Layered, Multi-Threaded, High-Level Performance Design

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

Abstract

A primary goal of high-level modeling is to efficiently explore a broad design space, converging on an optimal or near-optimal system architecture before moving to a more detailed design. This paper evaluates a high-level, layered software-on-hardware performance modeling environment called MESH that captures coarse-grained, interacting system elements. The validity of the high-level model is established by comparing the outcome of the high-level model with a corresponding low-level, cycle-accurate instruction set simulator. We model a network processor and show that both high and low level models converge on the same architecture when design modifications are classified as good or bad performance impacts.

1. Introduction

System on a chip (SoC) designs are projected to become increasingly complex with the potential for hundreds of interacting programmable processing elements on a single chip. Key design decisions for these concurrent systems will focus on how coarse-grained hardware and software design elements interact to affect system performance. Simulations must allow designers to manipulate such heterogeneous design elements as the numbers and types of processing resources, interconnect strategies, software tasks, scheduler types, task mappings, and arbitration strategies so that near optimal designs may be reached.

For such systems, designers would readily trade accuracy for the ability to explore more quickly a larger, less detailed design space. Important at this level is getting close to the "optimal" design through high level trade-offs, and then switching to a more detailed level to further improve the design. Indeed, for many complex design spaces the optimal may never be known, but competitive advantage is gained by getting closer within a limited design time.

Figure 1 conceptually illustrates this as two levels of performance modeling in a design space. The bottom level represents the detailed *instruction set simulator* (ISS) level. The top level represents a high-level model. While the planes are flat for the purposes of illustration, in reality each of the levels would imply a multidimensional curve characterizing the relative performance of each individual model. As a designer makes a move along the high-level curve, a set of more detailed designs are implied in the more detailed curve. As illustrated, these projections may overlap, and the absolute error between the high-level point and its projection may be sizable at times. However, so long as the measurements in the high level model track those in the lower level, the high-level modeling can be used to narrow a larger design space more quickly than if complete detail were required.

A primary criticism of current high-level modeling is that it does not support meaningful, early design exploration. ISS approaches are too detailed, requiring fully developed software and hardware models. Many higher levels of modeling are purely functional, specifying required system behavior, but not capturing timing information necessary for performance evaluation. The designer cannot separate the system into high-level, concurrent software and hardware design elements and discover their performance as they interact in a modeled system. Instead, we focus on providing a simulation foundation for a performance modeling environment that captures software executing on hardware in concurrent, layered thread relationships. Our modeling environment, MESH (Modeling Environment for Software and Hardware)[1][2], provides a basis midway between functional and ISS models, allowing for early, high-level performance modeling without the need for the detail of instruction set simulators or complete software models.

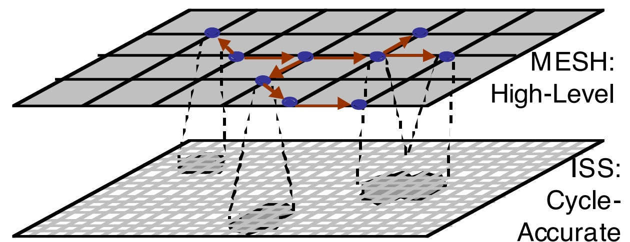

Figure 1 Experimental Design Exploration

We motivate our approach by showing how high-level software on hardware performance models must capture layering. Then we outline how logical *on* physical thread relationships capture high-level software on hardware execution for heterogeneous multiprocessing systems. Finally, we evaluate MESH in reaching a near optimal design over a set of heterogeneous design elements that include software, schedulers and processor resources.

2. High Level, Layered Performance Modeling

The design of concurrent systems with multiple programmable elements at the high level is fundamentally different than the design of systems with hardware elements at the high level, due to design element interactions. In hardware designs, high level models may be composed from groups of interconnected elements because the behavior of one group is not affected by the behavior of another group; component encapsulation holds and independence largely applies. Hardware design can be considered monolithic, even manipulated on a common graph basis.

In contrast, programmable systems are characterized by resource sharing. Shared system resources include busses, memories, buffers, queues, processors and networks, which are arbitrated by protocols, switches, arbiters, schedulers and memory controllers. Software and hardware design elements cannot be resolved to a single model [3]. Rather, the resource sharing requires a layered relationship on models — the

1530-1591/03 $17.00 © 2003 IEEE

upper layer is the common context across resources as illustrated by the programmer's view in Figure 2. The hardware model of the system, shown in the dark box, consists of multiple processors, P_i, with local memory, M_i, each connected to network N_1. The performance *potential* of the system increases as the number of processors. But only in this pure hardware sense does the system have independent design elements. The *actual* system performance is reduced from the potential speedup due to the interactions of the hardware and software elements in the system. The programmer's view of the system, shown as the grey arrow penetrating the hardware box, co-ordinates the otherwise separate hardware elements to work together on a common problem. The program is not an *equivalent* model of the hardware, as a high-level hardware specification of lower-level hardware might be, rather it is *part* of the programmed system, layered on top of the hardware model.

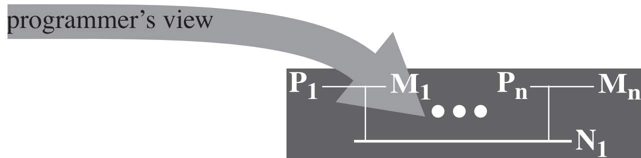

Figure 2 A Programmed, Networked System

The effects of the interactions among system design elements on overall system performance requires models that include a layering relationship of software executing on hardware. The challenge in high-level performance modeling lies in understanding what set of features are adequate to capture the interactions of the system's software and hardware design elements and how those features are affected by abstracting away detail. The step from low-level ISS modeling to high-level performance modeling is not straight-forward. Other high-level modeling approaches for heterogeneous multiprocessing have been focused on the integration of different event timing models [4] or resolving different models with component-like wrappers [5]. A formal, layered relationship on threads (described next) provides a basis for allowing the designer to manipulate design elements before components are formed or detailed timing models are established.

2.1 Thread Basis

We begin by summarizing our formal foundation for MESH which models systems using logical and physical sequencing of events [1]. 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 [7] as well as partially and totally ordered sequences [8] are both well established. We adopt the nomenclature in [9] referring to a time value in an event tuple as a tag. Our objective of high-level performance modeling and our layered, multi-threaded approach distinguish us from [9]. High-level performance modeling requires understanding how to model logical events as they are interleaved by and resolved to resources (processors) in a data-dependent manner when the resources are, themselves, interleaved in simulation time.

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. 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 [7]. The tags used in *logical ordering* specify a sequence which is not physically based; the ordering does not relate back to physical intervals or global time. Logical ordering often arises from functional language specifications at a high level of design. 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. The tags used in *physical ordering* represent a physical time basis; there is a real, physical interval of time between two tags *i* and *j* when $i \neq j$.

2.2 Logical-to-Physical Event Resolution

The means of resolving logical events to physical events impacts the design. Logical and physical thread sequences are denoted as Th_L and Th_P respectively. The thread

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

is ordered based on its 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 resolution to a physical order is the sequence

$Th = \{\dots, e_a, e_b, \dots\}$,

while another correct resolution is

$Th = \{\dots, e_b, e_a, \dots\}$.

Another possible resolution is that they will have the same physical tag; they will be concurrent. 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 can be determined at runtime.

The hardware design process specifies a resolution of logical events to physical events by binding them at design time, e.g., a logic synthesis tool binds Boolean algebraic functions (logical events) to gates (physical events).

In software design, the resolution of logical events to physical events does not occur until runtime. Clearly there must be a physical machine to execute the software, but the actual physical execution performance of the software is not part of the system model. From a design point of view, optimization of the logical ordering of a single-threaded software system (software design), can be done independently of the design of the physical system (processor design) upon which it will ultimately execute. In contrast, optimization of concurrent software executing on concurrent hardware requires the software design to be considered with respect to its underlying architecture. Adding or deleting a thread of execution or a physical resource does not necessarily give insight as to whether the overall performance of the system will be made better or worse. Performance modeling of concurrent software-on-hardware systems requires capturing the nuances of how the logical events will be resolved to the physical events.

2.3 The Role of Scheduling

MESH has two dimensions of scheduling: a time-based scheduling of the physical events, and self-timed scheduling of the logical events as they interact with each other in a

layer resolved to physical time by schedulers. This permits performance modeling of concurrent software executing on concurrent hardware.

Concurrent, multi-threaded software systems give rise to the need for 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

$$Th_{P1} = \{e_{P11}, e_{P12}, ..., e_{P1t}, ...\}$$
$$Th_{P2} = \{e_{P21}, e_{P22}, ..., e_{P2t}, ...\}$$
$$\vdots$$
$$Th_{PR} = \{e_{PR1}, e_{PR2}, ..., e_{PRt}, ...\}$$

Here we extend the thread notation of the previous section so that Th_{Pr} denotes the physical events of resource r, for r=1,…,R. The events are a totally ordered sequence with time tag $T(e_{Prt})$.

We also extend the logical thread notation to let 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, M = M_1 + ... + M_R. These threads are mapped by the resource's scheduler U_{Pr} to a physical thread Th_{Pr}. Each U_{Pr} below is a scheduling function that *logically* interleaves the M_r threads on resource r. M_r is typically unbounded.

$$U_{P1}(Th_{L11}, Th_{L12}, ..., Th_{L1M_1}) \longrightarrow Th_{P1}$$
$$U_{P2}(Th_{L21}, Th_{L22}, ..., Th_{L2M_2}) \longrightarrow Th_{P2}$$
$$\vdots$$
$$U_{PR}(Th_{LR1}, Th_{LR2}, ..., Th_{LRM_R}) \longrightarrow Th_{PR}$$

In general, the events of the threads Th_{Lrm} are grouped by the scheduler and assigned to execute at a resource's physical time. In so doing, each scheduler on a resource r has access to the logical event sequences of M_r threads:

$$Th_{Lr1} = \{e_{Lr11}, e_{Lr12}, \ldots, e_{Lr1i}, \ldots\}$$
$$Th_{Lr2} = \{e_{Lr21}, e_{Lr22}, \ldots, e_{Lr2i}, \ldots\}$$
$$\vdots$$
$$Th_{LrM_r} = \{e_{LRM_r1}, e_{LRM_r2}, \ldots, e_{LRM_ri}, \ldots\}$$

These are logical events with 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 modeling.

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

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

allowing scheduling to be considered as a single, common logical scheduling context across multiple processing resources. The individual physical resources owned by U_{Li} must each have their own local copies of a distributed, cooperative scheduler so that the common context is formed across the resources. For example, consider a pthread scheduler, distributed across R processor resources and scheduling M threads. While the pthread scheduler is a single scheduling entity, in reality it is a logical scheduling context, distributed across the R resources in a co-operative manner. The co-operation of the distributed schedulers is what allows M tasks to be mapped to R resources, where M and R are not known before runtime.

2.4 Performance from Layered Interleaving

Now consider R concurrent resources, three of which (1, 2, and R) are shown in Figure 3. 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 sub-sequence 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.) 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 all three resources, the actual event sequence of logical threads in the example is $\{\ldots, e_{L113}, e_{L122}, e_{L123}, e_{L222}, e_{LR11}, e_{LR21}, e_{LR22}, \ldots\}$ as implied by the boxed events. The actual sequence of logical events can be thought of as the *system trajectory.*

In Figure 3, the atomic groupings shown are executed left to right, as the sequence R_1, R_2, R_R is implied by the rate-based interleaving of the resources. 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}. 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.

Significantly, the actual trajectory of the system over time (its performance) is calculated by the simulation — not

$Th_{L11} = \{e_{L111}, e_{L112}, e_{L113}, ..., e_{L11i}, ...\}$
$Th_{L12} = \{e_{L121}, e_{L122}, e_{L123}, ..., e_{L12i}, ...\}$
$Th_{L13} = \{e_{L131}, e_{L132}, e_{L133}, ..., e_{L13i}, ...\}$
$\vdots$
$Th_{L1M_1} = \{e_{L1M_11}, e_{L1M_12}, \ldots, e_{L1M_1i}, \ldots\}$
U_{P1}
$Th_{P1} = \{e_{P11}, e_{P12}, e_{P13} ..., e_{P1t}, ...\}$
Resource 1

$Th_{L21} = \{e_{L211}, e_{L212}, e_{L213}, ..., e_{L21i}, ...\}$
$Th_{L22} = \{e_{L221}, e_{L222}, e_{L223}, ..., e_{L22i}, ...\}$
$Th_{L23} = \{e_{L231}, e_{L232}, e_{L233}, ..., e_{L23i}, ...\}$
$\vdots$
$Th_{L2M_2} = \{e_{L2M_21}, e_{L2M_22}, ..., e_{L2M_2i}, ...\}$
U_{P2}
$Th_{P2} = \{e_{P21}, e_{P22}, e_{P23}, ..., e_{P2t}, ...\}$
Resource 2

$Th_{LR1} = \{e_{LR11}, e_{LR12}, e_{LR13}, ..., e_{LR1i}, ...\}$
$Th_{LR2} = \{e_{LR21}, e_{LR22}, e_{LR23}, ..., e_{LR2i}, ...\}$
$Th_{LR3} = \{e_{LR31}, e_{LR32}, e_{LR33}, ..., e_{LR3i}, ...\}$
$\vdots$
$Th_{LRM_R} = \{e_{LRM_R1}, e_{LRM_R2}, ..., e_{LRM_Ri}, ...\}$
U_{PR}
$Th_{PR} = \{e_{PR1}, e_{PR2}, e_{PR3}, ..., e_{PRt}, ...\}$
Resource R

Figure 3 Atomicity Relationships

pre-specified in a static graph-like manner.

3. Design Exploration Example

A network processor serves as an example to illustrate how our approach allows the level of performance modeling to be raised above a cycle-accurate ISS [4]. The network processor, shown in Figure 4, is an embedded multiprocessor SoC, designed for the specific purpose of routing network packets efficiently. Chip multiprocessing (CMP) techniques are used to enhance system performance, processing control and data functions separately, and further parallelizing the processing of data packets across several programmable cores.

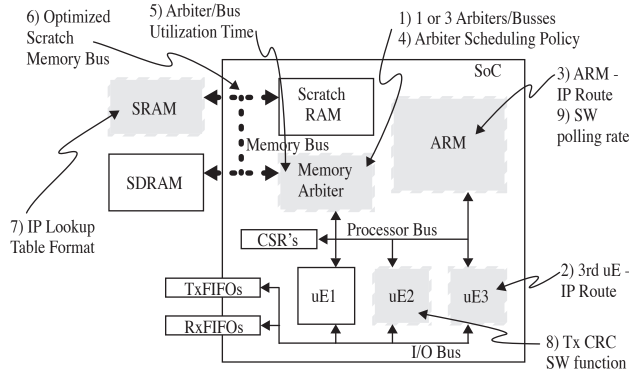

Figure 4 Network Processor System Architecture

Many elements of the Example Network Processor (ENP) are based on the Intel IXP1200 network processor[11]. 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 in section 2.

We tested nine physical and logical design changes which are also included on Figure 4:

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) memory arbiter schedule: 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-bit vs. 8-bit IP lookups
8) with or without Tx CRC function in transmit thread
9) control port polling: fast vs. slow

This set of design modifications includes change in each of the layers of MESH — 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. 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[10] 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.

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[12] is used to generate microengine binaries from IXP microcode (IXP assembly language).

Using the notation of section 2.1, the high-level simulation has thread relationships labeled as follows and shown in Figure 5:

- 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 microengines 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}
- Th_{L1}, Th_{L2}: Rx and Tx packet functionality
- Th_{L3}: IP routing functionality
- Th_{L4}: control port functionality

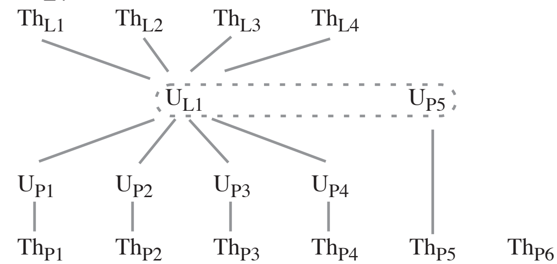

Figure 5 Example Thread Relationships

U_{L1} and U_{P5} are shown coupled because the memory arbiter co-operates 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}). The MESH model captures the same system behavior as the ISS with less detail. 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, arithmetic units, and register file. In contrast, the MESH model does not actually execute the instruction stream, but rather models the effects (i.e., behavior) of the instruction stream. The application software is a logical thread executed by the simulator host directly, instead of being compiled into a binary executable and run by the ISS processor model.

In order to tie the software application execution to a time base for performance modeling, the high-level software applications must be instrumented with 'consume' statements. These are callbacks to the schedulers indicating the amount of computational work a software function requires to execute relative to a high-level description of the computation power of the underlying resource[2]. The

scheduler tracks the software execution, monitoring the amount of resource processing power used. The software applications were instrumented by correlating the performance of the baseline MESH model with the baseline ISS model. Each high-level resource has a computational budget and a frequency of execution, representing the computational power of the processor. In our example, 16 clock cycles in the ISS are equal to unit frequency in MESH. 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[2]. The testbench and interface to the system are identical for both models.

In the network processor system, the primary shared resource causing interacting behavior between processors is the shared memory. If one processor is reading or writing memory, another must stall until the memory bus returns to the idle state. This access is determined by the memory arbiter. In the ISS, accesses to shared memory are explicitly modeled in the hardware architecture. In the MESH model, shared memory accesses must also be instrumented in the software application. The high-level memory arbiter affects the consume budget of each processor, creating stalls during contention, in order to capture the performance behavior of the low-level system. Thus the scheduling aspect of the memory arbiter (U_{P5}) affects the logical sequencing (U_{L1}) of the software application threads by ordering and delaying the shared memory accesses.

4. Design Exploration Experiments

These experiments focus on demonstrating a designer's ability to expediently explore a broader design space at the higher (MESH) level of Figure 1. 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.

Since the network processor is natively interleaved at the single ISS cycle rate, substantial error is introduced by reducing the interleaving rate. Consider two accesses to a shared resource by two processors executing at identical rates. Since contention results in access delay, the apparent access time varies according to the offset of the accesses with respect to each other. Two processors executing at the same rate with zero offset might experience a modeling delay up to that of the access period (assuming a fair arbitration scheme) — considerably more than would be in a real system. Conversely, if the processors are perfectly rate-interleaved with each other but are 180 degrees out of phase, a contentionless access pattern results. This is also not realistic in a performance sense.

In order to compensate for this error, a *functional approximation* of the memory arbiter was developed. Functional approximation increases state and functional complexity in the model in exchange for decreased interleaving. Functional approximation utilizes more system-wide information than would be available to the actual design element being modeled. The memory arbiter was calibrated across several design points of the ISS model. The results in this paper reflect the more detailed, exponential functional model of the memory arbiter. A less detailed, first order functional model yields similar results in allowing the MESH model to characterize the design space [6]. Future research will address more automated techniques for developing functional approximation.

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 1 Summary of Binary Design Decisions

We measured the average speed-up over the header match lengths for both the MESH and ISS models. Table 1 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 1 are discussed later). Of the nine experiments, seven have average percent error less than 5%, and seven have maximum percent error less than 5%. This error is due to limitations in the functional approximation of the memory arbiter and inaccuracies in instrumenting the software applications. Design change 3 has rather large percent error because that change to the software application running on the ARM was instrumented by a completely uncalibrated guess by the designer.

4.1 Binary Classification of Performance

Significantly while the error is sometimes significant, it is never enough to recommend an incorrect design decision. In this set of experiments, each of the nine modifications in section 3 were tested individually according to performance improvement or degradation, ignoring the magnitude of the error. Modifications were made to both the MESH and the cycle-accurate ISS models, in order to verify the results.

Figure 6 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

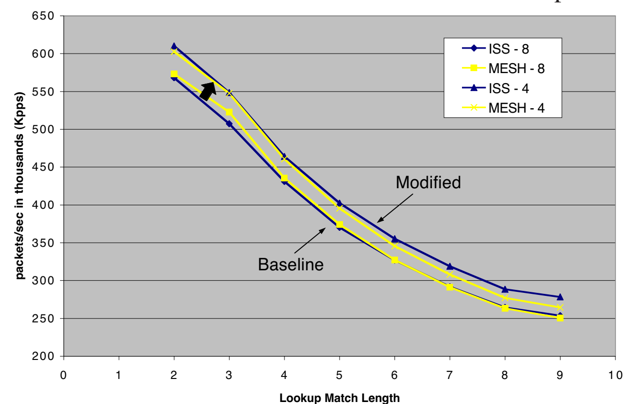

Figure 6 System Performance Gain

per second (Kpps) forwarded by the network processor. The MESH results are in yellow and the ISS results are in blue. 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.

Table 1 summarizes all nine design modifications according to whether the modification can be evaluated correctly in terms of performance improvement or degradation. The design change index is the modification as listed in section 3. The binary decision is made based on whether the "speedup" 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. All nine design modifications were correctly indicated by the MESH model. Eight of the nine modifications were correctly identified using the first order functional approximation of the memory arbiter (one was indeterminate) [6]. Significantly, these experiments show that correct design decisions can be made even with a substantial absolute percent error.

4.2 Design Space Exploration

The design space was then explored across successive design changes, in the order of their speedup ranking. Figure 7 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 (yellow) and ISS (blue). Adding the third microengine initially produces a performance degradation (also shown in Table 1). However, in combination with other changes, such as increasing the number of memory arbiters, a performance improvement is seen. 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).

Thus, the MESH 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 system with optimal performance.

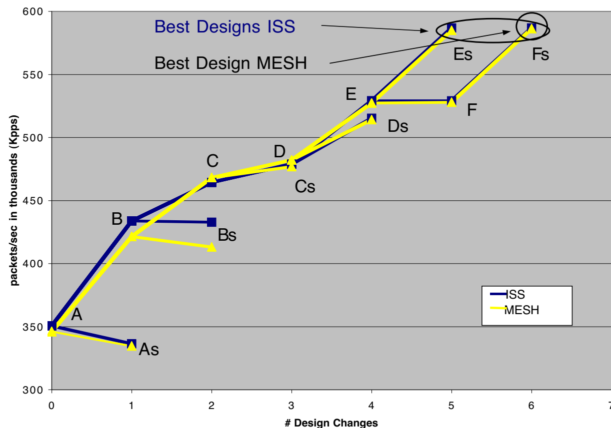

Figure 7 Optimal System Architecture

Finally, we ran an exhaustive search of the design space. Using six of the nine design modifications, 64 designs were evaluated. The results, not plotted, show a convergence upon a region of optimal system architecture. For each memory arbiter, the top five MESH designs select five of the top seven ISS designs. Across all 64 design points of the exponential memory arbiter, the average percent error is only 1.8% and the maximum percent error is only 6.7%. This emphasizes that high-level approaches do not need to reach an absolute optimal design. Rather, with convergence upon a region in the less detailed design plane of Figure 1, resulting designs can then be modeled in greater detail.

5. Conclusion

The design of SoCs with the potential for hundreds of programmable heterogeneous processing elements requires modeling early in the design process, enabling decisions regarding the performance impacts of coarse-grained hardware, software and scheduler design elements. Current approaches using functional modeling are too high level and do not capture performance; ISS models are too low level, requiring fully developed software. We defined our formal basis for layered software-on-hardware modeling as a general means for modeling the performance of these future SoCs. Using an embedded chip multiprocessor as an example, we showed that MESH effectively tracks heterogeneous design trade-offs when design changes are classified as good or bad with respect to performance. This approach allows designers to trade off accuracy for broader design space exploration.

6. Acknowledgements

This work was supported in part by ST Microelectronics and the General Motors Collaborative Research Lab at CMU. This material is based upon work supported by the National Science Foundation under Grant 0103706. 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. We thank Chris Eatedali for his suggestions.

7. References

[1] J. Paul, D. Thomas. "A layered, codesign virtual machine approach to modeling computer systems," *DATE 2002*.

[2] N. Tibrewala, J. Paul, D. Thomas. "Modeling and Evaluation of Hardware/Software Designs,"*CODES 2001*.

[3] B. Grattan, G. Stitt, F. Vahid. "Codesign-extended applications." *CODES 2002*.

[4] K. Richter, R. Ernst. "Event model interfaces for heterogeneous system analysis," *DATE 2002*.

[5] F. Gharsalli, S. Meftali, F. Rousseau, A. Jerraya. "Automatic generation of embedded memory wrapper for multiprocessor SoC," *DAC 2002*.

[6] A. Cassidy. "High-Level Performance Modeling and Design Exploration," CMU-CSSI 02-38 Tech Report. October 2002.

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

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

[9] E. Lee, A.Sangiovanni-Vincentelli. "A framework for comparing models of computation," *IEEE Trans. CAD,* Dec. '98.

[10] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. "Scalable high speed IP routing lookups," *SIGCOMM '97*.

[11] Intel Corp. IXP 1200 Network Processor Datasheet. Aug. '01.

[12] Intel Corp. Intel IXA Software Development Kit. Ver. 2.0.