VHDL Performance Modeling

Fred Rose, Todd Steeves, Todd Carpenter

Honeywell Technology Center
3660 Technology Drive
Minneapolis, MN 55418

rose (steeves, carpent)@src.honeywell.com

Abstract

The design and development of high performance computing systems is becoming increasingly complex. A primary ingredient of a sound design methodology is a detailed system performance model. A performance model expressed in VHDL serves as a simulatable specification, aids the identification of bottlenecks, and supports performance validation. It also supports trade-off studies for what-if analysis and documentation of design decisions. This document describes a methodology for developing a VHDL performance model within the RASSP design environment. Such a model can be used for capturing and documenting architectural level designs, and as a testbed for architectural performance analysis studies. This document outlines an approach based on the VHDL performance models developed by Honeywell. This is meant to serve as guideline for the model developers, system architects, and review teams. This document will also serve as a tutorial on performance and architectural assessment using VHDL modeling.

1 Introduction

Throughout the 1990's and beyond, computationally demanding applications will rely on extremely complex architectures to deliver the required performance. We must develop products that are more functionally complex, fault tolerant, and cost-effective with higher overall performance. Development of subsystem breadboards and prototypes are costly approaches to demonstrating complex system performance. Minimizing the size, weight and power of embedded computer systems depends upon our understanding of the application problem and its computing requirements. This requires analysis, simulation and prototyping of hardware and software. Analysis alone in such complex instances is usually insufficient because the problem complexity can restrict the fidelity of the analytical model. Simulation is a comprehensive, cost-effective approach to evaluate the performance of these highly complex new designs. As budgets continue to shrink, less prototyping will be possible, forcing more simulation to verify performance. VHDL provides an excellent platform for such performance simulations. Prototyping provides high fidelity but at great expense, too great to use as a cost-effective means for computing system trade-off studies. But trade-off studies must be conducted to gain understanding and solidify the design to a sufficient degree that physical prototyping will be affordable yet have high impact. We have developed a system design approach based on analysis and simulation, employing VHDL based modeling and simulation of both hardware and the software, prior to commitment to prototyping.

Honeywell has developed this methodology and library in VHDL using standard commercial VHDL capabilities. The library consists of high-level building blocks such as configurable input/output devices, memories, communication elements, and processors. The processor model is the key element to the performance modeling methodology as it facilitates hardware/software codesign and coanalysis. These building blocks can be rapidly assembled and configured to many degrees of fidelity with minimal of effort. Standard output routines tabulate and graph performance statistics such as utilization and latency. These statistics can be used for performance verification studies.

There are three basic statistics collected as part of the system performance evaluation: latency, utilization, and throughput. Performance and trade studies will be interpreted with respect to these metrics. Performance studies will be used to analyze the performance of the system. Architectural trade studies will be used to evaluate and select between different architectural features.

This library has been used to model several architectures including the Air Force next generation Cockpit Display Generator, Theater High Altitude Area Defense (THAAD) image acquisition, and Boeing 777 avionics. It will also be used for several RASSP demonstrations and benchmarks.

2 Overview

This section provides a quick overview of the philosophy behind this performance modeling methodology. It describes the intended target of the modeling efforts, the uses of such a model, and its limits.

- **Performance models support performance and architectural trade-offs, early integration of hardware and software models, and design documentation**
- **Facilitates hardware/ software codesign which is essential to successful system design**
- **Generic library can be easily configured to a wide variety of designs**
- **Interoperability guideline will ensure models from multiple sources will integrate smoothly**
- **Hybrid models support smooth integration between performance and functional models**

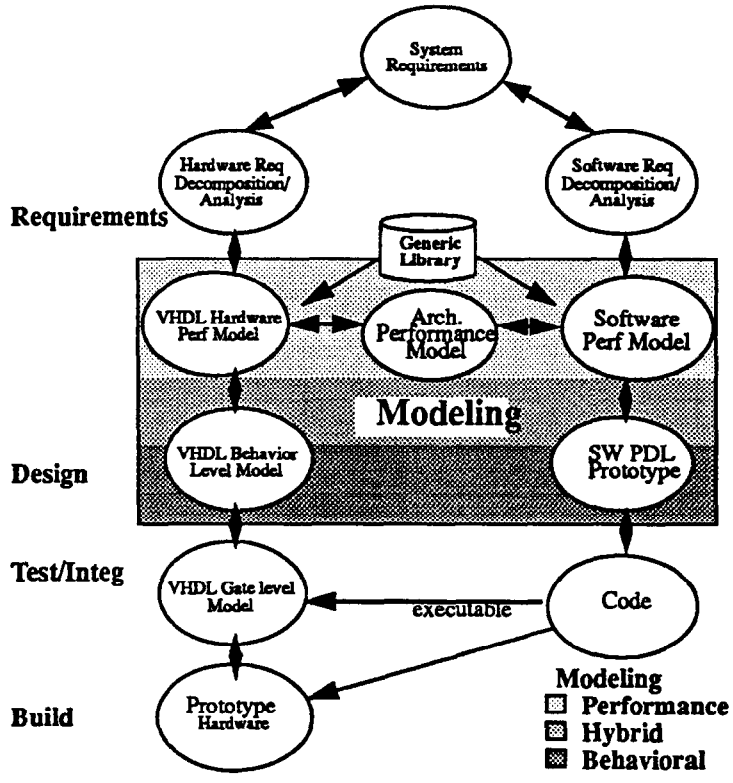

Figure 1: Performance models role in the RASSP design environment

The VHDL performance modeling methodology, Figure 1 is targeted towards high level description, specification and performance analysis of computing systems. The tools and techniques themselves are not targeted towards any particular application. The level at which is appropriate to apply these tools is at the architectural level. Architectural level includes the actual device or entity under study such as a signal processor, and its environment, such as sensors and actuators. In the case of an electronics system, an architectural level description would include information about both the hardware and software. Note that the definition of "system" is loose here. While we constrain the application of our performance models to electronic systems (although we have performed human==>electronic console operability studies with the performance library), the library is fully capable of representing systems consisting of ASICs, boards, and subsystem cabinets, and sensor networks.

A model is a representation of some system. A model is necessarily lacking in some detail of the real system. Typical models include textual specification and requirements documents, analytical models, simulation models, and physical models. A model is typically used to describe the system in something other than the actual physical realization. Systems may be represented with models in spreadsheets, analytical models, textual descriptions, computer languages, etc. Analytical, simulation and physical models can be used to predict behavior of the system. One might wish to use models instead of the real system for a number of reasons. One frequent use is during the design phase of the system, when the actual device is not yet available. Other reasons include: the schedule might not accommodate use of the real system; the necessary experimentation might be destructive; the real system may be prohibitively expensive to create or acquire; the real system might not be instrumentable or sufficiently analyzable. Documentation and the numerous benefits reaped from following a rigorous process are other reasons for developing detailed models

Models are evaluated in numerous ways. They may be analyzed, simulated, emulated, or prototyped. The performance models discussed here are primarily simulated. Analytical models can rapidly become too complex to fully represent important system features such as resource contention. Emulation and prototyping are expensive, time-consuming, and necessarily occur late in the design cycle. Simulation provides results that may not be easily attained via analytical models, and faster and at less cost than prototyping. Additionally, simulation can accommodate mixed levels of design and various levels of fidelity and accuracy. However simulation does suffer from significant startup costs, complexity, and significant execution (CPU) times. A robust system design process should utilize all model evaluation methods as appropriate. This also means that a model useful for one purpose, may not be appropriate for another use. For example, an analytical

model used for reliability analysis is not necessarily applicable to performance analysis.

Performance modeling is applied during the early stages of system development. It provides another tool to the system designer, but is not intended to be stand-alone nor discarded at the end of the system development stage. Performance modeling can aid evaluation of design alternatives, capture design decisions and assumptions, examine system behavior at boundary conditions, and help determine bottlenecks and overdesign. The system designer can also utilize performance modeling for examining system sizing, topology, partitioning and capability issues. An important benefit of performance modeling is that it provides early interaction of system, hardware, and software designers.

Performance modeling and analysis has been done for years. In the mid-70s, N.2 and other languages were developed for performance analysis. There are currently several commercially available software tools for system performance analysis. These include SES Workbench, Transcend from TD Technologies, Bones from Cadence, Network 2.5, and several others. All these tools provide their own language, capture, and output analysis capabilities. Some also will output VHDL or Verilog or even better in the case of Transcend, provide cosimulation capabilities. However performance modeling with VHDL provides several advantages. VHDL is a standard language and is vendor independent (usually!). Models are easily transported, an important feature for multi-company or division projects, and for projects that span long time periods. VHDL is a expressive language with full hierarchy and configurations which allow development and application of highly configurable and flexible models. Consistency and completeness checks are automatic. There are a wealth of VHDL-based tools available. A VHDL performance model also provides tight coupling to lower levels of design. Hybrid modeling provides the capability to mix multiple levels of abstraction, which can provide useful, detailed information for part of the design. A VHDL model also provides useful documentation of the design and increases the likely of model reuse due to the standard and long term, wide-spread support of VHDL. VHDL for performance modeling makes sense for designs which will have custom hardware built. It should be kept in mind that for certain applications, such as network communications analysis, some of the above arguments may not be as pressing. In those cases, a commercial tool may be a better solution.

This VHDL performance environment allows the systems architect to capture the system under study in a consistent, verifiable form. The VHDL simulation produces metrics which can be used by any commercial analysis package, spreadsheet, or other appropriate format to aid the design decision process. The results can be directly compared with the system specification to verify that the architecture meets the performance requirements. Once the architecture is verified (the latency, utilization, and throughput meet requirements, the system is self consistent, and size, weight, and power limits are met), the system is ready to proceed to detailed design. The performance model can also produce the architecture's characterization for its use in a higher level model, where this design would just be another building block.

The following subsections provide a high level view of the components of our modeling library. The primary library support packages, and some of the objects critical to the modeling environment are briefly described.

3 Library Description

3.1 VHDL Methodology

The architecture of the system is captured in an architectural or performance VHDL model. The architecture is broken down into major functional blocks, processors, memories, bus communication devices, etc. and the interconnection of these blocks is modeled. These are performance level models. The GPD program developed a library of generic performance models which can be rapidly customized for particular architectures. A performance model does not address system functionality, at least insofar as that real data (such as video input) is not processed and outputs (such as real display formats) are not generated. The data is characterized and represented symbolically. For example, one token could represent one frame of data. Honeywell has also developed a technique for characterizing and modeling the software's effect on the architecture. This allows VHDL simulation of the complete system.

The performance model is used to evaluate the design in terms of utilization, latency and throughput. The interpretation of these metrics is highly application and architecture specific, and must be driven by the system requirements. Some general rules are obvious: latency should be minimized, throughput maximized, and utilization should be uniform and at an acceptable level to allow for planned future growth. There are also relationships between these metrics. For instance, throughput can usually be increased by pipelining the architecture. However, this tends to increase latency. All of these metrics must be balanced against the size, weight, power, and cost requirements.

3.1.1 Signal Definition and Bus Resolution Function

Signals represent the physical interconnection of the hard-

ware elements. In this performance modeling environment, the value of a signal at any point in time is referred to as a *token*. A token represents some user definable quantum of data. A special signal value exists which is the null token, and indicates the absence of a token on the signal line. The signals are all the same VHDL record type. The signal definition is also the starting point for interoperability.

The exact nature of the bus resolution function is very much tied to the development of the communication elements. The bus resolution function is used to resolve the value of a signal that has more than one driver.

Aylor, et al, at the Univ. of Virginia have done some pioneering research in the area of VHDL for performance modeling. They developed a scheme for using the bus resolution function to perform handshaking for token passing between components. We are using a similar set up. Our system is somewhat more complicated since we are modeling multiple-source, multiple-sink protocols.

We have defined an enumerated type called bus_status. It has four values: (idle, request, ack, busy). The basic token description then has a field for bus status. The bus resolution function uses this field, along with the fields protocol and priority to arbitrate the bus.

3.2 Generic Models

A description of the hardware architecture is required for the analysis. This description can generally be broken into four basic categories: the sensors and display devices, memories, communications elements and processing modules. The interconnection of these components is simply done via structural VHDL, and is not discussed here. Additional components, such as the power supply, do not affect the functionality of the high-level architectural performance model. In the following sections, the term attributes refers to parameters, not to VHDL attribute constructs. Setting the value of these attributes is what is referred to as "characterizing" the component. This is a task for the system architects and directly or indirectly, the customer (for instance, by requiring a particular system throughput). The following sections discuss the utility, composition, and characterization of each of the above listed generic components.

3.2.1 IO devices

A key issue in the performance of the system architecture is the collection and concentration of data from sensors in physically disparate locations on the vehicle, and the distribution of data to display devices. These input/output devices are probably the easiest elements for which to build performance models. They tend to be (from a performance point of view) purely data sources or data sinks, and can generally be characterized by the rate at which they consume or produce data. An assessment of the sensor-output latency will require detailed information on the nature and distribution of all sensors and displays. Information required for all such I/O devices in the system includes the input or output rates, the performance loading (amount of time or processing per data item), and the data type.

3.2.2 Memory

A memory's contribution to system performance can generally be modeled by a simple delay line. A request for data at a particular location results in the memory responding after some delay with that value. Caches are slightly more complex since they may respond with some failure indication, or at variable rates. The attributes necessary for the performance modeling of memories are the physical size of the memory, the speed and access type, implementation technology, cache types and expected hit ratios.

Examples of memories which can be modeled include mass storage elements such as hard disks, CD ROMs, or tape drives, program and data memories such as ROM and RAM, buffer memories like video RAM or communication FIFOs, and caches.

3.2.3 Communications

Communications elements are reasonably difficult to develop, since they model control as well as data flow. They must accept requests, arbitrate, and route those messages to the appropriate addresses. Associated with communications are the interfaces and the actual transmission media. Certain media, such as satellite links, have sufficient latency to warrant their inclusion in the high-level performance models. Other media, such as the PI-Bus backplane, have negligible transmission latencies compared to their high arbitration overhead. A communications interface can generally be abstracted to its arbitration overhead costs, bandwidth, latency through the interface and media, and protocol/arbitration mechanisms. Routing might also be included for point to point and message passing schemes. A mechanism for modeling bus protocol with VHDL record types, similar to University of Virginia's, was developed. Typical communication elements found in a system include disk interfaces, backplanes, video distribution networks, internal processor-to-memory busses, processor-to-processor links, and external communication elements such as satellite up/downlinks.

3.2.4 Processing

The processor model was the most complicated element to develop. The processing node must make the control flow

decisions for a simulation. It also embodies the primary effect of the software upon the system performance. We have implemented generic VHDL processor models which execute user supplied VHDL programs that capture software performance aspects.

Attributes necessary for modeling the processor include the throughput, available resources, instruction timing, OS and software overhead, and interface information. In addition, *software* must be written which will control the time dependent behavior of the processor. This software characterization is currently done by using VHDL as an HOL software language, where high-level performance facets of the software are captured at a level appropriate for the hardware modeling. The software effects can be as abstract as "Consume 10% of the CPU for OS Overhead," or as detailed as an instruction set architecture (ISA) simulation.

The trade-off in the level of software modeling is cost and time spent modeling versus the fidelity necessary to obtain the required data. It must be emphasized that the utility of an extremely detailed hardware model is severely limited if the software is not known and modeled with the same fidelity. Although such modeling of software at early stages of system design might be contrary to popular design practices, we contend it is necessary in a good system design methodology.

Processor Model

A critical factor in evaluating performance of complex system architecture is very often accurate characterization of the processor(s) contained in the design. In particular, the capability to model software, if appropriate, with the full detail of actual code executing in the system. Therefore, the driving requirement for the performance modeling methodology was to model the highly complex software and hardware level interactions. As a result, much effort was expended to develop a flexible, high fidelity processor model. This model provides a powerful software modeling capability over a wide range of model levels: from modeling of actual code to high level data flow representations. The software modeling is based on the full capability of the VHDL language. Additional features have been added to handle interrupt service, preemptive tasking, task communication (like the Ada rendezvous), task synchronization and other services one would expect from a executive or general purpose operating system. The scheduling model can also support static and dynamic tasking and even rate monotonic scheduling.

The processor model can automatically provide reports that detail processor task activity timelines, missed deadline reports, processor utilization, task processor utilization profiles, and overall latency.

The processor model is divided into four parts:

The software models

The scheduler or thread manager

The processor hardware model

Dedicated hardware under processor control

The relationship between the components of the processor model is shown in Figure 2 . The model is best visualized in a top down arrangement. At the highest level, the application task software models communicate and request services through the task bus. The scheduler communicates and controls all tasks via this bus. Tasks can request the full set of processor model services through the task bus. These services will be passed over the bus and then down through the one or more stages of the model. At each level, the request will be examined and those supported by that level will be fulfilled and the request returned back up to the top level task. The request for services contains the standard token as well as processor model specific information. The application tasks do not need to have any knowledge of what components in the model will handle its specific request. The scheduler will process the services corresponding to basic operating system services. The scheduler supports a variety of services such as semaphores, intertask communication, and interrupt service routines. Requests for actual processing are passed down to the processor hardware model. The user can define the processors performance requirements of each of the services or instruction supported by the hardware model. At lowest level, the processor distributes requests to separate hardware component models. These models are attached to the processor bus. All devices on the bus interface with the standard BIU local bus requiring no special modification to the processor model or the component models. The processor examines request to the hardware components and performs a logical to physical address translation. It will supply the needed destination addresses for all communications on the bus. Attaching hardware component models to the processor model is optional and most likely will not be needed for high level performance models.

The processor hardware model supports three types of requests. First, the fixed command set controls the internal processor state, allowing the upper level components to request services that change state in the interrupt, cache, and other miscellaneous resources in the processor. The user cannot readily change the instruction behavior regarding processor state, but can define the processing requirements for the instructions. The second set, soft instructions, correspond to actual data processing that will be performed by the processor. This will contain commands such as floating-point-multiply, divide, integer add and so on. The user can define the names of the services as

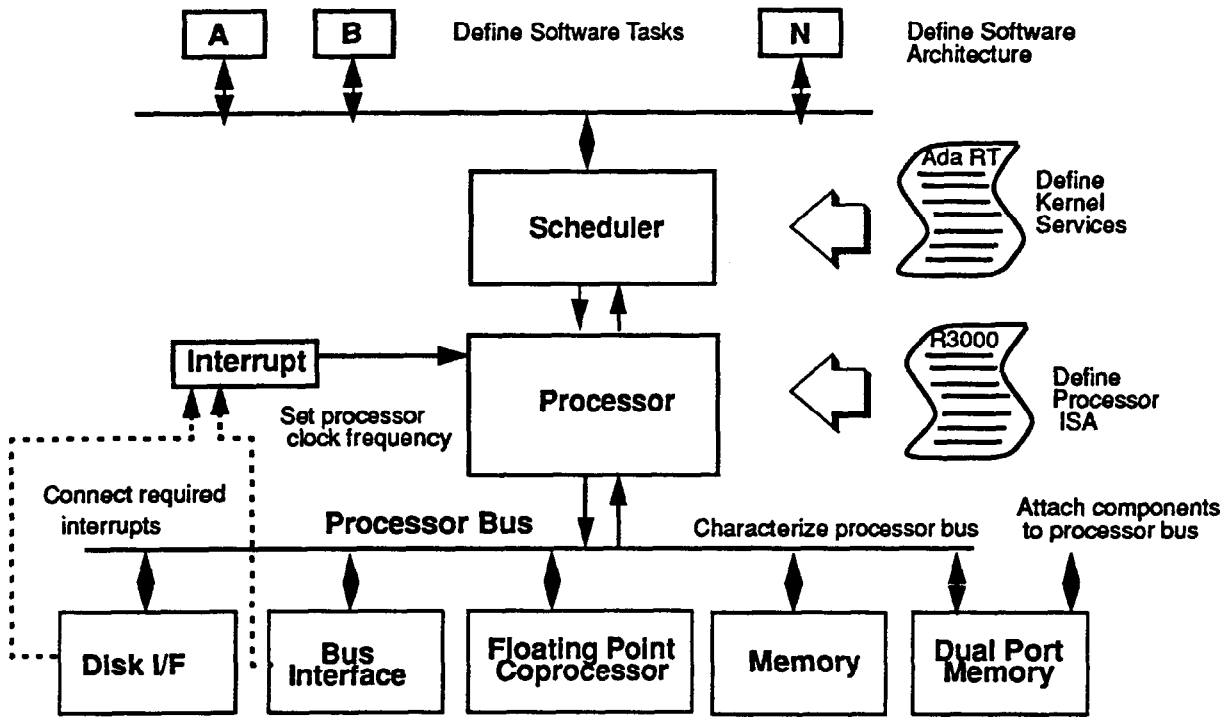

Figure 2: Processor model supports multiple characterization levels

well as the processor demands of each. The final set supports interaction with separate hardware components under the processors control. This would include such items as coprocessors, bus interface units, and disk controllers. To request services from this set one must have characterized the processor bus and attached the appropriate components to it. The token generated by the requesting task is used to communicate with the low level hardware models. The software task must fill in the appropriate subfields of the token, since the processor model will only supply the bus destination address for the token. The translation of request to BIU destination is defined on a request by request basis. As with the soft set, the user can readily specify the number of requests needed, the request names, the BIU address strings, as well as any processor resources that might be needed in support of the service. The user can also specify if the processor should be blocked until a token is returned from the chosen component. If blocking is specified, the originating task will be returned the actual token sent by the hardware device. This enables the software models to both send and receive information outside the processor model.

Software Modeling

Starting with the software model, a process or task is represented by a VHDL procedure that requests services from the scheduler and hardware by calling procedures defined in the packages SchServices. This procedure provides the means for the user to simulate the software activities on the processor. The standard procedure prototype for the software process models is shown below.

```
PROCEDURE do_Task_foo (me: INOUT task_id;
 SIGNAL send: OUT task_message;
SIGNAL receive: task_message;
parms: NEW_task_parms)
```

Procedures representing processes must accept three formal parameters known as ME, SEND, and RECEIVE. SEND and RECEIVE are VHDL signals that are required by the procedures in SchServices for communicating with the scheduler. The simulated software should not explicitly change or make any other use of these signals. The formal parameter ME has three components, HOST, ID and GLOBAL. HOST and ID are also used for communication between the procedures in SchServices and the scheduler and should not be changed by the simulated software. GLOBAL is provided as a substitute for the application's global memory. For each processor the scheduler maintains a value for GLOBAL. The scheduler multiplexes many processes onto one processor by only

allowing one process to be running at a time. Each time it allows a process to run (by invoking the process or completing a service request), it passes to the process the current value for GLOBAL. Each time the process makes a request for service (or terminates), the scheduler updates its value for GLOBAL with the value provided by the process. Currently GLOBAL can contain any custom information needed for the software model.

Procedures defined in SchProgram may use all the VHDL construct permitted for procedures with two exceptions:

They may not use the VHDL wait statement; SchServices provides a number of substitutes

They may not do anything with VHDL signals other than passing SEND and RECEIVE to the SchServices procedures.

The SchThreadManager entity consists of a scheduler and a collection of control threads. The scheduler uses the thread to run software processes or tasks. When it is time to run a process, it allocates a thread to the process and instructs the thread to execute the procedure in SchProgram corresponding to the process.

A thread that is READY means that its associated process is ready to run; SUSPENDED means that the associated process is awaiting the occurrence of one or more events. A thread that is INACTIVE is not currently assigned to a process (i.e., it is available for assignment to a process). type thread_state is (READY, SUSPENDED, INACTIVE);

The scheduler passes software request for services that require intervention from the hardware (execute CPU instructions and dedicated hardware operation) to the CPU for execution. Other requests (such as post event, wait for an event, delay, schedule process, etc.) are performed by the scheduler.

The scheduler can support both dynamic and static tasking as well as periodic tasks (including rate monotonic). The scheduling algorithm is preemptive and priority driven. The scheduler always chooses for execution the process with the highest priority among those processes that are ready to run. The scheduler's view of time is defined by the real-time clock. It updates it internal time value only when it receives a clock interrupt. Consider the following scenario. Assume that the clock's period is 5 ms (starting at 0 ms) and that process P is scheduled to become ready for execution at time 22 ms. At time 20 ms a clock interrupt occurs, causing the scheduler to update it time to 20 ms. Assume that at simulated time 23 ms the scheduler has cause to reschedule (for example, an executing process suspends itself). The scheduler will not run process P at this time because in the scheduler's view, the time is still 20 ms and will remain so until the next clock interrupt which won't occur until simulated time reaches 25 ms.

The scheduler issues a busy-wait command to the CPU whenever its ready queue becomes empty. This command tell the CPU not to report back to the scheduler until an interrupt has occurred.

The CPU entity receives input control via the external world via requests to perform hardware operations from the scheduler and from interrupts from its peripherals (via the interrupt manager). While the CPU is performing an operations it waits for either the operation to complete or for an interrupt to occur. If the operation completes, it signals the results to the scheduler. If an interrupt occurs, it signals the partial results and the interrupt to the scheduler.

3.2.5 Characterization - Generic Distribution Language (GDL)

The Generic Distribution Language is a language used to describe generic mathematical modeling distributions. The original syntax was developed by Honeywell as part of our ADAS modeling environment. We have adapted it for VHDL. Only a subset is currently implemented. GDL allows time and data dependent behavior to be specified in an easy and robust fashion.

Distributions which are currently supported can be seen in Tables I and II.

For example, assume we must model an input device which sends information to one of three video merge devices. The device to which it is sent is an exponential distribution focussed on sending it to VM0. Finally, we might want to provide a seed for the distribution so the sequences are uncorrelated (And of course the seed is from a handbook of random numbers!) We describe this distribution in Figure 3.

3.2.6 Nonfunctional Attributes

The VHDL performance model is also annotated with generics such as TRACE, SIZE, COST, and POWER. These attributes do not affect the performance of the model, but are for statistics collection. TRACE is an inherited attribute which allows the user to specify which parts of the VHDL model will report their performance. Since it is an inherited attribute, the user can specify, at any point in the hierarchy, that from there on down the simulation will track the throughput, latencies, and utilizations of the components.

COST, SIZE, and POWER are other attributes which can be used in a post-processing fashion to synthesize each of those attributes for intermediate levels in the hierarchy.

Distribution	Example	Notes
NULL	Null	No range (returns 0.0)
COUNTER	COUNTER RANGE 0. 20	repeatedly counts from 0 to 20
CONSTANT	CONSTANT 3	constant value of 3
ITEM	ITEM Engine_LOCALRAM	always return string "GP..."
UNIFORM	UNIFORM RANGE 100 200	uniform dist from 100 to 200
EXPONENTIAL	EXPONENTIAL 5	exponential dist with mean of 5
GAUSSIAN	GAUSSIAN 10.5	gaussian (normal) distribution with mean of 10 and standard deviation of.5
POISSON	POISSON 10	poisson distribution with mean 10

TABLE I. Distributions

Distribution	Notes
RANGE 5 100	Specifies range from 5 to 100. This bounds the distribution to reasonable limits. I suggest to always use it. And avoid zero times and sizes because they can hose the simulation.
MAP 1 ONTO GP_DSM_LOCALRAM	For a particular distribution, the MAP will assign a string value to a dist generated value. This is useful for destination fields. Note that the range must be fully mapped.
SEED 1 3	Seeds a particular distribution. The second seed is optional.

TABLE II. Distribution Modifiers

```
GENERIC destination_info: STRING: = "EXPONENTIAL 1.5"
                                    "RANGE 1 3"
                                    "MAP 1 ONTO GP\_VMO":
                                    "MAP 2 ONTO GP\_VM1":
                                    "MAP 3 ONTO GP\_VM2":
                                    "SEED 17";
```

Figure 3: Detailed distribution example

Alternatively, they can be used in results analysis to provide measures just as throughput per dollar or utilization per watt.

3.3 Tool Support

3.3.1 Commercial VHDL Tools.

Full language support from the commercial VHDL simulator is crucial, since many advanced features of the language are used by the models. Care was taken in the model development to insure that fully compliant IEEE-1076 VHDL code was created. Stimulus to a performance model must be applied in the form of test benches, which is standard VHDL. Vendor dependent features, such as simulator control languages, should be avoided at all costs.

3.3.2 GNU Emacs

Honeywell has developed a VHDL mode for GNU Emacs. This mode is loosely based upon some Public Domain (PD) tools obtained from UseNet. Honeywell has significantly enhanced this mode to allow syntax directed editing, enforce certain capitalization rules, control indentation, and handle comments. There is expressly NO guarantee for either GNU Emacs itself or the VHDL mode.

To use the mode, all one has to do is create and edit VHDL files using GNU Emacs. When the mode is properly installed, it will unobtrusively aid even the experienced VHDL designer in producing more readable code in an efficient and rapid manner.

This VHDL major mode for GNU Emacs is available via Internet, vhdl-mode@src.honeywell.com.

4 Evaluation

This section describes the performance data produced by the simulation and how it is collected. There are three basic statistics, latency, utilization, and throughput. Performance and trade studies are interpreted with respect to these metrics. Performance studies are used to analyze the performance of the system. Architectural trade studies are used to trade off different features of the architecture, such as FIFO size.

Latency The time it takes to get from pt A to pt B.

Utilization time busy / total simulation time

Throughput Data processed / time period

4.1 Latency

This attribute is the most difficult to obtain from the simulation. This is because latency can be specified in so many different fashions and over a wide range of locations. For instance, the driving requirement might be that a particular pilot directive, say a push-button event, results in a cockpit effect, for instance, a display format change, within 100 milliseconds. In terms of the simulation, a single token representing that push-button event must be generated, and the time of its creation recorded. That single event might trigger hundreds or thousands of other events in the simulation, which might happen to be concurrently handling other events. Once the effects of that initial token propagate through the system and reach the display, that time must be recorded. If every event were recorded, the simulator performance would degrade due to memory and file I/O limitations. Means must be identified to track only particular types of information through the system. This situation is even more complex when a single event results in multiple events. A single file is kept containing information such as token id, source node, intermediate stops, final destination, and times associated with each. A post-processor extracts the desired latency values.

4.2 Utilization

A process is added to every architecture to monitor utilization. It is essentially a counter which keeps track of the amount of time the element is busy. At the end of simulation, each module, outputs this information. A post-processor can sum it to any level within the hierarchy to provide a conglomerate utilization number. Such post-processing can even be augmented by such parameters as size, weight and power, to give a utilization per dollar, or throughput per area measure, or whatever measure is deemed appropriate and meaningful by the system architects.

4.3 Throughput

Throughput is handled very much like utilization. Each module keeps track of the amount of data processed. A post-processor can be devised which will handle multi-level calculation of throughput.

4.4 Output Analysis

Each simulation study has a basis in the statistics captured above. Each study, and the expected or required output will have to be described in those terms. Each trade study will have a preferred output format defined. Typical metrics which are evaluated include:

Task activity state timelines

Task schedule time budgets

Missed deadline reports

Component utilization

Task utilization profile

Memory bandwidth utilization

System and component latency

4.5 Output formats

Possible means for representing the output statistics can include the following. Keep in mind that all the output formats are generated via two steps, data reduction and data presentation. Each step can be accomplished in a number of different fashions, such as custom C programs, generic Unix scripts, or commercial tools. Our approach has been to develop generic, flexible, platform independent Unix scripts. Past experience has demonstrated that there is no singular method of presentation that will satisfy everyone, so it is paramount that those techniques be easily adaptable. The Unix script approach provided us with a broad

platform base upon which we wouldn't need to worry about porting or recompiling, while at the same time affording us the flexibility to rapidly display the data in any way thus far desired.

tables

histograms

scattergrams

activity plots

Utilization and latency are displayed in the form of a heated object scale. This provides the most visual way to find extremes, and is a good way to identify trends. However, it is by far the most complicated to implement, especially in a system portable fashion.

Figure 4 shows a sample output of an activity plot.

5 Interoperability

For the RASSP program to truly realize the full benefits of performance modeling, a common approach is required. The approach to interoperability will be two fold, first a style and usage guideline to establish VHDL as the common representation language will be developed, second specific interfaces to allow VHDL performance modeling tools to be integrated with the overall RASSP development environment will be developed.

The style consists of documented guidelines and generic VHDL models and utilities to support the guidelines. VHDL is the Lingua Franca, the language for doing business, of the EDA world. As the EDA industry increasingly moves to a "plug and play" business model, a common language becomes essential. However, common usage or style is also required. This has been recognized from the beginning in the VHDL community and a variety of style related standardization activities have occurred. However, to date, these efforts have focused on lower levels of design abstraction. The performance model interoperability guideline will raise that level of abstraction.

The first commonly accepted interoperability standard was the IEEE 1164 9-state logic package. This package defined a 9-state signal type and associated resolution functions. This state system is useful for RTL and gate level designs. The second standard is the recently completed VITAL (VHDL Initiative Toward ASIC Libraries) initiative. The VITAL standard description consists of a definition of a model style standard which is described as "VITAL compliant", the definition of a VITAL_TIMING package which includes timing-related subprograms and definitions, and the definition of a VITAL_PRIMITIVES package which contains combinatorial primitives as well as truth and state tables.

It is important for an interoperability guide to define the minimum amount of information to achieve interoperability. If too many constraints are present, widespread use may be difficult. If too few constraints are present, interoperability may be difficult to achieve. IEEE 1164 and/or VITAL defined a signal type, package utilities, general style and primitive descriptions. The performance interoperability guideline will be similar.

For model capability, the two modeling/simulation areas which the current performance models do not address are large multiprocessor systems and signal and image processing application specific models. Since the application specific models are best addressed on a case by case basis, we will direct our activity to the multiprocessor modeling area.

The current modeling capability has concentrated on small scale multiprocessing systems. As a result, the processor model has developed into a powerful, highly flexible generic model. Communication models on the other hand have been limited to processor-memory bus models with rudimentary arbitration schemes.

The proposed modeling capability will include an efficient "light weight" processor model as well as a generic interprocessor communication model. Multiprocessor systems will require more elaborate communication models capable of more advanced protocols and arbitration mechanisms. Scalable point to point communication models capable of supporting several different protocols are needed to model large multiprocessor designs.

It is critical that the performance model interoperate with the tools used to specify and capture the RASSP requirements and system design information. The interoperability guideline will define how Honeywell's performance model library will function with other elements of the RASSP environment. Honeywell will work with the primary architecture-level tool vendors, including JRS, Vista, and Omniview, on the Martin Marietta team to establish interoperability with their tools and VHDL performance models.

6 Related Programs

Under separate RASSP technology base contract, Honeywell, in conjunction with Omniview, will develop commercial quality performance models to the requirements specified in the interoperability guideline. This package will also include multi-processor modeling tools such as output analysis, capture, route/message cost, and architecture visualization.

Lastly, the performance models are proposed to be constructed in a manner to support hybrid modeling. Hybrid modeling supports performance analysis with interfaces to

functional models of communication and other device models. Functional modeling of selected system components will be necessary. A good example of this is the detailed functional modeling of the underlying communication mechanisms. Under a technology base contract, Honeywell, along with the University of Virginia and Omniview, will develop models and utilities to support hybrid modeling.

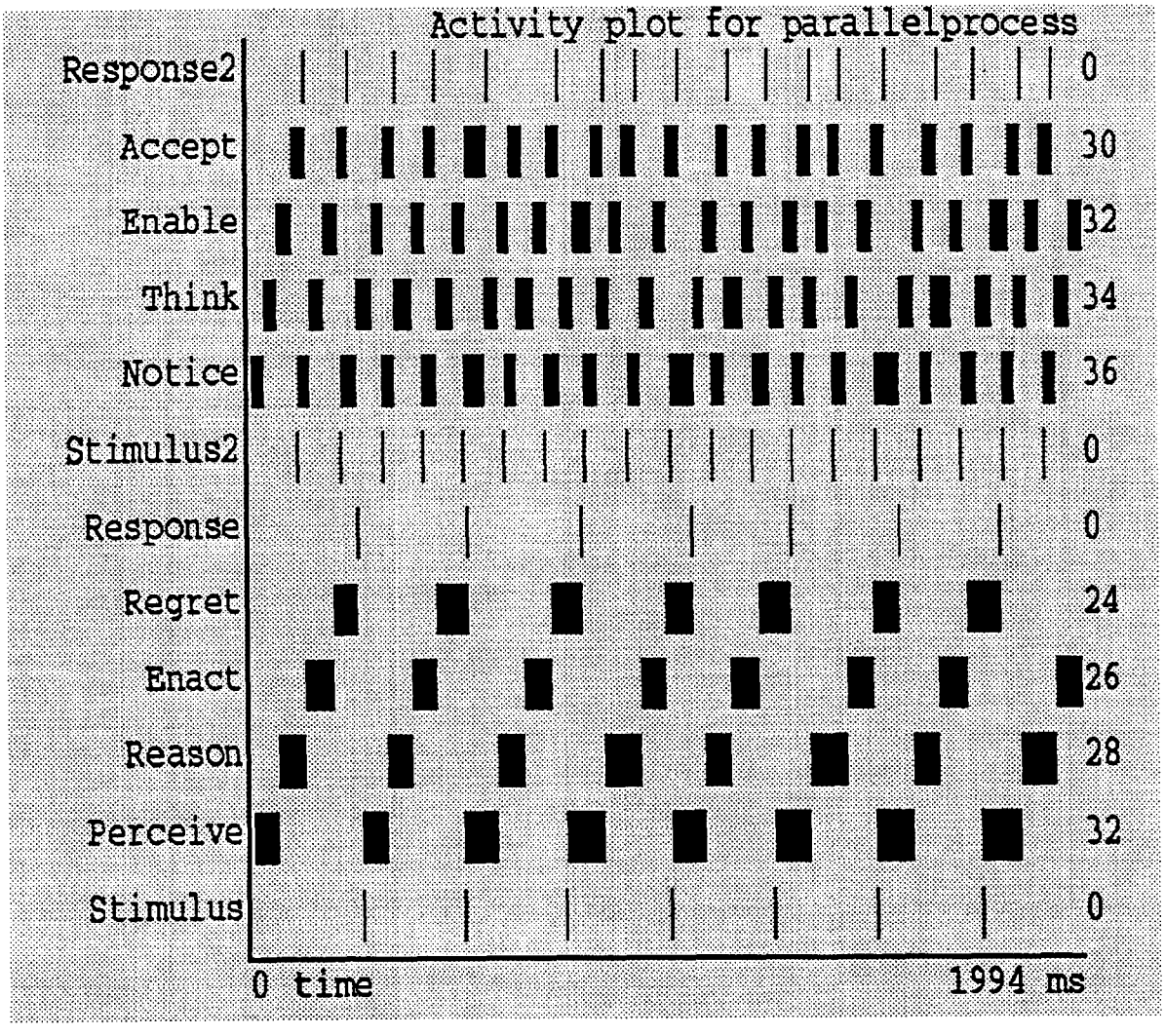

Figure 4: Activity plot shows utilization of components over time. This example is from a "human in the loop" study. The right most column is overall utilization.