

Modeling and Simulation of Steady State and Transient Behaviors for Emergent SoCs

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

Abstract

We introduce a formal basis for viewing computer systems as mixed steady state and non-steady state (transient) behaviors to motivate novel design strategies resulting from simultaneous consideration of function, scheduling and architecture. We relate three design styles: hierarchical decomposition, static mapping and directed platform that have traditionally been separate. By considering them together, we reason that once a steady state system is mapped to an architecture, the unused processing and communication power may be viewed as a platform for a transient system, ultimately resulting in more effective design approaches that ease the static mapping problem while still allowing for effective utilization of resources. Our simulation environment, frequency interleaving, mixes a formal and experimental approach as illustrated in an example.

Keywords

Hardware/Software Codesign, Computer System Modeling and Simulation, System on Chip Design

1. Introduction

Emergent system on chip (SoC) designs are fundamentally different from other computing systems for the combination of three primary reasons. The first is that they contain multiple, heterogeneous *clock domains*. Clock domains are an abstraction of the physical limits of synchronous design — beyond these limits, computation and communication must be partitioned among multiple interacting domains. These domains give rise to custom hardware devices, heterogeneous processor types, and communication links (busses and networks). Secondly, their complexity requires that designers consider system wide effects of anticipated hardware and anticipated software over the lifetime of the product. That is, architectures and software evolve over time. The third reason is that the systems will contain a mix of real-time and untimed behaviors. The notion of the purely reactive, embedded system is disappearing. Increasingly, untimed, desktop-style functionality is being integrated on embedded computers.

Computer systems can be thought of as consisting of two types of behavior — that which is steady state, and that which is transient or non-steady state. The behavior of a *steady state* system is fixed with respect to an external, absolute time reference. For systems with hard real-time deadlines the most significant time reference is typically determined by the response time of the system to sets of inputs. The system must compute a bounded amount of work within

a bounded amount of time in reaction to a time-bounded set of inputs. Over a system period, the system can be thought of as having a fixed or steady-state behavior. In contrast, the behavior of a *transient, non-steady state system* is not designed to external time constraints nor finite computation. Rather they adapt their behavior to unknown execution times, aperiodic input arrival times, and internal state via dynamic scheduling techniques. These behaviors are sometimes called untimed because the system timing is internal and not tied to an absolute, external reference. Because these untimed forms of system behavior are being mixed with hard real-time behavior, effective SoC design requires architecture, schedulers, and functionality to be designed together.

Steady state system design has been dominated by formal analysis — these reactive systems are statically analyzed as a task mapping problem ensuring that hard deadlines will be met for tasks with periodic or sporadic arrival times for a given set of resources and task execution times [12]. The givens in such schedulability analysis include the set of tasks to be scheduled, their deadlines, their execution times on the hardware, the scheduling algorithm, and the hardware resources upon which they will execute [13]. Transient system design has been dominated by simulation and benchmarking — analyzing performance under a representative set of operating conditions often with the objective of improving the relative balance of system processing. For this, performance enhancements are proposed and their designs are then explored via simulation-based experimentation.

Systems that are designed to only a steady state reactive paradigm rarely attain a perfect match between processing power and load — leftover processing power will result. However, if the leftover processing power, that in excess of the power needed to execute the steady state functionality, can be utilized for non-steady-state processing, the system resources may be over-designed intelligently without waste. A motivation for designing systems with both steady state and transient behavior is that transient system applications scale more effectively with the processing power of their platforms because they are not limited to the processing of externally time-bounded inputs. Thus they effectively take advantage of Moore's Law.

When designing steady and non-steady state digital systems, three distinct *design styles* dominate: *hierarchical decomposition*, *static mapping*, and *platform-based*. A *hierarchical* style is traditionally associated with hardware design, and implies a self-consistently partitionable and composable specification of resources, where detail is filled in by sub-models without destroying the hierarchy of the higher-level system.

A *static mapping* design style arises from a separation into two distinct design domains. The most pertinent example at this level of design is that of functional tasks and processing resources, or function and architecture. The resultant designs in each domain must be resolved (mapped) to the other for the whole system to be formed. The mapping is static and is motivated by real-time, physical constraints placed on the overall system behavior.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISSS'01, October 1-3, 2001, Montreal, Canada.

Copyright 2001 ACM 1-58113-418-5/01/00010...\$5.00.

A *platform-based* system is constructed by an application layer that directs a service or resource layer. The layering permits resource-layer details to be hidden from the application layer, and for the resource layer to be designed independently of higher-levels of the system model. In computer systems design, the platform layering might consist of a processor providing resources to a scheduler (e.g., an operating system — O/S), which in turn provides resources to the tasks it schedules.

We formalize how a mix of these distinct design styles will be needed to support the design of emergent systems and motivate how design approaches that blend formal analytical and experimentation-based simulation techniques are needed. Further, we will show how effective future SoC design requires that the function, architecture and scheduling of the system be considered concurrently in support of a mix of steady-state and transient processing. This is a step not considered previously in the codesign of reactive systems (e.g. [5]). We show how our modeling environment, frequency interleaving, is a basis for our approach.

2. Function, Architecture, and Scheduling

An architecture, $A(t)$, is a weighted graph of resources with a set of N processing nodes, and J communication arcs.

$$A(t) = (\{P_1(t), \dots, P_n(t), \dots, P_N(t)\}, \{B_1(t), \dots, B_j(t), \dots, B_j(t)\})$$

Each processing node has power $P_n(t)$, and each communication arc has bandwidth $B_j(t)$; all are a function of time (t).

System functionality is a weighted graph $F(t)$ of M computation threads and D sets of information exchanged between them.

$$F(t) = (\{L_1(t), \dots, L_m(t), \dots, L_M(t)\}, \{E_1(t), \dots, E_d(t), \dots, E_D(t)\})$$

Each potentially concurrent thread of execution represents a loading $L_m(t)$ on a computing resource, and a demand $E_d(t)$ on communication resources; all are a function of time (t).

$F(t)$ is the net computation of the system and not the behavior required to construct a system. For instance, schedulers are a set of behavior, $U(t)$, specifying how a set of computational threads is shared by a resource. Together, the functionality, architecture, and schedulers are mapped by Y to form a system $S(t)$ as in

$$(S(t), X(t)) = Y(F(t), A(t), U(t)) \quad \text{Rel 1}$$

Y implicitly requires consideration of system-wide effects while the $U(t)$ schedulers are local to system resources. This is a key difference between a static mapping and a platform-based design style. A platform-based style permits local scheduling decisions to be made in the absence of system-wide detail.

Rel 1 also defines the result of the mapping of the functionality, architecture and schedulers as a tuple, $(S(t), X(t))$. $S(t)$ is the *net behavior* of the system. $X(t)$ is the *remainder architecture*. When the computation power of the original architecture, $A(t)$, exceeds the computation demand of the functionality mapped to the system, $F(t)$, and the schedulers, $U(t)$, a net amount of processing and communication power $X(t)$ is extra in a system. The original system architecture is transformed by this mapping and scheduling to another (the remainder) architecture which is potentially capable of carrying out additional behavior.

$A(t)$ and $F(t)$ also have an associated amount of storage. Since our focus is the temporal response of systems and not their storage requirements, the memory/state model is omitted for brevity.

2.1 Steady State Systems

A steady state system is designed to respond to its externally time-bounded I/O, conceptually performing a bounded amount of work over a time. Steady state systems result from either a hierarchical hardware decomposition or a static mapping design

style. We define a pure *hierarchical hardware decomposition* system as a 1:1 correspondence between tasks and architecture:

$$((L_i \Leftrightarrow P_i), (E_j \Leftrightarrow B_j)) \forall (i, j)$$

In these systems, $M=N$, $J=D$ and load perfectly matches computation demand. For example, a hardware state machine and datapath provide the exact power needed to implement a task represented by load L . Because of the exact match, $X(t)=0$.

A *statically mapped system* is one where $M>N$ — the number of tasks in the system exceeds the number of computation nodes. Consider a system for which $M=2$ and $N=1$. The two tasks must be mapped to a single processor resource; a scheduler is needed. Given the time-bounded characteristic of the system, there is a finite period, T , over which the system's behavior is conceptually constant. More formally, the mapping of systems with $M>N$ requires a scheduler, U , itself a function of time, that maps the computation loads onto the processors; here scheduler U_1 resolves our two tasks to a processor:

$$U_1(t, L_1(t), L_2(t)) \rightarrow P_1(t)$$

Schedulers are viewed as an entity mapped to the processing resource, while tasks are mapped to the schedulers. In general, schedulers can be null for single tasks mapped to individual resources; the scheduling mechanism for the resource is the task itself. In this way, $M>N$ tasks are resolved to N resources, implying N schedulers as

$$\{L_1(t), \dots, L_M(t)\} \rightarrow \{P_1(t), \dots, P_N(t)\} \Rightarrow \{U_1(t), \dots, U_N(t)\}$$

Rel 2

Over the steady state system period, T , behavior may be considered constant for Q steady-state tasks as shown below.

$$\{L_1, \dots, L_Q\} \rightarrow \{P_1, \dots, P_N\} \Rightarrow \{U_1, \dots, U_N\} \quad \text{Rel 3}$$

The time-bound T effectively acts as a barrier within which sets of computation must be completed, but also to which faster internal processing is wasted. A *purely* steady state system can do no more computation than the processing required in the system period. Thus, for purely steady state systems, when processing power is increased, the system does no more work. More formally for a steady state mapped system

$$(S(T), X(T)) = Y(F(T), A(T), U(T))$$

when $A(T)$ is increased, $F(T)$, $U(T)$, and $S(T)$ remain constant, and leftover processing power $X(T)$ results. The most conservative design of steady state systems is when $X(T)$ is large, while the most efficient system in terms of resource usage is when $X(T)$ is small. Rather than trying to match $A(T)$ and $F(T)$, it can be more productive to understand how to utilize $X(T)$ for transient system processing. We discuss how transient systems achieve this next.

Although we have considered only computation nodes and the computation complexity of tasks without regard to information exchange in the architecture, consideration of a mapping of information exchange to communication channels follows the same line of reasoning.

2.2 Transient Systems

The design goal for transient systems is a *platform* for which a time-independent range of applications, task durations, input sets, and internal state will have acceptable performance. In transient systems, the platform, functional processing, net processing and extra processing power may all be functions of time. Thus, one important goal may be to minimize $X(t)$ over all t . In contrast to steady state systems, net processing, $F(t)$ is not bound to the availability of externally time-bounded inputs. An increase in

architectural power, $A(t)$, could result in a corresponding net increase in $F(t)$ since the system can process more information so long as there are inputs or internal state to process — this is architecture-power scalability. Although perfect scalability is limited, at least some scalability results for many applications.

Transient systems are constructed from the *platform design style* as a generalization of the task mapping problem. Schedulers map K non-steady state tasks to processor resources as

$$U_n(t, L_1(t), \dots, L_K(t)) \rightarrow P_n(t) \quad \text{Rel 4}$$

For some transient systems, K may be dynamic. For example, an unknown number of server threads may be spawned to service separate requests. Or, the system may contain early notions of anticipated functionality, which may not be finalized until well after the system is shipped. The essence of platform design is that the task set is presumed to be larger than may execute over some period of time. The scheduler dynamically composes the system functionality at runtime in processing inputs and internal state.

2.3 Merging Steady and Non-steady State

Emerging SoC computing systems will include both steady and non-steady state behavior. While resources may be shared in both types of systems, the scheduling of the sharing in a steady state system is matched to its external, temporal processing requirements. Resource sharing is based on time periods and functionality is never resource starved over T . In contrast, scheduling of functionality in a transient system is based first on the availability of data, then on the availability of resources — transient functionality may be resource starved.

The response of transient processing is measured as execution time after a data-dependent start time, or latency. The data- and resource-dependent progress of a non-steady state task, k , is in general measured over an interval $\delta_k = [s_k, s_k + i * r_k]$. s_k represents a start time since the start of system execution, or relative to $t=0$. In general, s_k may be tied to some other data-dependency. r_k is a rate typically drawn from some *internal* system reference, such as a processor frequency or communications bandwidth. The length of the interval is $i * r_k$, constructed as an integer-valued number, i , of execution times of the reference rate.

When a designer is interested only in the relative progress of a set of data-dependent tasks and not latency, s_k can be arbitrary and taken to be zero. δ_k then becomes an *observation window* for task k of the effects of data, resource power, and task scheduling interactions across a number of resources. While T captures an external constraint as a data-independent interval during which a set of steady state tasks is *guaranteed* to execute, δ_k is a *designer's reference*. Indeed, a designer may choose to observe the cumulative progress of one or more tasks over a number of successive δ_k , simplifying the need to consider the complete execution history of the system, since when $s_k=0$ and $i=\infty$, $\delta_k = (0, \infty)$ for the range of t . This also allows the designer to design systems relative to *internal* system performance references. In general, for a system with K non-steady state tasks, K observation windows exist in a set $\Delta = \{\delta_1, \dots, \delta_K\}$, one for each transient task.

Mixed steady-state and non-steady state systems are constructed when schedulers can allow for a partitioning of the utilization of the system resources. Thus, for mixed digital systems

$$\{F(T), F(\Delta)\} \Rightarrow \{U_1(T, \Delta), \dots, U_N(T, \Delta)\} \quad \text{Rel 5}$$

The mixed steady state and non-steady state system behaviors F are formed by schedulers local to each resource. These form the guaranteed, steady-state system mapping along with non-steady-state, data-dependent resource sharing. Relation 6 combines

relations 2, 3, and 5 for an architecture with constant processing power in a system with M tasks (Q of which are steady state and K of which are non-steady state) scheduled by N schedulers, each of which resolve steady state and transient functionality.

$$\begin{aligned} \{L_1(t), \dots, L_K(t), L_{K+1}, \dots, L_M\} &\rightarrow \{P_1, \dots, P_N\} \\ \Rightarrow \{U_1(T, \Delta), \dots, U_N(T, \Delta)\} &\quad \text{Rel 6} \end{aligned}$$

Thus, the most effective means of scheduling the individual system resources is first with respect to all of the steady-state processing (i.e., T) and then with respect to a designer's reference which captures the relative execution of all transient tasks in the system on the system resources on which they are executing, using the system-wide set, Δ . Note that while $Q=M-K$ is bounded in relation 6, K may be dynamic and unbounded for dynamic multitasking.

The key design consideration for general concurrent computer systems is captured in relation 6. While resource sharing scheduling decisions are made by schedulers local to resources (the U_n), the most effective resource sharing scheduling decisions for a set of data-eligible tasks minimizes data starvation of the tasks in other parts of the system. Some global system context is required in each local resource. While relation 6 represents a simplification over the transient response of the entire system over all time, t , additional simplification is possible.

The effective set of observation windows required for consideration at each U_n is clearly much smaller. We define τ_n , as a *scheduling scope*, permitting simplification of the system-wide set of observation windows required for effective resource scheduling decisions at resource n . Each τ_n may be a set of intervals in

$$\begin{aligned} \{L_1(t), \dots, L_K(t), L_{K+1}, \dots, L_M\} &\rightarrow \{P_1, \dots, P_N\} \\ \Rightarrow \{U_1(T_1, \tau_1), \dots, U_N(T_N, \tau_N)\} &\quad \text{Rel 7} \end{aligned}$$

where we also allowed the steady state scheduling considerations of each local scheduler to be designed to a set of local periodic references, T_n .

A scheduling scope must permit at least some degree of knowledge about the flow of data outside of its scope, in addition to having adequate knowledge about the most effective use of heterogeneous resources within its own scope. Clearly, this requires an explicit form of scheduling that simultaneously takes into consideration functional data dependencies for data-dependent task activation along with architectural resource capability for effective resource sharing — thus, the concurrent design of function, architecture, and schedulers.

We are defining the basis of a design environment which will allow the designer to more effectively design the schedulers $U_n(T_n, \tau_n)$ for the way function, architecture and schedulers interact in the formation of a layered platform system. Clearly the complexity of such interactions requires a simulation and design environment that simultaneously captures system functionality, architecture and schedulers, modeling the internal behavior of the system along with its net response.

3. Frequency Interleaved Simulation (FI)

Our design environment [2] uses a textual specification to capture a layered system model composed of software models, software scheduler/protocol models, and resource models. It includes *frequency interleaved simulation* targeting integration of steady-state hardware and mapped design styles with non-steady state platform-based design. It introduces pure software models into a simulation environment, allowing for unrestricted modeling of the software and hardware virtual machines.

All computation in FI [2] is modeled by two fundamental thread types — C and $G(F)$. C and $G(F)$ threads execute in a shared

memory modeling environment, from which communications can be explicitly modeled or inferred from the manner in which inputs are read and outputs are generated among individual threads or thread groupings.

C threads are rate-based threads that continuously sample inputs and generate outputs at fixed rates or frequencies (f_i), regardless of any other type of data events such as changes on inputs. Thus their activation is guaranteed. Since each C thread potentially has a different frequency, the activations are interleaved in time.

C threads capture the steady state response of traditional hardware resources, including the cycle accurate models of processors, busses and ASICs. C threads are, however, a more general modeling abstraction than cycle accurate hardware models. They can be used to model finer details of rate-based hardware descriptions [11], thus permitting a basis for partitioning a cycle accurate hardware model.

The most significant difference between C threads and other modeling environments with multi-rate capabilities [16], including HDLs, is that C threads provide the resource basis for a platform style of design. C threads are the foundation of all scheduling in the system, just as real hardware resources provide a foundation for all system execution.

C threads provide this capability by supporting G(F), or *guarded functional* threads. In general, G(F) threads can be activated by any other thread in the system. However, some specific types emerge (discussed later). Used to model software, G(F) threads:

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

We also define G(C) threads which are scheduler threads; they are guarded by a C thread and schedule G(F) threads. We also define G(T) threads which are guarded by a real-time constraint. Together, C threads and G threads form the basis of mixing hierarchical decomposition, static mapping, and directed platform design styles for simulation and analysis of steady-state and transient system responses.

3.1 Thread-based Design Abstractions

Consider the simple system of Figure 1 which represents two busses interconnected by a network. Each bus has a single processor, one or two memory mapped hardware devices, and some global (to the bus) memory space. This architecture was selected as it and Figure 2 will be used in a later example. (Thus, this system is meant as an illustration. It can be extended to include an arbitrary number of processors and hardware devices per bus, and an arbitrary number of busses and networks.) Each of the processor resources, hardware devices, and communication channels could be a *conceptual clock domain*. In this view, the system resource model

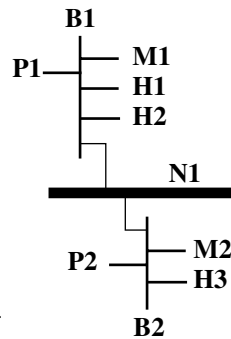


Figure 1 A System Architecture

contains eight clock domains. Thus $A = (\{P\}, \{B\})$ where $P = \{P_1, H_1, H_2, P_2, H_3\}$ and $B = \{B_1, B_2, N_1\}$.

Each of these clock domains allows independent manipulation of its power P or bandwidth B to translate state from inputs to outputs. Each is modeled as a separate C thread in FI. Not shown on the purely architectural diagram of Figure 1 are the software threads and software schedulers important to the static mapping and platform-based design styles. Figure 2 shows these threads as G(C) threads above the C threads. We draw the set of C threads that models the system resources at the bottom of the figure, labeled P_1 , and P_2 , associated with clock domains (or frequencies) f_1 , and f_6 respectively. These are also scheduling domains because they provide the basis for functionality that executes on the C thread resource, loading its resource power. Note that we have left B_1 and B_2 out of Figure 2; to simplify the example, we assume their execution rates are sufficiently large compared to other rates in the system and thus do not slow the processing.

Scheduling domains, or U_n are labeled in the FI scheduling diagram as $U_1(T_1, \tau_1) = G_{11}(C)$ and $U_6(T_6, \tau_6) = G_{61}(C)$. The first subscript denotes the unique clock domain to which the scheduler is mapped. The second subscript is I , implying that the only function which may activate the scheduler is the underlying C thread itself. Thus, $G_{61}(F) = G(C_6)$, where C_6 is clock domain 6.

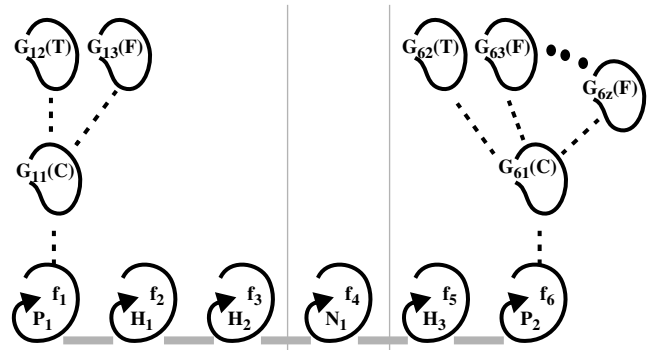


Figure 2 A Frequency Interleaved Schedule

A *platform* is the coupling of a scheduling domain to an underlying C thread, as illustrated in Figure 2 and defined by relation 6. Outside of the schedulers, all other G threads are activated by schedulers. Since the system depicted in Figure 2 is following an example presented later, a G(F) and G(T) thread are bound to clock domain 1 (the processor on Bus 1). An unbounded number of additional threads are bound to clock domain 6, the processor on Bus 2. This can be captured as in relation 6 as

$$\{G_{13}(F), G_{63}(F), \dots, G_{6z}(F), G_{12}(T), G_{62}(T)\} \rightarrow \{P_1, P_2\} \\ \Rightarrow \{G_{11}(C_1) = U_1(T_1, \tau_1), G_{61}(C_6) = U_2(T_2, \tau_2)\}$$

Thus, G(F) threads model transient forms of processing with dynamic functional and data dependencies, G(T) threads represent steady state behaviors scheduled by a statically mapped scheduler [1], and G(C) threads are scheduling domains.

Comparing our approach to others, none allow the designer to reason directly about the resulting interactions and system-level characterizations of independently manipulable models of resource power, resource loading, and resource sharing captured by relation 2 of our model. Further, some current high-level system design methodologies ignore the modeling of the hardware [3], limit software to hardware-like finite models of computation [5], or apply synthesis to a restricted portion of the design space [6][7][8][9]. Others view all system scheduling as being either hardware-like or single processor RTOS-based [10], or they

resolve all modeling to either token-based encapsulated computation and communication [4][14], or gate-like discrete event scheduling [15].

3.2 G(F) Thread Time Resolution

All execution in an FI simulation is resolved back to a set of C threads each with its own frequency defined relative to the others. The frequencies are normalized with respect to the smallest and inverted producing time periods. Note that some threads are based on physical timing data, giving the simulation a real time basis.

G(C) threads that schedule other G(F) threads are *scheduling domains*. The underlying resource C thread provides *power* P to the scheduling domain thread, which in turn selects which of the G(F) threads should execute and keeps track of how much load L it put on the resource. If one G(F) thread finishes and there is still power left for others, execution continues with one of them. Finally, the scheduler returns to the C thread which will reactivate later as specified by its calculated period.

Schedulers have a variety of means of resolving functional execution to a time budget. For instance, a collection of G(F) threads can consume a time budget on the basis of thread execution, memory access, or more complex means. Different operations may be instrumented as determined by the designer. Coarse time estimates allow for system designers to utilize intuition and allow software execution to be relatively resource independent for high-level models.

The designer is completely free to select the appropriate scheduling policy and time budgets. This permits a rate for system resources which is far lower than that which would be required if instruction set simulators were being utilized. Since the lower rate represents a more abstract model, it also allows the resource to be viewed as a platform for more complex behavior early in the system design process, thus enabling early system level modeling.

4. Example

In previous research [2], our simulations results were within 5% of an actual measured laboratory setup, giving confidence to the modeling and simulation capabilities of our approach. In this work, we modeled and simulated a streaming MPEG-1 Layer III audio decoder [17] in a client-server model distributed over a network. Our system architecture is that of Figure 1, and FI thread relationships are that of Figure 2. This example contains an illustrative set of interacting steady-state and transient system behaviors. Our goal is to simulate the system to determine the steady state processing requirements for P₁, and then to determine the performance of the transient tasks also being processed by P₁. As before, the busses are not modeled. Bus domain 2 (P₂,H₃) is a thread-on-accept file server, which dynamically loads the CPU and network. A network buffer (NIC), H₃, provides a clean interface to the switch. The network C thread, N₁, fills and empties the interface buffers asynchronously. The high-level model of the network utilized only a single C thread.

Bus domain 1 (P₁,H₁,H₂) depicts the thread relationships for the client. It has an additional piece of hardware (H₂) in the form of a buffer for the audio DAC. The client scheduling domain thread, G₁₁(C), schedules two threads, G₁₂(T) and G₁₃(F). G₁₂(T) is the MPEG decoder task. It is computationally complex with a steady state response which can be met when the processing power of the client CPU, P₁, is adequate. Once the power is determined, P₁ is free to execute transient tasks. For this, we modeled G₁₃(F) which simply writes data to the network buffer. Because of space constraints, we focus on the behavior of the client with the assumption that the network and the server are always fast enough

to meet the data demands of the CPU. However, the network buffer remains bounded in size and only empties itself periodically.

4.1 Instrumentation

The hardware is instrumented in two places: the output buffer of the DAC, H₂, and the output buffer of the NIC, H₁. We monitor the performance of G₁₂(T) and G₁₃(F). The NIC buffers are fixed in size and are emptied each time they execute, the rate of which is determined by the buffer's frequency. Performance is measured as percentage of occupied buffer slots at each execution.

The frequency of the C thread that models P₁, and so the execution rate of its scheduling domain thread G₁₁(C), is varied in our simulation. Time budgeting of the G(T) and G(F) threads mapped to a scheduling domain is resolved to this underlying power. Each time G₁₁(C) executes it gives priority to the MPEG task, G₁₂(T), allowing it to run until the DAC output buffer is filled or the MPEG task has exceeded the processing power of P₁'s C thread. This is measured by assigning a cost to the inverse modified DCT (mDCT) function of the MPEG software. Each time this function is executed, P₁'s C thread is charged a fixed amount by the scheduling domain thread mapped to it, G₁₁(C).

When the total time budget of the execution of G(F) and G(T) threads mapped to a G(C) scheduling domain exceeds some maximum value, the computational capacity of the underlying resource has been exceeded. Other C threads in the system simulation are then allowed to execute. Thus, steady state and non-steady state system responses are determined by a mix of processing power (the rate of the C threads), scheduling decisions (modeled in the G(C) threads), computation loads (determined in the G(F) and G(T) threads), and data availability.

4.2 Simulation Results

For this experiment, the audio and NIC buffers have a period of one and the frequency (power) of the client processor P₁ resource is varied. With a 44.1 kilosamples per second per channel output rate of the DAC buffer, each channel 16 bits, and a 64 byte buffer, each flush of the buffer accounts for about 0.36 ms. When we assign a frequency of one to this buffer we also give a meaning to the frequency of the other C threads in the system. For example, if the processor resource has a frequency of 0.5, then each execution of the processor resource C thread accounts for 2*0.36 ms or 0.72 ms. The audio buffer is 64 bytes and the network buffer is 256 bytes. On P₁, each inverse mDCT costs 15 and each network write costs 2, with a total computation cost not to exceed 100.

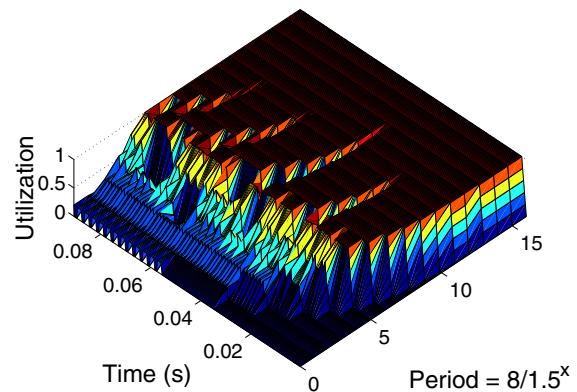


Figure 3 Steady State System Response

Figure 3 is a plot of DAC buffer utilization over time vs. the period of P_1 (inverse-frequency). The graph is semi-logarithmic on period, averaged over 5 time values for clarity. On the x-axis, 8 is the period at which the buffer utilization starts rising appreciably. The denominator is chosen to condense the data for better viewing. The start-up time is strongly dependent on the period of the processor resource. P_1 is unable to meet demand until its period is near $8/1.5^{13} = 0.04$, or about 24 times the speed of the buffer. With a 0.36 ms buffer cycle time, P_1 must cycle every $0.36 \text{ ms}/24 = 0.015 \text{ ms}$. The maximum amount of computation is 100, and each inverse mDCT costs 15, then we can have at most 6.66 inverse mDCTs per cycle. P_1 must therefore sustain about $4.4 \cdot 10^5$ inverse mDCT/s in order to meet steady state demand for $G(T)$.

The periodic horizontal streaks result from the way the algorithm reads a frame of data and then proceeds to do all the inverse mDCT operations at once, starving the buffer as P_1 is overloaded. From this, a designer might reduce peak loads by either altering the algorithm or using a buffer design that is better able to handle bursty traffic.

Figure 4 is a plot of the utilization of the NIC, H_2 , which is filled by the transient task, $G_{13}(F)$, which $G_{11}(C)$ only executes when $G_{12}(T)$ is I/O bound. When $G_{12}(T)$ is compute bound, $G_{13}(F)$ suffers, as is evident by the streaks which match those in Figure 4. $G_{13}(F)$ eventually plateaus when it also becomes I/O bound.

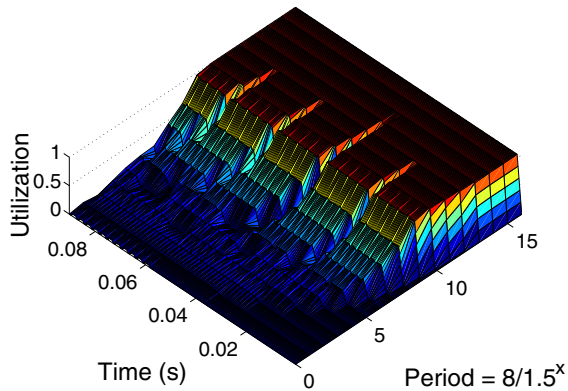


Figure 4 Transient System Response

The key difference in observing the behavior of the steady-state MPEG thread, $G_{12}(T)$, and the transient processing of the background thread, $G_{13}(F)$, is seen by examining each graph on the basis of three regions. Each graph has a sloped region for low processor frequencies, a rilled region in which processing is mostly saturated, and a completely saturated region. But while the steady-state task, $G_{12}(T)$, is only correct in the region in which steady state performance is met all the time, the performance of a transient task may be acceptable over all three regions. While performance of the transient task is clearly better in the completely saturated region, its net performance may also be measured as a cumulative amount of processing over the scheduling scope, i.e. for $\tau_6=0.10$ s. That is, the net difference between the processing in the rilled region as compared to the completely saturated region may be negligible. Thus, our modeling and simulation capability allowed us to determine the processing power required in a complex system and then determine the performance of the leftover transient tasks.

5. Conclusion

Current digital system design approaches have been limited by the view that the entire system must be characterized by a single design style, such as reactive, real-time, or general-purpose. By starting with an observation that all system behaviors may be classified as steady-state or transient, and all design styles may be classified as hierarchical decomposition, static mapping or directed platform, we developed a formalism to show how existing design styles may be related. We used this to show how viewing system design as the simultaneous manipulation of function, architecture, and schedulers can result in novel design methodologies that more closely match emergent systems as well as simplify the need to match real-time processing requirements to resources. We introduce how local scheduling decisions for transient behavior can be optimized across broader system scheduling scopes. We presented FI simulation as the basis for unifying design styles through a mix of formalism and simulation-based experimentation. We are continuing to develop our modeling approach.

6. Acknowledgments

We thank our other research team members: Henele Adams, Chris Andrews, Chris Eatedali and Neal Tibrewala. This work was supported in part by the General Motors Collaborative Research Lab at Carnegie Mellon, ST Microelectronics, NSF Award EIA-9812939, and the Pittsburgh Digital Greenhouse.

7. References

- [1] J.M. Paul, S.N. Peffers, D.E. Thomas. "A Codesign Virtual Machine for Hierarchical, Balanced Hardware/Software System Modeling," *DAC*, 2000.
- [2] N.K. Tibrewala, J.M. Paul, D.E. Thomas. "Modeling and Evaluation of Hardware/Software Designs," *9th international Workshop on Hardware/Software Codesign*, 2001.
- [3] B. Selic. "Turning Clockwise: Using UML in the Real-Time Domain," *Comm. of the ACM*, pp. 46-54. Oct. 1999.
- [4] J. Davis II, M. Goel, C. Hylands, B. Kienhuis, E. Lee, et. al, "Overview of the Ptolemy Project," ERL Technical Report UC/ERL No. M99/37, Dept. EECS, Berkeley. July 1999.
- [5] F. Balarin, M Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, et.al, *Hardware-Software Co-design of Embedded Systems. The Polis Approach*. Boston: Kluwer. 1997.
- [6] D. Gajski, F. Vahid, S. Narayan, J. Gong. "SpecSyn: An Environment Supporting the Specify-Explore-Refine Paradigm for Hardware/Software System Design," *IEEE Trans. VLSI* '98.
- [7] Y. Li, W. Wolf, "Hardware/Software Co-Synthesis with Memory Hierarchies," *Proc. of ICCAD98*, pp. 430-436. 1998.
- [8] R. Ortega, G. Borriello. "Communication Synthesis for Distributed Embedded Systems," *ICCAD98*. pp. 437-453. '98.
- [9] K. Rompaey, D. Verkest, I. Bolsens, H. De Man. "CoWare - A design environment for heterogeneous hardware/software systems," *Proceedings of EURO-DAC* 1996.
- [10] D. Desmet, D. Verkest, H. De Man. "Operating System Based Software Generation for Systems-on-chip," *DAC* 2000.
- [11] A. Dasdan, D. Ramanathan, R. Gupta. "Rate derivation and its applications to reactive, real-time embedded systems," *DAC* 1998.
- [12] G. Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*. Boston: Kluwer. '97.
- [13] P. Pop, P. Eles, Z. Peng. "Schedulability Analysis for Systems with Data and Control Dependencies," *EURO-DAC* 2000.
- [14] <http://www.inmet.com/sld/>
- [15] <http://www.systemc.org/>
- [16] <http://www.mathworks.com/>
- [17] <http://www.iis.fhg.de/>