to consider how timing is calculated by a simulation. We define a system to have *sequence-invariant physical timing* (SIPT) if physical time delays for a

calculation do not vary from one input set to the next regardless of any prior set of data presented to the system. Time delays for such systems can be accurately specified for a given set of input data without regard to execution history, that is, the data that has been previously presented to the system. A common example of this might be a synchronous finite state machine (FSM) designed using the process described previously. All of the processing is designed to take place within a well-defined clock period. Regardless of what input is presented to the FSM, timing analysis was used in design to determine that the processing of the next state and output functions will occur within the clock period. The FSM's performance can accurately be specified a priori.

Systems with *sequence-variant physical timing* (SVPT) cannot be modeled with fixed time delays for a given input set; the actual execution time is dependent upon the prior execution history of the system which is captured in its internal state. As a very simple illustration, consider a pipelined processor. Normally, the processor completes the execution of an instruction per cycle. However, when certain instruction sequences occur, a stall is required, slowing down the pipeline and thus the system's performance. Note that the processor's FSM still has sequence-invariant physical timing. But, when we consider the higher layers of design where the input is a data-dependent sequence of instructions and data, the processor's instruction processing has sequence-variant physical timing. The history of instructions interact to affect overall system performance with respect to the optimal case for which the pipeline is designed.

As an analogy, consider the way calculated data is contrasted for combinational logic vs. a finite state machine (FSM). Combinational logic is defined by calculating the same result regardless of prior sets of information presented to the logic. An FSM is defined as having state that affects the calculation of its outputs for a given set of input; prior information presented to the system affects the current calculation of system output for an FSM. Note that while an FSM is sensitive to prior execution history when it comes to data calculation, its timing is insensitive to prior execution history. It is actually an SIPT design style.

Combinational logic and FSMs result in distinctly different categories for design. Similarly, SIPT and SVPT result in different categories of design. SIPT systems tend to be hardware designs and embedded systems with fixed execution cycles. SVPT systems afford far greater flexibility by capturing nondeterministic interactions of system elements and data-dependent execution times within parts of the system. Both SIPT and SVPT systems can be simulated. However, performance modeling is required in order to capture SVPT designs.

The SIPT follows naturally from the logical and physical sequencing which is one and the same thing for physically-specified systems. While logical sequencing can imply a certain amount of design flexibility, including runtime decision making, that flexibility has been removed, and performance is fixed for hardware designs. While specification can result in a performance-optimized design, it can also require a large amount of upfront design time to explore the space for the optimal design as well as it can limit the responsiveness of the design to a wide variety of situations. Thus, programmable solutions have dominated most computer designs for the past several decades. However, performance modeling

for programmable SVPT designs have been limited to ISS-level models that resolve the actual sequence of instructions to a physical time-base supplied by a processor model. This level is becoming too detailed for meaningful design of complex multiprocessor programmable systems. Performance modeling of multiprocessor programmable systems must be rooted in physical modeling for meaningful timing information, while resolving the layered execution of software in which timing is not specified as a physical model. In the next section, we more fully explain why traditional simulation approaches only capture SIPT designs and are not adequate to capture high-level performance modeling of SVPT designs.

3.3 Widely Used Simulation Foundations

A variety of simulation foundations have been developed for the physical modeling of digital systems using the event model. The events are physical because the time tags are totally ordered, with sized intervals between any two time values. Simulators of physical systems use values of time that relate back to numbers, most typically the counting numbers. Physical models are required for performance analysis. However, the introduction of physically-sequenced events into the simulation semantic has traditionally forced software to be an input to the model. That is the only way logical and physical tags have been related in computer models. Effectively, this is resolving a logical model to a physical model by building a model of the actual computer system that resolves the layering.

Continuous time (CT) simulators are the most fundamental model of physical systems. CT simulators model differential equations using numerical integration techniques. State is advanced at evenly spaced time ticks; physical events tags have value nT where T is constant and n increments from 0. This tick approximates continuous time change in models of the physical world with discrete intervals. As T goes towards 0, implying ever-greater numbers of simulation cycles for the same amount of time being simulated in the continuous domain, the error goes towards zero as the model approaches an exact representation of the system being modeled.

Discrete time (DT) simulators model a system that is inherently discrete (as opposed to CT simulators) by selecting a time period T between which nothing of interest is presumed to happen in the system being modeled. When all state advancement is calculated in each time period (interval) of a digital signal processing (DSP) system, the system is exact and not merely an approximation. Time is therefore discrete and not continuous. Thus, if the complexity of the computation that occurs each cycle in a DT system does not exceed the physical capacity of a digital computer to compute it each and every cycle (accounting for numerical representation as well), the error is zero.

Discrete Event (DE) simulators differ from both CT and DT in that not all system state is recalculated for each simulation time interval. DE models are a faster means of executing CT or DT models if there are many elements in a system that do not update their local state each cycle which is the case for many digital system models. This is achieved by associating a time advance function

with the state advance function right in the model specification. Unlike CT and DT models, DE models associate time values with individual design elements that form the model as opposed to a global interval that applies to all system functionality. DE has evolved into a semantic, even resulting in unique design elements that arise not so much because they represent parts of real designs, but because they are consistent with design languages such as Verilog, VHDL, and System C [see http: //www.systemc.org/] .

The time advance function is captured in hardware description languages (HDLs) as a delay (e.g., #delay in Verilog) which designers use to annotate gate-level and functional-level blocks. These time values imply the physical interval required to execute the functionality. Thus, DE models inherently bind physical time with functional, logical specifications at the level of the individual design element. Performance is specified at the level of the individual design element. The time annotation of physical intervals forces the design elements of DE to carry out the functionality in the time specified regardless of the execution history of the rest of the system. When a DE system is presented with a given input dataset, the time required to calculate the result is not dependent upon the prior system inputs. The inputs propagate through the system elements at prespecified execution times until outputs are calculated. Thus DE simulations model systems with SIPT. By contrast, software is written to execute relative to the resources it is assigned to; it executes with SVPT. While additional software adds functional complexity to the system, it does not, at the same time, specify the physical machine to carry out the functionality. That is why most accurate performance models are at the ISS-level where software is an input to a physical model and otherwise not a design element of the model. Those that try to model software as a design element tend to specify its performance with time annotations in the same manner as a physical time-tag simulation foundation for hardware. This ultimately results in software serving as a specification for hardware—a physical model—instead of modeling something that executes on hardware.

4. A NEW SIMULATION FOUNDATION

In contrast to HDLs, general software languages do not provide physical time as a basic semantic construct in the language. Thus, software is modeled using logical events which specify logical precedence. The resolution of logical events to physical events does not occur until software runtime when a scheduler directs that the software execute. A simulation foundation must capture this resolution in order to model sequence variant physical timing. In this section, we describe the simulation foundation of Modeling Environment for Software and Hardware (MESH) [Cassidy et al. 2003; Paul and Thomas 2002; Paul et al. 2002].

4.1 Logical-To-Physical Event Resolution in Software

Consider a concurrent software system with at least two logical threads. Whereas two hardware physical threads are totally ordered, the logical threads are partially 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 resolution to a physical (total) order is the event sequence

$$\mathrm{Th} = \{\ldots, e_a, e_b, \ldots\},$$

while another correct resolution is

$$\mathrm{Th} = \{\ldots, e_b, e_a, \ldots\}.$$

Another possible resolution is that they will have the same physical tag; they will be concurrent, for instance in a multi-issue processor. Describing a system with a partially-ordered sequence allows greater flexibility in the design of the system; partially-ordered events give rise to alternate implementations of the system where actual concurrency and ordering can be determined at runtime.

Optimization of concurrent software executing on concurrent hardware requires the software to be considered with respect to its underlying machine—the nuances of how the logical events resolve to physical events. Two observations support this. As discussed before, adding or subtracting a thread (software process) or processor from a concurrent system does not tell the designer if the system will be faster or slower. The second is that relationships between containment and detail in hardware-like components are not preserved when modeling the logical on physical layering of software on hardware execution.

To see this, consider the decomposed thread event sequence from Section 3.1:

$$\mathrm{Th} = \{(e_{11}, e_{12}, \ldots, e_{1j}, \ldots), (e_{21}, e_{22}, e_{23}, \ldots, e_{2j}, \ldots), \ldots\}.$$

If we were assuming a physical ordering view, the ordering of events in the decomposed, more detailed micro-event sequence can be implied by the substitution on design elements. For instance, registers can be decomposed into gates which may, in turn, be decomposed into transistors. The execution sequence is preserved as detail is added to a design—the micro event sequence is contained, component-style, by the macro event.

However, hardware is not a more detailed model for software. Rather, software executes on hardware. For instance, a network may make a packet available to a waiting software task (modeled as logical events e_{22}, e_{23}, etc.), permitting the interleaving of the above events in the more detailed model as

$$\mathrm{Th} = \{e_{11}, e_{22}, e_{23}, e_{12} \ldots\}.$$

In this case, the parentheses that implies the containment is removed since the order on events is not preserved by containment. In a layered system, the event ordering—the runtime execution sequence—of higher design layers is affected by the execution sequencing of lower layers. Further, the execution of event e12 is delayed for an arbitrary period of time because events e_{22} and e_{23} were inserted in the execution due to an outside data-dependency.

4.2 Two Dimensions of Scheduling Required

We have seen that the physical time-tag foundations of discrete event simulation do not capture the execution of software because logical events are interpreted as physical events. Our approach to simulation has two dimensions of scheduling: that of the physical events based on integer-valued global

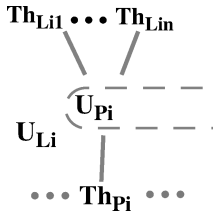

Fig. 3. A slice of the layering.

simulation time and that of self-timed scheduling of the logical events that are resolved to physical time by schedulers. This permits performance modeling of concurrent software executing on concurrent hardware, thus capturing systems with SVPT.

Concurrent, multithreaded software system design requires scheduling the logical threads on concurrent hardware resources. Consider a system with M logical threads of execution executing on R resources, where $M > R$. We define the physical event sequences in the system as a sequence for each resource as shown here:

$$\begin{aligned} &\mathrm{Th_{P1}} = \{\mathrm{e_{P11}}, \mathrm{e_{P12}}, ..., \mathrm{e_{P1t}}, ...\} \\ &\mathrm{Th_{P2}} = \{\mathrm{e_{P21}}, \mathrm{e_{P22}}, ..., \mathrm{e_{P2t}}, ...\} \\ &\qquad\vdots \\ &\mathrm{Th_{PR}} = \{\mathrm{e_{PR1}}, \mathrm{e_{PR2}}, ..., \mathrm{e_{PRt}}, ...\} \end{aligned}$$

In our notation, logical and physical thread sequences will have a base notation of $\mathrm{Th_L}$ and $\mathrm{Th_P}$, respectively. To the physical thread notation, we add a subscript ($\mathrm{Th_{Pr}}$) to denote the physical events of resource r, for $\mathrm{r} = 1, \ldots, \mathrm{R}$. The events are a totally-ordered sequence with time tag $\mathrm{T(e_{Prt})}$.

We also add to the logical thread notation to let $\mathrm{Th_{Lrm}}$ denote the logical thread m which is mapped to resource r. Each physical resource, r, can, in general, support M_r logical threads (M_r indicates r as a subscript of M). Thus, $\mathrm{M} = \mathrm{M\_1} + \cdots + \mathrm{M\_R}$. These threads are mapped by the resource's scheduler $\mathrm{U_{Pr}}$ to a physical thread $\mathrm{Th_{Pr}}$. Each $\mathrm{U_{Pr}}$ (as shown in the following) is a scheduling function that logically interleaves the M_r threads on resource r. M_r is typically unbounded.

$$\begin{aligned} &\mathrm{U_{P1}(Th_{L11}, Th_{L12}, ..., Th_{L1M\_1})} \text{ —> } \mathrm{Th_{P1}} \\ &\mathrm{U_{P2}(Th_{L21}, Th_{L22}, ..., Th_{L2M\_2})} \text{ —> } \mathrm{Th_{P2}} \\ &\qquad\vdots \\ &\mathrm{U_{PR}(Th_{LR1}, Th_{LR2}, ..., Th_{LRM\_R})} \text{ —> } \mathrm{Th_{PR}} \end{aligned}$$

The mapping described previously is illustrated in a high-level MESH diagram as shown in Figure 3. Note how the diagram emphasizes layered thread relationships as well as the overlap of information shared by multiple resources. Thus, it does not appear as a traditional component diagram where software on hardware is not captured.

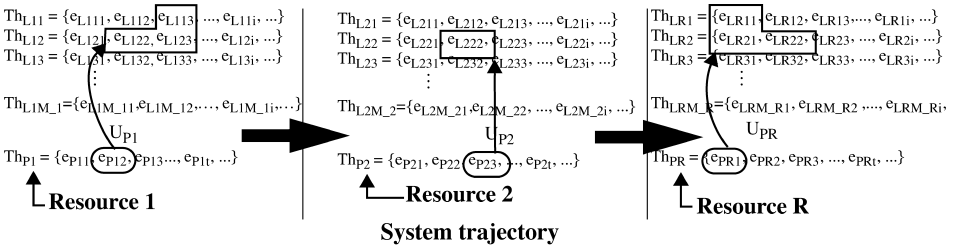

Fig. 4. The atomic groupings shown are executed left or right, as the sequence R_1, R_2, R_R is implied by the rate-based interleaving of the resources.

Logical events have no implication on physical interval sizes; logical event ordering in and of itself does not model performance. However, the schedulers map these logical events to physical events, thus capturing performance.

In general, the events of the threads Th_{Lrm} are grouped by the scheduler and assigned to execute on a resource. The current state and computational complexity of the functionality executed between logical events determines the order and extent of computation that executes in a given physical time slice on a given resource. In so doing, each scheduler on a resource r has access to the logical event sequences of M_r threads as shown.

$$\begin{aligned} Th_{Lr1} &= \{e_{Lr11}, e_{Lr12}, \ldots, e_{Lr1i}, \ldots\} \\ Th_{Lr2} &= \{e_{Lr21}, e_{Lr22}, \ldots, e_{Lr2i}, \ldots\} \\ &\vdots \\ Th_{LrM\_r} &= \{e_{LRM\_r1}, e_{LRM\_r2}, \ldots, e_{LRM\_ri}, \ldots\} \end{aligned}$$

Alternately, a logical collection of schedulers, mapped to individual resources, may be formed as

$$U_{Li} = \{U_{P1}, U_{P2}, \ldots, U_{PR}\},$$

allowing scheduling to be considered as a single, common, logical scheduling context across multiple processing resources with cooperative scheduling.

Now consider R concurrent resources, three of which (1, 2, and R) are shown in Figure 4. As physical entities, the resources can be interleaved by the relative interval sizes implied between the physical events; this is the time-based scheduling. Consider the one subsequence of physical events: $e_{P12}, e_{P23}, e_{PR1}$, as shown from left to right. (Note that the second subscript of the events (time t) represents time within the timebase of the physical thread; these are not global time tags. Global time is calculated from the relative rates of interleaving of the physical resources.) The scheduler of each resource selects logical threads to be executed in the physical time period implied by each event. Here, scheduler U_{P1} might select the events shown in the box: e_{L113}, e_{L122}, e_{L123}, implying a logical interleaving—resource sharing among threads—on Resource 1. (Each of these logical events could represent a large amount of software functionality.) This is the self-timed scheduling in our approach since decisions about logical event sequences are made in concert with data dependencies from data generated elsewhere in the system.

Considering only the three resources shown in Figure 4, the actual event sequence of logical threads in the example is

$$\mathrm{Th} = \{\ldots, e_{L113}, e_{L122}, e_{L123}, e_{L222}, e_{LR11}, e_{LR21}, e_{LR22}, \ldots\}$$

as implied by the boxed events. This sequence is the *system trajectory* or the actual execution sequence of the system's logical events over physical time.

The trajectory can be affected by many high-level factors such as resource rates, scheduling policies, and data dependencies. For instance, an increase in the computation power of a resource, say Resource 2, could allow it to execute an extra logical event as part of e_{P23}. The selection of which extra event to schedule is determined by U_{P2}. If this extra event, say e_{L233}, was being waited for by thread Th_{LR3} on Resource R, then scheduler U_{PR} might make a different scheduling decision, executing e_{LR31}along with other logical events on R instead of those shown in Figure 4. The resulting thread sequence with e_{L233} and e_{LR31} inserted would be

$$\mathrm{Th} = \{\ldots, e_{L113}, e_{L122}, e_{L123}, e_{L222}, e_{L233}, e_{LR11}, e_{LR31}, e_{LR21}, \ldots\}.$$

If the software in this system were modeled solely with DE-like physical time durations, its dependence on properties of individual resources would not be captured.

Our approach allows for the functional execution times, resource access delay, and communication delay to be calculated by the simulation where actual physical times are affected by the interaction of the system elements. This interaction results in internal system state which can affect the performance of the system over time without affecting its overall functional correctness; the system has SVPT. Put another way, the physical intervals required to execute logical functionality are not prespecified by designers as in DE simulations but calculated by the simulation based upon the systems' prior execution history and performance capabilities of the resources in the system.

Finally, the example in Figure 4 further illustrates how modeling based solely on hierarchical containment is inappropriate for PHM systems. Here, a change in Resource 2 affected a change in the logical scheduling on Resource R, violating the basic notions of component-containment. Significantly, it also illustrates the need to simulate both logical and physical scheduling. The actual trajectory of the system over time (its performance) is calculated by the simulation through the interaction of heterogeneous design elements with the separate (logical and physical) time bases of software and hardware.

4.3 MESH Simulation—An Operational View

Our system view (Figure 5) is a layered one, where the base level is the hardware architecture. On top of that, we model schedulers and protocols that give the top-level software tasks access to the underlying architecture. Heterogeneous, hardware resource models are shown on the left as Processors (P_i), memories (M_i), busses (B_i), hardware devices (H_i), and networks (N_i).

In the middle of Figure 5 is a layer of scheduler and protocol models that group resources in an overlapping manner, capturing both inter and intraresource sharing decisions. At the right is the concurrent software layer.

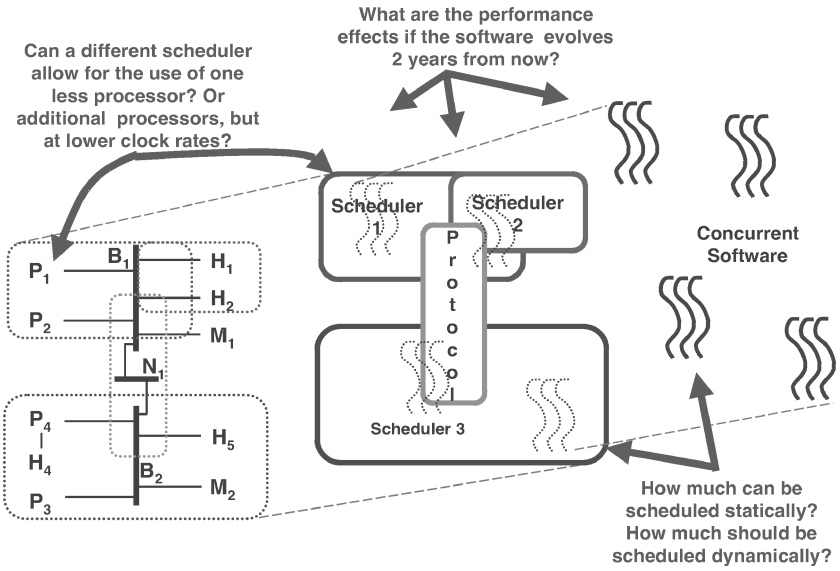

Fig. 5. A system-level design scenario.

(Many more layers may be in a programmable system.) MESH is unique in providing a layered modeling basis above ISS models and in using schedulers to model concurrent, high-level software running on high-level models of processor resources. The resolution of timing through design layers where unrestricted software executes on hardware models is what distinguishes our work from component-based, ported models that propose component-like software elements [Cesário et al. 2001; Cornea et al. 2003] and are focused on the development of a ubiquitous, wire-like interface to which all software components adhere and the separation of function and architecture [Keutzer et al. 2000]. Unlike approaches in which message-passing between components is optimized and resolved to platforms through middleware [Cornea et al. 2003], our modeling is more fundamental, making no assumptions about the structure of software components. Interestingly, the design of software components for just single processor systems can be problematic when it comes to performance modeling [Sitaraman et al. 2001], and a primary reason for using software components in the first place, reuse, has failed to capture anything like the reusability of hardware components [Glass 1998]. Further, our approach is not limited to application-specific systems where the application is assumed to be a given [Gharsalli et al. 2002] but captures the evolutionary nature of software applications. Our focus on the timing resolution of model elements distinguishes us from approaches that unite the timing models of separate simulation domains [Richter and Ernst 2002].

Rather, we focus on the simultaneous development of software and hardware as different design elements that can not be distilled to a single component-like model [Grattan et al. 2002]. Further, we agree that the performance and real-time demands of future single chip designs will necessitate the development of multiple heterogeneous multiprocessor architectures [Wolf 2003] that will be

semicustom and where the resultant parallel processors can only be effectively performance-optimized if the software is designed in concert with the underlying architecture [Karim et al. 2004; Skillicorn and Talia 1998]. Only our vision also includes schedulers as separate design elements that can optimize software execution on semicustom hardware [Paul et al. 2003].

In MESH, the schedulers also enable performance modeling by resolving the different timing of software and hardware. Schedulers serve multiple purposes in real programmable heterogeneous multiprocessing systems. At a local level, they select the logical thread which will have access to a physical resource. At a more global level, schedulers are layered to coordinate the activity of multiple resources. Thus, schedulers are important design elements that model resource sharing.

Figure 3 illustrates MESH's layered logical-on-physical relationship. A dynamic number of logical threads (the software, labeled as $Th_{L11} \ldots Th_{L1n}$) are shown at the top of the diagram. Their execution is scheduled onto a single resource (a processor, modeled as a physical thread, Th_{P1}) by a scheduler U_{P1}. The U_{Pi} threads are actually models of schedulers in the system that can make scheduling decisions based on the state of the threads being scheduled and other system state. This scheduling resolves the logical events of the software threads to physical timing. Thus the schedulers serve two roles: (1) modeling scheduling decisions and (2) resolving logical computation to physical time. Figure 3 shows a single vertical slide of a model—a complex system would have many resources (Th_{Pi}) along with their schedulers and logical threads.

The resource threads are scheduled to activate based on simulation time; at each timed activation, they provide resource power to their scheduler which then selects a logical thread to execute based on its state (e.g., is it waiting for data or not). U_{Li} threads can also logically group resources for inter-resource scheduling as shown by the dashed oval going off of the figure. This permits M threads to be dynamically mapped to N resources, for example, a pthread scheduler.

The key here is that the schedulers and logical threads use *consume calls* to resolve the logical computation of the software threads to the physical resource power provided by the resource thread. Consume calls enclose a set of logical events where these events do not exchange information between physically concurrent resources. The consume calls are inserted by the software writer and appear as callbacks to the scheduler. For instance, an annotation of "consume(9)" indicates that 9 computational units have been consumed by the software since the preceding consume call. The scheduler resolves this computational consumption to the underlying resource which is time based. The software thread may continue executing if the resource has more computation power left during its simulation interval, the thread is ready, and a more critical (shared) resource is not required to execute. Given a more powerful processor, it would execute more logical events per activation giving a different state trajectory.

An example of how consume calls work is illustrated in Figure 6. The figure shows the three levels of the MESH model as n logical threads, $Th_{Li1}, \ldots Th_{Lin,}$ which are interleaved to execute on a physical resource, Th_{Pi}.

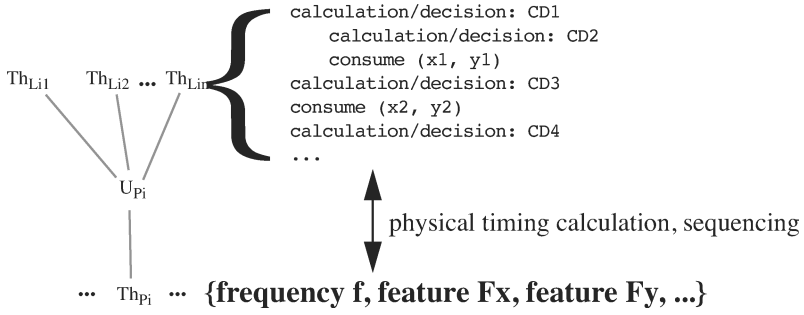

Fig. 6. Timing resolution illustration.

Many more physical resources are presumed to exist in the system. A single physical resource is shown in the figure in order to more fully explain the resolution of logical to physical sequencing within a given resource. The interleaving of the execution of the logical threads is controlled by scheduler U_{Pi} that resides on physical resource Th_{Pi}. The scheduler captures the logical decision making of which thread should execute on a given resource next, the same as a real scheduler would on a real processing element. However, in MESH, these same schedulers serve the additional purpose in the simulation environment of resolving logical complexity to physical capacity. This enables the logical sequencing to be performance-simulated with execution times determined by resource capabilities. In turn, the interleaving of physical resources captures the overall performance of the system because inter-resource information exchange is both data and resource capacity dependent.

On the right of Figure 6, lines of source code, abstracted as "calculation/decision" blocks, are labeled by number as CD1 . . . CD4. The calculation/decision blocks imply one or more lines of source code which may or may not include the possibility of decision making, that is conditionals such as if statements and loops. The inclusion of conditionals is what distinguishes CD blocks from basic blocks. MESH annotations are shown as consume calls which are functional calls to the MESH simulator. The consume call's parameters capture the amount of complexity between successive consume calls. Since consume call parameters can be calculated from runtime data in the simulation, they can represent the performance impact of conditionals and loops without having to annotate source code at the basic block level. The code between two successive consume calls is also known as a code fragment. For example, in logical thread Th_{Lin}, there are two consume calls with two different sets of parameters: consume (x1, y1) and consume (x2, y2). The first consume call states that x1 units of computational complexity associated with feature Fx on resource Th_{Pi}, and x2 units of computational complexity associated with feature Fy on resource Th_{Pi} have occurred since the preceding consume call in thread Th_{Lin}. Since the proceeding consume call has not been shown, it may be assumed that the top of the source code shown is either the top of the thread or the beginning of a new block of code to be annotated by consume calls. If this is the case, then CD1 and CD2 together imply x1, y1 units of computational complexity

as related to the features of resource Th_{Pi}. Similarly, CD3 implies x2 and y2 units of computational complexity as related to features Fx and Fy of resource Th_{Pi}.

The resource features can relate to whatever features are considered significant by the system modeler. MESH simulations have used consume calls that imply numbers of instructions which tends to work fairly well for RISC-style processors. However, the presence of processor features that separate instructions into categories necessitates the grouping of source code into styles that map to processor features. For example, the presence of a Multiply-Accumulate (MAC) unit on a RISC-style processor allows source code that can take advantage of the MAC unit to behave significantly better than if the MAC unit were not there. In this case, the complexity of the source code falls into two categories: general purpose instructions, and MAC-style instructions. This is but one example of the many that are possible as source code styles and resource features become new design elements. Significantly, the performance effects of CD blocks can be captured without the need to include full source code. For example, the presence of a computation that has locally data-dependent performance effects, but where the actual computed result does not otherwise affect decision making in the system, can be captured in a single line of code. One example of this is the time to calculate successive MPEG frames. The performance effects of the calculations are data-dependent but the calculations are not necessary to model if loading effects are being simulated at a high level. When MESH models are executed with full source code, they are typically two orders of magnitude faster than ISS-level simulations. In actual experiments, we have measured MESH runtimes on the order of seconds, while corresponding ISS-level simulations have been on the order of minutes. With this additional simplification, MESH simulations can execute even faster while they allow system-level designers to think in abstract terms that affect overall system performance.

Within a period of time defined by the frequency, f, of execution of Th_{Pi}, resource Th_{Pi} can carry out a fixed amount of computation. This amount of computation is captured by the X, Y portion of the tuple that defines the resource, also shown in Figure 6. Resource Th_{Pi} is powerful enough to carry out X units of computation of type Fx and Y units of computation of type Fy in 1/f unit time. Each resource in the system can execute the set of logical threads that are mapped to it so long as the amount of computation within the unit time implied by 1/f is not exhausted. If this is the case, then resource Th_{Pi} continues to execute logical threads or logical thread fragments until its execution budget is exhausted. Once the execution budget of a resource is exhausted, another resource in the simulation executes. Because the actual amount of computation that occurs within a period of simulation time is affected by data values in the logical threads, performance is calculated in MESH by resolving logical sequencing to physical sequencing, both inter and intrathread.

Significantly, this view is consistent with ISS simulations when processing resources are modeled at instruction or cycle-accurate rates, and software is instrumented at the assembly language level. Thus, our modeling basis has a path to physical design. However, the goal of MESH is to be able to explore designs well above the ISS-level.

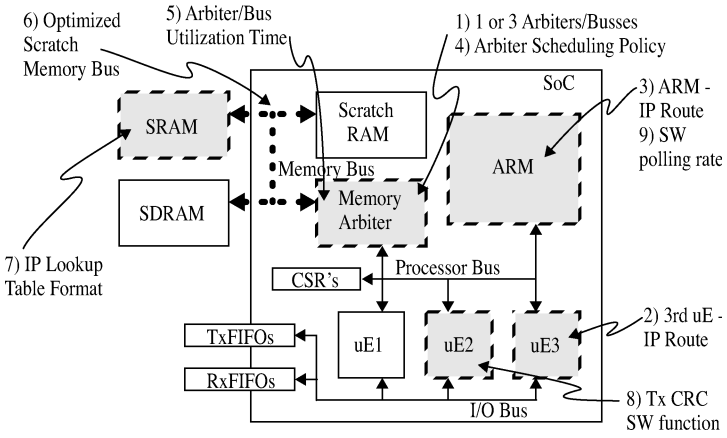

Fig. 7. Network processor system architecture.

5. NETWORK PROCESSOR EXAMPLE

We modeled and simulated a network processor to illustrate our approach [Cassidy et al. 2003]. Network processors are interesting in part because they have very simple functionality—they route packets from input ports to output ports. Functional models of network processors do little to help network processor designers. The art of network processor design is in the many performance trade-offs that lead to a rich design space. Performance modeling is required to provide network processor designers the ability to explore this design space rapidly so that an optimal design is more likely to be found.

Our approach is not limited to network processor designs. The usefulness of high-level performance modeling in any system with a rich performance-relative design space can easily be inferred. In MESH, designers can model the resource interactions and resource sharing that leads to SVPT which is required for performance modeling of programmable designs and designs in which resources are shared in a data-dependent manner. The MESH model simulates the effects of these interactions at system runtime under a variety of system loading and system configuration scenarios.

5.1 Design Space

The network processor, shown in Figure 7, is an embedded multiprocessor SoC. Chip multiprocessing (CMP) techniques are used to enhance system performance by processing control and data functions separately and further parallelizing the processing of data packets across several programmable cores. Many elements of the Example Network Processor (ENP) are based on the Intel IXP1200 network processor [Intel Corp. 2001]. Our ENP is composed of three microengines (uE's), based on the Intel IXP instruction set, a StrongARM processor, three types of shared memory, IXP1200 Control and Status Register (CSR) set, and a set of Tx/Rx FIFOs. The ENP is our *baseline design* around which we explored a broader design space using the logical and physical thread relationships described earlier.

We tested nine physical and logical design changes which are included in Figure 7.

(1) 1 memory arbiter vs. 3 memory arbiters
(2) 2 uE's vs. 3 uE's
(3) IP lookup on ARM vs. on uE
(4) arbitration: Hierarchical v. Round Robin
(5) memory arbiter utilization time: 8 vs. 4 cycles
(6) Scratch memory access time: 9 vs. 18 cycles
(7) IP lookup table format: 4 vs. 8-bit IP lookups
(8) with/without Tx CRC in transmit thread
(9) control port polling: fast vs. slow

By combining these modifications, a space of 512 different designs can be explored. Significantly, this set of design modifications includes changes in each of the layers of MESH, that is, software, schedulers, and resources.

Contention on the processor and memory busses is accounted for in the memory arbiter as well as base memory and bus latency modeled with a constant time value. Two different models of the memory arbiter were utilized, as discussed later in this section.

There are four primary software application functions running on the processing elements, a receive function, an IP address lookup-function, a transmit function, and a control port function. The IP address lookup-function performs a shortest prefix match-lookup on the packet destination address using the IP lookup table [Waldvogel et al. 1997] stored in SRAM. The control port function resides on the ARM while the other functions usually reside on two or three of the microengines. This creates a separation of the control and data plane and makes use of the heterogeneous processor resources.

5.2 Two Levels of Modeling

We developed an ISS-level model in order to validate the high-level MESH simulation results. The ISS model is composed of the actual multiprocessor system model, the application software, and cross compilers to generate binaries for the processor cores. The multiprocessor system model is derived from a GNU ISS model of the ARM processor. A processor model of the microengines is written in "C" as well as the memory arbiter, shared memories, FIFO's, the testbench, and other system logic. A GNU cross compiler is used to generate executable binaries for the ARM processor from "C" code. The cross compiler in the Intel IXP1200 Developers Workbench [Intel Corp.] is used to generate microengine binaries from IXP microcode (IXP assembly language).

Thus, we modeled and simulated the design space discussed in Section 5.1 at both the ISS-level as well as the MESH-level. The MESH-level models were considerably less detailed with source code annotated with consume statements as discussed in Section 4.3. In particular, the models of the programmable cores are simplified considerably. The ISS models a programmable core with detailed micro architectural functional units, such as the pipeline, instruction decode,

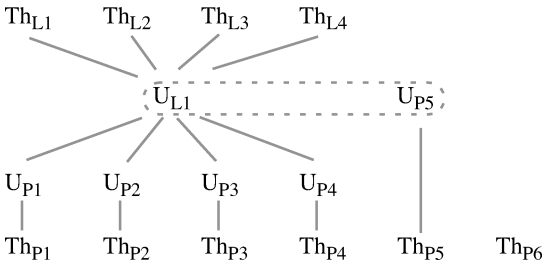

Fig. 8. Example thread relationships.

arithmetic units, and register file. In contrast, the MESH model does not actually execute the instruction stream. Rather, the application software is a logical thread executed by the simulator host processor. The consume calls capture the logical to physical timing resolution.

For these experiments, the software applications were instrumented by correlating the performance of the baseline MESH model with the baseline ISS model. This was done in order to validate the MESH model against the ISS model. A design methodology is currently under development that will allow the development of high-level models independently of this step.

Each high-level resource has a computational budget and a frequency of execution, representing the computational power of the processor. In this design example, the frequencies in the MESH model are chosen so that 16 clock cycles in the ISS equal unit frequency in the MESH model. Every programmable core runs at the same system clock frequency in order to match the ISS architecture, although this is not required to be the case [Paul et al. 2002]. The testbench and interface to the system are identical for both the ISS and the MESH model.

Using the notation of Section 4, the MESH simulation has thread relationships labeled as follows and shown in Figure 8. While a very simple diagram with only twelve thread relationships (future systems will contain hundreds if not thousands), four things distinguish the figure. First, the performance dependencies of software executing on hardware are emphasized. Second, common contexts which require resources to cooperate are emphasized. Third, the layered thread diagram is fundamentally different from a component diagram. Fourth, the thread types (Table I) are heterogeneous both inter and intralayer.

—Th_{P1}, Th_{P2}, Th_{P3}: rate-based models of uEs 1-3
—Th_{P4}: a rate-based model of the ARM processor
—Th_{P5}: a rate-based model of the memory arbiter
—Th_{P6}: a rate-based model of the system testbench
—U_{P1}, U_{P2}, U_{P3}: schedulers local to, uEs 1-3
—U_{P4}: scheduler local to the ARM processor
—U_{P5}: scheduler on the memory arbiter
—U_{L1}: logical scheduler; co-operation of U_{P1}-U_{P4}

Table I. Glossary of Threads Types

Th_{Lij}—One of j logical threads (software) that will execute on processor i.
Th_{Pi}—A Model of the ith physical resource in the system, such as a processor.
U_{Pi}—A scheduler that selects logical threads intended to execute on resource Th_{Pi}.
U_{Li}—A logical scheduler that can schedule M threads to N resources. For example, a pthread scheduler.

—Th_{L1}, Th_{L2}: Rx and Tx packet functionality

—Th_{L3}: IP routing functionality

—Th_{L4}: control port functionality

U_{L1} and U_{P5} are shown coupled because the memory arbiter cooperates with the logical scheduling of the rest of the system (as discussed later). Design changes 1, 2, 3, 6 involve adding or subtracting a physical thread (Th_{Pi}) from the system, changes 7–9 involve adding, subtracting, or modifying a logical system thread (Th_{Li}), and changes 1, 4, 5 involve modifying one of the scheduler threads (U_{Pi} and so U_{Li}).

5.3 Communications Modeling in MESH

For the network processor example, communications takes place on the bus. This is modeled in both the MESH and ISS-level models by the memory arbiter. In MESH, the memory arbiter is a common resource model and all resources connected to it belong to it. This is shown in Figure 8 as U_{L1}. In MESH, high-level models of communications are captured as penalties, dependent on the state and physical resource capabilities of the communications being modeled. Ideally, communication does not hinder processing. Busses with infinite bandwidth, caches that never miss, and networks that never drop packets appear ideal to the computation the rest of the system carries out. MESH captures this by modeling communications as a penalty. Shared memory access and message-passing forms of communication are modeled as a resource common to a set of other processing resources that hinders state exchange when there is an excess of demand placed upon its physical capacity (its bandwidth), when there is a programmatic choice made because the communications media must be shared and so conflicts must be arbitrated via some type of protocol, or both. In MESH, a protocol is a scheduler that arbitrates shared information exchange rather than shared access to a processor by a set of threads. Thus, communications is modeled as a common resource with underlying physical capacity and logical scheduling to which sets of other processor resources belong. A variety of communications models, in both level of modeling detail and communications type, are possible [Bobrek et al. 2004].

5.4 Functional Performance Model of Memory Arbitration

In the network processor system, a primary interprocessor interaction is the sharing of the memory resource. If one processor is reading or writing memory,

another must stall until the memory bus returns to the idle state. All accesses are officiated by the memory arbiter. In the ISS, accesses to shared memory are explicitly modeled in the hardware architecture, initiated by load and store instructions to shared memory regions. In the MESH model, shared memory accesses are instrumented in the software layer. Shared memory accesses make a function call to the memory arbiter. The high-level memory arbiter affects the consume budget of each processor, creating stalls during contention, emulating the performance behavior of the low-level system. Thus, we developed two models of the memory arbiter in order to establish a relationship between detail and accuracy for critical design elements.

The *functional performance model* of the memory arbiter determines an appropriate penalty to be levied on the access to shared memory. A simple first-order arbiter penalty function, based on the accesses in the current and previous high-level periods, models delay contention as a linear function based on the number of accesses in the previous high-level period (16 ISS cycles). Head-to-head contention is modeled with a simple step function. The first access in the high-level period executes with zero penalty and subsequent accesses in the same period have a uniform 1/4 period penalty imposed.

A more complex, but also more realistic, exponential functional specification was also developed. The behavior is more realistic as bus delay increases exponentially as a function of bus usage [Hennessy and Patterson 1996]. Both the delay contention and head-to-head penalty functions are specified as exponential functions. Both are based on the number of accesses in the previous high-level period as well as the position of the access within the order of accesses in the current period (for more details, see Richter and Ernst [2002]).

5.5 Design Exploration Experiments

We measured the average speed-up over the header match lengths for both the MESH and ISS models. The performance of the network processor design is evaluated based on the maximum number of packets per second the architecture is able to forward. All packets are 64-bytes in length, the worse case scenario. The packet destination address is adjustable in the testbench, and the lookup match length is determined by the routing table entries as well as the packet destination address. The binary design decision experiments were run across a range of fixed match length destination addresses. The design space exploration experiments were run with a random Poisson distribution of match lengths. Results were compared between the high-level MESH model with the first-order model of the memory arbiter, the high-level MESH model with the exponential functional performance model of the memory arbiter, and the ISS.

Table II summarizes the average and maximum error for the MESH with respect to the ISS model for nine design modifications (the first and last columns of Table II are discussed later). The results in Table II reflect the more detailed, exponential functional model of the memory arbiter. Of the nine experiments, seven have an average percent of error of less than 5%, and seven have a maximum percent of error of less than 5%. This error is due to limitations in the

Table II. Binary Design Decisions Using an Exponential Memory Arbiter Model in MESH

Design Change Index	ISS Avg. Speed-up	MESH Avg. Speed-up	Avg. % Error	Max % Error	Binary Decision Correct
1	9.5%	11.4%	1.9%	3.8%	Y
2	−4.1%	−3.3%	1.0%	3.5%	Y
3	−74.0%	−66.4%	27.3%	39.2%	Y
4	2.1%	0.8%	1.6%	3.7%	Y
5	8.6%	9.3%	2.1%	4.2%	Y
6	7.0%	8.7%	1.8%	3.5%	Y
7	21.7%	19.3%	1.9%	3.5%	Y
8	−0.7%	−0.4%	2.2%	4.0%	Y
9	−13.0%	−8.9%	5.0%	8.1%	Y

Table III. Binary Design Decisions Using a First-Order Memory Arbiter Model in MESH

Design Change Index	ISS Avg. Speed-up	MESH Avg. Speed-up	Avg. % Error	Max % Error	Binary Decision Correct
1	9.5%	8.4%	0.5%	3.0%	Y
2	−4.1%	−1.8%	2.9%	3.6%	Y
3	−74.0%	−66.7%	27.3%	34.7%	Y
4	2.1%	0.7%	1.2%	3.0%	Y
5	8.6%	5.4%	2.4%	5.0%	Y
6	7.0%	8.3%	1.9%	3.0%	Y
7	21.7%	17.5%	2.9%	9.2%	Y
8	−0.7%	+/−0.5%	1.6%	3.9%	???
9	−13.0%	−6.5%	8.1%	11.0%	Y

functional performance model of the memory arbiter and inaccuracies in instrumenting the software applications. Design change 3 has a rather large error because that change to the software application running on the ARM was instrumented by a completely uncalibrated guess by the designer. Interestingly, the designer initially thought the segment of code was unimportant in the larger system; we include the result to illustrate that sometimes designer intuition can be incorrect.

The comparison of the average speed-up over the header match lengths for both the MESH and ISS models using a less detailed, first-order functional model shows similar results in allowing the MESH model to characterize the design space. These results are shown in Table III . With the less detailed model, seven of the experiments have an average percent of error of less than 5%, and five have a maximum percent of error of less than 5%.

5.6 Binary Classification of Performance

Significantly, the error is never enough to recommend a design decision that negatively affects performance or vice-versa for either level of detail in the functional model of the memory arbiter when using a high-level MESH model. This binary or “thumbs up vs. thumbs down” classification of a design decision captures the essence of decision making in high-level modeling. So long as the error does not exceed the ability to make an informed decision, the high-level model permits the design space to be efficiently explored with considerably reduced model development and simulation time.

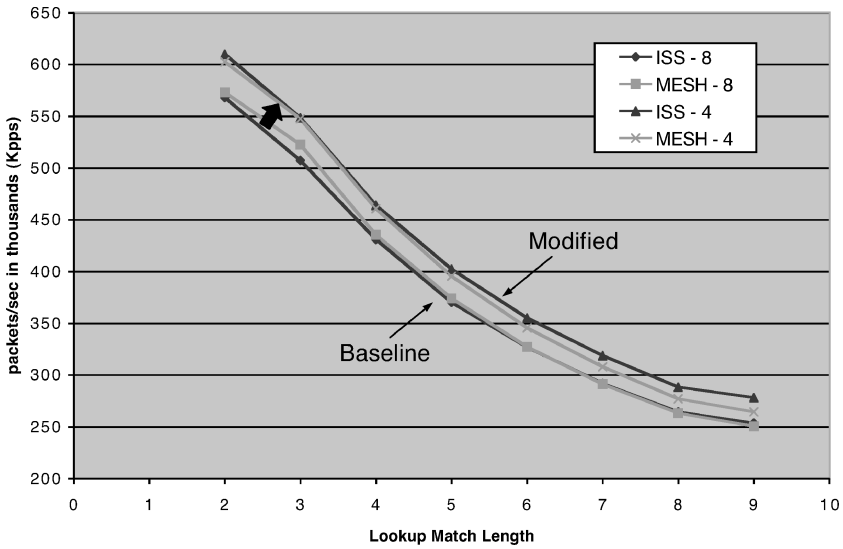

Fig. 9. System performance gain.

The binary decision, shown in the last column of both Table II and Table III, is made based on whether the speed-up of both models is in the same direction. Of the nine architectural modifications, all nine were correctly identified when using the exponential model of the memory arbiter. Eight of nine were correctly indicated by the high-level design when using a first-order model of the memory arbiter and the ninth was indeterminate. Design change 8, removing the transmit CRC function, shows an indeterminate result in Table III due to trying to model a fine-grained design trade-off at the high level. The effect of the change is too small to be captured, disallowing a binary decision to be made. Importantly however, both high-level MESH models allow designers to converge on the optimal or near-optimal designs in the design space of the network processor. These results are discussed next.

Each of the nine modifications listed were tested individually according to performance improvement or degradation, ignoring the magnitude of the percent of error. Modifications were made to both the MESH and the cycle-accurate ISS models in order to verify the results. Figure 9 is a representative result, showing the increase in performance as the bus utilization is decreased from 8 to 4 cycles per transaction (design change 5). The horizontal axis is 'Lookup Match Length,' a data-dependent parameter, and the vertical axis is the number of thousands of packets per second (Kpps) forwarded by the network processor. Visually, the performance of both models corresponds very closely as the lines nearly overlap one another. The lower pair of curves denotes the performance before the design modification, while the upper pair indicates the performance after the change. This shows an increased performance at each match length predicted by both models; the MESH and ISS curves shift together with the design modification.

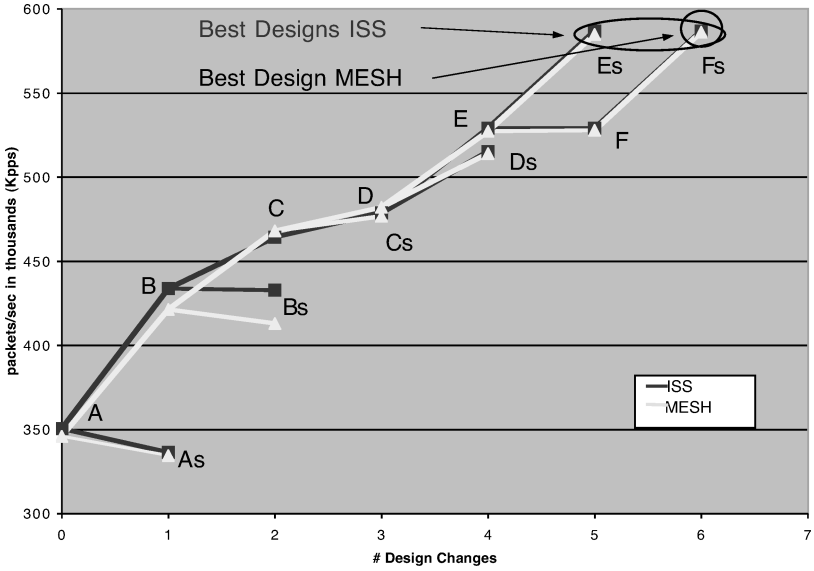

Fig. 10. Successive design changes.

Table II and Table III summarize all nine design modifications for each level of detail of the memory arbiter model according to whether the modification can be evaluated correctly in terms of performance improvement or degradation. The design change index is the modification as previously listed. The binary decision is made based on whether the speed-up of both models is in the same direction. When the binary decision is correct, both models agree that the design change either improved or negatively impacted performance. Significantly, these experiments show that correct design decisions can be made even with a substantial, absolute percent of error.

5.7 Design Space Exploration

The design space was then explored across successive design changes in the order of their expected speed-up ranking. We include results only for the exponential model of the memory arbiter and discuss the results for the first-order model of the memory arbiter. Figure 10 shows performance as design enhancements are added. The sequence A-F shows a particular design exploration path using two uE's. Points As-Fs were created along the same design path, except using three uE's (design change 2) instead of two. The graph shows the number of thousands of packets per second (Kpps) forwarded by the network processor vs. a design change step for both MESH and ISS. Adding the third microengine initially produces a performance degradation (also shown in Table II). However, in combination with other changes, such as increasing the number of memory arbiters, a performance improvement is seen. When using the exponential model of a memory arbiter, the MESH model predicts an optimal specific combination of five design modifications (Fs) which is one of the two optimal design change sets found in the detailed, low-level model (Es, Fs) as shown in

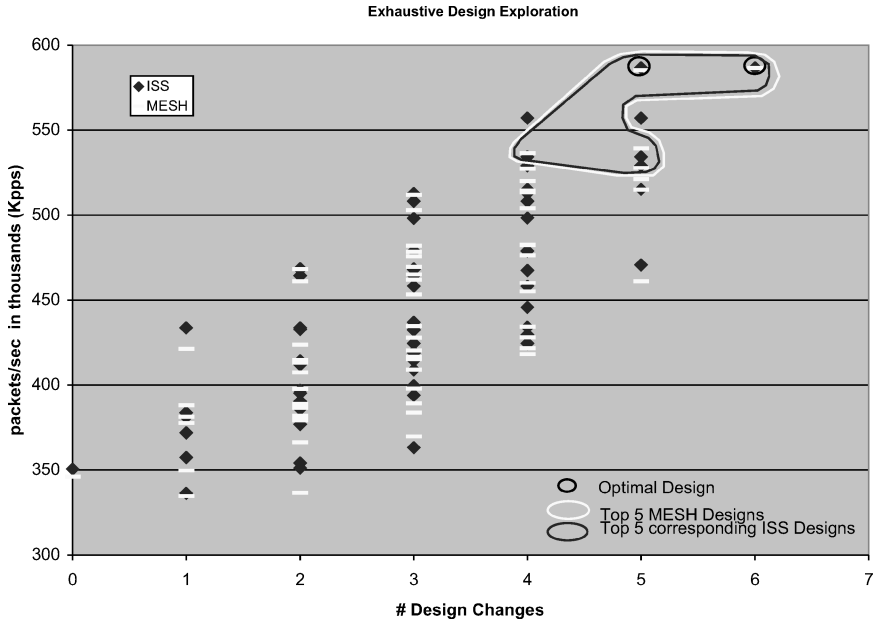

Fig. 11. Exhaustive design exploration.

Figure 10. When using a first-order model of the memory arbiter, the MESH model predicts the same specific combination of five design modifications (Fs) as the best overall design. Thus, the high-level model correlates to the underlying system that it represents, giving designers the ability to manipulate the coarse-grained, heterogeneous design elements in a broad design space in order to achieve a favorable system with near optimal performance.

Finally, we ran an exhaustive search of the design space as shown in Figure 11. Using six of the nine design parameters, 64 architectures were evaluated for both the first-order and the exponential model of the memory arbiter. The top five MESH designs select five of the top seven ISS designs for both the exponential and first-order memory arbiter model. Across all 64 design points, the average percent of error is only 1.8% and the maximum percent of error is only 6.7% for the exponential model of the memory arbiter. When using a first-order model of the memory arbiter, the top five MESH designs again select five of the top seven ISS designs despite the fact that the average percent of error for these five design points is 8.4%. This emphasizes that high-level approaches do not need to reach an absolute optimal design. Rather, after convergence upon a design region in a high-level design space, a more detailed design space can be explored with lower-level tools.

6. CONCLUSIONS

SoCs are becoming programmable heterogeneous multiprocessor systems with rich mixtures of concurrent software functionality executing on multiple hardware resources as directed by software schedulers. Practical design exploration of these complex systems requires high-level performance-informed

design with modeling and simulation well above the instruction set simulator level.

We illustrated how programmable systems require fundamentally different models from pure hardware systems. Thus the high-level modeling, simulation, and design of these systems will increasingly pose unique challenges, representing a break from traditional hardware and software design. Layered modeling for software was contrasted with component-containment for hardware models. From this basis, we defined and developed a layered approach to modeling and simulating SoCs. This approach represents a new category of design that focuses on capturing the interacting effects of the execution of layered concurrent software on concurrent hardware well above the ISS-level.

The layers capture the logical sequence base of software models along with the physical time base of hardware models. By defining a layer of scheduling between these bases, we provide an intuitive means of modeling how the execution of the logical events of a software model are resolved to the physical time events of a high-level hardware model. Interestingly, the intuitive notion of scheduling corresponds to a key design element in Programmable Heterogeneous Multiprocessor systems where a layer of programming resides on top of a layer of individually programmable elements. Thus our approach captures the fundamental aspects of PHM systems: software models, schedulers, and high-level hardware models.

We presented MESH which implements our layered modeling approach in a simulation environment where designers can model the performance of programmable designs and designs in which resources are shared in a data-dependent manner. We illustrated MESH by modeling and simulating a network processor where performance modeling is required to provide network processor designers with the ability to explore a rich design space. The MESH model simulated the effects of system runtime interactions under a variety of system loading and system configuration scenarios. The usefulness of high-level performance modeling in any system with a rich performance-relative design space can easily be inferred.

REFERENCES

BENINI, L. AND DE MICHELI, G. D. 2002. Networks on chips: A new SoC paradigm. *IEEE Comput. 35*, 1 (Jan.), 70–78.

BERGAMASCHI, R., BOLSENS, I., GUPTA, R., HARR, R., JERRAYA, A., KEUTZER, K., OLUKOTUN, K., AND VISSERS, K. 2001. Are single-chip multiprocessors in reach? *IEEE Design Test Comput. 18*, 1 (Jan.–Feb.), 82–89.

BOBREK, A., PIEPER, J., NELSON, J., PAUL, J., AND THOMAS, D. 2004. Modeling shared resource contention using a hybrid simulation/analytical approach. In *Proceedings of Design Automation and Test in Europe. 2* (Mar.), 1144–1149.

CASSIDY, A., PAUL, J., AND THOMAS, D. 2003. Layered, multi-threaded, high-level performance design. In *Proceedings of Design Automation and Test in Europe*. 954–959.

CESÁRIO, W., NICOLESCU, G., GAUTHIER, L., LYONNARD, D., AND JERRAYA, A. 2001. Colif: A design representation for application-specific multiprocessor SoCs. *IEEE Design Test of Comput. 18*, 5 (Sept.–Oct.), 8–20.

CORNEA, R., DUTT, N., GUPTA, R., KRUEGER, I., NICOLAU, A., SCHMIDT, D., AND SHUKLA, S. 2003. FORGE: A framework for optimization of distributed embedded systems software. In *Proceedings of the International Parallel and Distributed Processing Symposium*. 208–220.

DIJKSTRA, E. W. 1968. The structure of the THE-multiprogramming system. *Commun. ACM 11*, 5, 341–346.

EKER, J., JANNECK, J., LEE, E., LIU, J., LIU, X., LUDVIG, J., NEUENDORFFER, S., SACHS, S., AND XIONG, Y. 2003. Taming heterogeneity—the Ptolemy approach. In *Proceedings of the IEEE 91*, 1 (Jan.), 127–144.

GHARSALLI, F., MEFTALI, S., ROUSSEAU, F., AND JERRAYA, A. 2002. Automatic generation of embedded memory wrapper for multiprocessor SoCs. In *Proceedings of the ACM/IEEE Design Automation Conference*. 596–601.

GLASS, R. 1998. Reuse: What's wrong with this picture? *IEEE Softw. 15*, 2 (Mar.–Apr.), 57–59.

GRATTAN, B., STITT, G., AND VAHID, F. 2002. Codesign-extended applications. In *Proceedings of the International Workshop on Hardware/Software Co-Design*. 1–6.

HENNESSY, J. L. AND PATTERSON, D. A. 1996. *Computer Architecture: A Quantitative Approach*, 2nd Ed. Morgan Kaufman, San Francisco, CA.

INTEL CORP. 2001. Intel IXA Software Development Kit. Ver. 2.0.

INTEL CORP. 2001. IXP 1200 Network Processor Datasheet.

KARIM, F., MELLAN, A., NGUYEN, A., AYDONAT, U., AND ABDELRAHMAN, T. 2004. A multilevel computing architecture for embedded multimedia applications. *IEEE Micro. 24*, 3 (May–June), 56–66.

KEUTZER, K., NEWTON, A. R., RABAEY, J. M., AND SANGIOVANNI-VINCENTELLI, A. 2000. System-level design: Orthogonalization of concerns and platform-based design. *IEEE Trans. Comput.-Aid. Design. 19*, 12, 1523–1543.

LEE, E., AND SANGIOVANNI-VINCENTELLI, A. 1998. A framework for comparing models of computation. *IEEE Trans. Comput.-Aid. Design. 17*, 12, 1217–1229.

PAUL, J. AND THOMAS, D. 2002. A layered, codesign virtual machine approach to modeling computer systems. In *Proceedings of Design Automation and Test in Europe*. 522–528.

PAUL, J., EATEDALI, C., AND THOMAS, D. 2002. The design context of concurrent computation systems. In *Proceedings of the International Workshop on Hardware/Software Co-Design*. 19–24.

PAUL, J., BOBREK, A., NELSON, J., PIEPER, J., AND THOMAS, D. 2003. Schedulers as model-based design elements in programmable heterogeneous multiprocessors. In *Proceedings of the ACM/IEEE Design Automation Conference*. 408–411.

RICHTER, K. AND ERNST, R. 2002. Event model interfaces for heterogeneous system analysis. In *Proceedings of the Design Automation and Test in Europe*. 506–513.

SEITZ, C. L. 1980. System Timing. In *Introduction to VLSI Systems.* C. Mead, L. Conway. Eds. (Chapt. 7). Addison-Wesley, Reading, MA.

SITARAMAN, M., KULCZYCKI, G., KRONE, J., OGDEN, W., AND REDDY, A. L. N. 2001. Performance specification of software components. In *Proceedings of the 2001 Symposium on Software Reusability: Putting Software Reuse in Context 26*, 3, 3–10.

SKILLICORN, D. AND TALIA, D. 1998. Models and languages for parallel computing. *ACM Comput. Surv. 30*, 2 (June).

SYSTEMC. Available at http://www.systemc.org/.

WALDVOGEL, M., VARGHESE, G., TURNER, J., AND PLATTNER, B. 1997. Scalable high speed IP routing lookups. In *Proceedings of the ACM SIGCOMM '97 conference on Applications, Technologies, Architectures, and Protocols for Computer Communication 27*, 4, 25–36.

WOLF, W. 2003. How many system architectures? *IEEE Comput. 36*, 3 (Mar.), 93–95.

ZEIGLER, B., PRAEHOFER, H., AND KIM, T. 2000. *Theory of Modeling and Simulation*, 2nd Ed. Academic Press, San Diego, CA.

Received January 2004; revised November 2004; accepted December 2004