

Early Performance Evaluation using Abstract Hardware/Software Models *

Sanjaya Kumar
Honeywell Technology Center

James H. Aylor, Barry W. Johnson, Wm. A. Wulf, Ronald D. Williams
University of Virginia
Charlottesville, VA 22903

Email: skumar@src.honeywell.com, jha@virginia.edu

Abstract

To address the separation between the hardware and software domains, this paper presents an abstract hardware/software model employing a unified representation. The model supports early hardware/software evaluation and trade-off exploration. In addition, systems can be evaluated at different levels of detail, allowing those aspects of interest to be focused on. The benefits of a unified representation are also discussed. Some examples are presented to illustrate the abstract hardware/software model and demonstrate the concepts and ideas embodied in the paper.

1. Introduction

Current design practice dictates the separation of the hardware and software design paths early in the design cycle [1]. These design paths remain independent with very little interaction occurring between them until system integration. In particular, hardware is often specified without fully appreciating the computational requirements of the software. Also, software development does not influence hardware design and does not track changes made during the hardware design phase. Thus, the ability to explore hardware/software trade-offs is restricted, such as the movement of functionality from the software domain to the hardware domain (and vice-versa) or the modification of the hardware/software interface.

During system integration, the software and hardware are finally combined. Problems that are encountered at this time may require modification of the software and/or hardware, resulting in potentially significant cost increases and schedule overruns. For example, the premature selection of hardware may require that the software attempt to correct hardware inadequacies [2].

Also, poor software performance may necessitate the development of additional hardware late in the design process [3].

To address the problems described above, a more unified, cooperative approach to the design of hardware/software systems is required, termed *hardware/software codesign* [4]. This capability leads to more efficient implementations and improves overall system performance, reliability, and cost effectiveness. Codesign can aid the design of embedded systems [5]. Because the complexity of embedded systems is increasing, it is becoming more important to employ decomposition techniques and abstractions to manage this complexity.

This paper presents an abstract hardware/software model employing a unified representation that allows hardware/software evaluation and trade-off exploration to occur early in the design process. Section 2 briefly describes the codesign approach. Section 3 introduces several modeling concepts and provides background material. Section 4 presents the abstract hardware/software model. Some modeling examples are provided in Section 5. Section 6 summarizes the contributions of the work.

2. Codesign Approach

As illustrated in Figure 1, we envision a codesign methodology [6] that supports hardware/software evaluation early in the design process (see bold region). This approach allows hardware/software partitioning decisions to be analyzed before committing to a particular design. The methodology is iterative and employs the ADEPT modeling environment [7][8]. In this environment, models are constructed with a collection of primitive elements, which communicate using tokens and a uniform handshaking protocol. Each primitive is described as a VHDL [9] process and has a corresponding colored Petri net representation [10].

* This work was performed while the first author was at the University of Virginia.

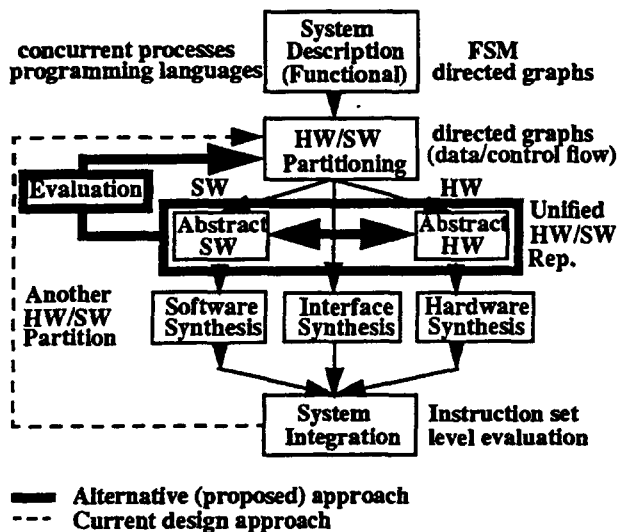


Figure 1. Codesign Approach

The codesign process starts with a functional, system representation independent of hardware and software. Next, hardware/software partitioning [11][12][13][14] is performed on an internal representation or on the system representation directly. An abstract hardware/software model is used to evaluate the quality of the partitioning decision. The resulting descriptions are then synthesized and evaluated at the instruction set level.

There are several benefits of this approach. One benefit is that costly changes to both hardware and software can be minimized. This cost arises due to system requirements not being satisfied because of late evaluation, as in the current design process (see dotted line). Another benefit is the ability to evaluate hardware/software systems quickly using abstract models, as opposed to detailed, instruction set level models (see Figure 1). This evaluation is possible due to the use of a unified hardware/software model within a common simulation environment. As a result, common analysis techniques can be utilized to examine such aspects as performance and reliability. For example, identification of software bottlenecks can reveal software functions which require improvement or perhaps hardware support. Also, this approach does not require that all descriptions be provided at the same level of detail (such as the instruction set level) before any evaluation is allowed. Finally, the use of an integrated environment allows the consequences of different hardware/software decisions to be evaluated within the context of the system being designed and supports model continuity [1], the gradual migration of system models into hardware/software implementations.

3. Hardware/Software Modeling Concepts

This section presents some concepts that are pertinent to hardware/software modeling. These concepts include unified representations, interpretive systems, and the request/resource paradigm.

3.1 Unified Representations

A *unified representation* is a representation that can be used to describe either hardware or software. Functional abstractions or data abstractions can be utilized to construct a unified representation. As an example of functional abstractions, data/control flow graphs, which contain nodes that represent operations and arcs that correspond to data/control flow between nodes, provide an abstract representation of an element. Each node corresponds to a functional specification whose abstract implementation can be expressed in terms of another, more detailed data/control flow graph. Data abstractions, commonly used in software development, can also serve as a unified representation for both hardware and software [15].

One application of a unified representation is illustrated by MCC's integrated modeling substrate [16] shown in Figure 2. This substrate provides a unified, internal representation. The intent of this modeling substrate is to support early evaluation and a wide range of hardware/software trade-offs, permit continuous and incremental hardware/software integration, allow more alternatives to be explored, and improve model continuity. Although this concept has not been implemented, it represents one approach to codesign.

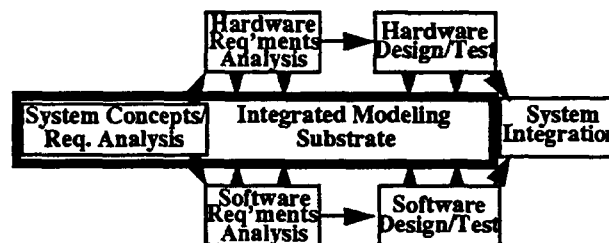


Figure 2. MCC's Integrated Modeling Substrate

In addition to the potential advantages described above, a unified representation promotes the evaluation of hardware/software systems in a common simulation environment and thus enhances the communication between hardware/software developers through the use of a common modeling paradigm. Also, a unified representation allows the possibility of cross-fertilization [17] between the software and hardware domains. Specifically, techniques and results from one domain can be applied to the other.

3.2 Interpretive Systems

Hardware/software systems can be viewed in terms of one or more layers of interpreters [18][19][20], with each interpreter having a fetch/execute model of behavior. Hoffman [21] developed a classification of interpretive systems based on the number of interpreters present within a system. In Hoffman's model, two types of interpreters are described. In a command interpreter, the commands, that is, the instructions to be interpreted, are not stored in a program memory. In a program interpreter, the instructions to be interpreted are part of a stored program. Thus, a one level program interpreter can be represented as " $P \rightarrow I^S \Rightarrow H$ " or as a triplet (P, I^S, H) . Here, a program P is interpreted by executing a software interpreter I^S on hardware H . This representation corresponds to a machine level program being interpreted by a microcoded processor.

Given this model of interpretive systems, a number of design problems can be investigated based on what portions of the interpretive system are fixed or variable. For example, if only the hardware resources and the microinstructions are fixed within a processor, several possible microprograms can be written to provide various types of functionality, such as different instruction sets. Thus, the portions of the interpretive system that are fixed influence the allowable hardware/software trade-offs that can be performed.

3.3 The Request/Resource Paradigm

The request/resource paradigm [22] has been used as a basis for descriptions of computer organizations. In this approach, a system model consists of a requester and a server. The requestor is a program, and the server is a set of resources. This model of functions requesting resources has been used by several system level modeling capabilities to support hardware/software analysis early in the design process, such as ADAS [23]. Models based on this paradigm are useful for analyzing the effects of concurrency and resource contention on performance.

4. An Abstract Hardware/Software Model

In this section, an abstract hardware/software model that incorporates the concepts above is described. The model is based on a unified representation for software and hardware, referred to as a decomposition graph. As indicated in equations (1)-(3), a decomposition graph DG is a hierarchical, directed graph consisting of nodes N and edges E . The nodes can correspond to either functional abstractions or data abstractions. The edges represent "consists of" relationships between the nodes, reflecting the fact that a node can be expressed in terms

of more primitive nodes. Thus, a DG employing functional abstractions depicts a functional decomposition, and a DG expressed in terms of data abstractions describes a data decomposition [15], a decomposition based on abstract data types. As can be observed, complexity management is an important consideration in this representation.

$$DG = (N, E) \quad (1)$$

$$N = \{n_1, n_2, \dots, n_z\} \quad (2)$$

$$E \subseteq N \times N, (n_i, n_j) \in E \leftrightarrow n_i \text{ consists of } n_j \quad (3)$$

As indicated in equation (4), an abstract hardware/software model HSM consists of a software model SM and a hardware model HM , each of which can be described using the unified representation. This description of a HSM represents a single software program executing on a single processor. However, the definition can be extended to accommodate more complex systems. Embedded in this HSM are the ideas of interpretive systems and the abstract request/resource paradigm.

$$HSM = (SM, HM) \quad (4)$$

Using functional abstractions, the modeling structure in Figure 3 was employed for describing abstract hardware/software models in the ADEPT environment. The HSM uses data and/or control flow graphs based on structured programming concepts [24][25]. The SM is a decomposition graph in which the operations are represented in terms of their resource requirements. This description can take several forms, such as three-address code representations used in compilers [26]. The HM is a decomposition graph consisting of a collection of abstract resources, such as functional units, registers, or local memory, which are utilized by the software operations. If the HM is a processor, the resources are arranged to provide a fetch/execute behavior. Software execution involves the request and subsequent utilization of the hardware resources.

The modeling structure in Figure 3 is quite general. The structure can be utilized to represent software execution on a processor, or a dedicated hardware element consisting of microcode controlling a set of hardware resources. Because the fetch/execute behavior is common to both hardware and software interpreters, this modeling structure is useful for representing and analyzing several types of interpretive systems. This structure also allows a hardware (performance) model to be refined by incorporating lower level implementations, that is, the model supports hybrid modeling [7][27]. Thus, hybrid modeling allows a system to be described at different levels of detail.

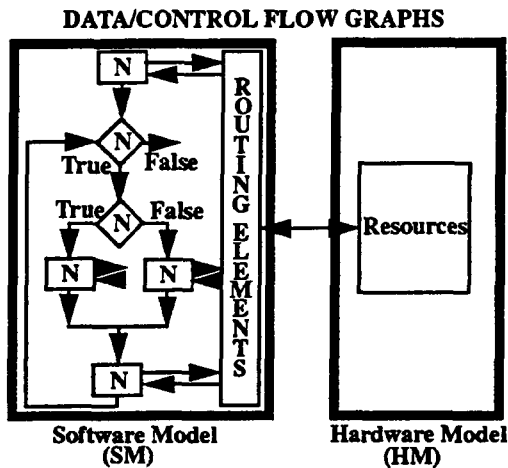


Figure 3. An Abstract Hardware/Software Model

5. Examples

Some examples are presented in this section to demonstrate the concepts and ideas developed earlier. The examples illustrate how the ADEPT environment can be utilized to evaluate systems early in the design process using appropriate abstractions and the abstract hardware/software model presented in the previous section.

5.1 Performance Evaluation of Algorithms

A *HSM* consisting of a finite impulse response (FIR) algorithm executing on an abstract digital signal processor (DSP) is evaluated in this section. A control flow graph representation of the FIR algorithm in ADEPT, containing parallel multiply-add ($x \parallel +$) and floating point add operations ($+$), is shown in Figure 4. The circular symbols correspond to process nodes, and the diamond-shaped symbol represents a decision node ($\leq c$). The other nodes are used to control token flow.

Execution of the *HSM* is initiated by the arrival of a token at the topmost process node. This node sets a field within the token to an integer, corresponding to an initial value for the number of loop iterations. The Y-shaped icon below the process node transfers either the initialized token or a token with a newly incremented value to the decision node. The decision node routes the token along either the true path (bottom arc) or the false path (right arc). The decision node, multiply-add node, and the increment node (+1) are part of a loop that is executed N times, where N is a user-specified parameter corresponding to the length of the filter. The rectangular icon in the feedback path of the graph ensures that the tokens flow properly as the model executes.

In this figure, only the multiply-add and floating point add operations are “executed” by the DSP. Execution of a software node occurs in three steps. In the first step, the arrival of a token at the top arc of a node enables the node for execution. Each software node to be executed “colors” (populates) the token with the node’s resource requirements as shown below. In this example, a floating point add operation (*FLTADD*) is to be performed on two register operands (*REG*), with the result to be written to a memory location (*MEM*).

$$token = (func, dest, src1, src2)$$

$$token = (FLTADD, MEM, REG, REG)$$

This request token is then sent to the DSP model via the node’s right arc through a *UNION* module (far right), the sole routing element in this software model (see Figure 3). The *UNION* module places a token on its output (*exec_req* port) when a token appears on any one of the two inputs. Once this output token has been fetched by the processor, a token is sent to the node’s bottom arc, enabling the next node.

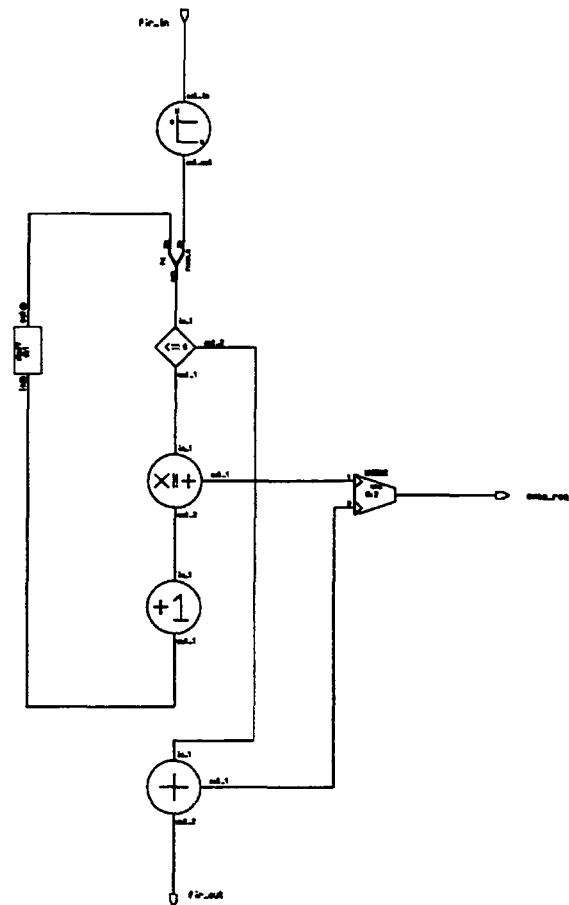


Figure 4. FIR Software Model

The two DSPs employed in this example are the TMS320C30 and the TMS320C30-40 which have single-cycle execution times of 60 ns and 50 ns, respectively. There are several features of these processors which make them particularly suited for executing filter algorithms, such as circular addressing and parallel multiply/add operations. In addition, pipelined operation provides high throughput.

The DSP model consists of three portions: a fetch stage, a decode stage, and an execute stage. The operand fetch stage has been neglected for simplicity but can also be included to more accurately characterize the software performance. Note that the processor model acts as an interpreter, fetching and executing operations. Thus, this modeling structure can also be used to represent and analyze software interpreters as well.

An abstract model of the fetch stage is shown in Figure 5. The model accepts a software operation (request token) from the left of the figure via the *fetch_in* port, waits for a user-specified fetch delay, and passes the operation to the decode stage (not shown) via the *fetch_out* port. The *BUFFER* module (far right) sends the software operation to the decode stage and also allows a new incoming token to be received by the fetch stage, allowing pipelined behavior. Upon arriving at the execute stage, the request is either granted or blocked until a resource is available.

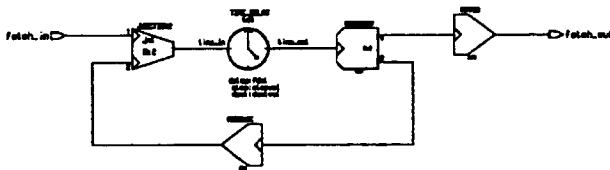


Figure 5. Fetch Stage of Digital Signal Processor

Figure 6 contains the simulation results for the execution of several FIR filters on the TMS320C30 and TMS320C30-40 processor models. The execution times in microseconds are displayed versus the filter length. In the simulation, it was assumed that the fetching and execution of the operations each require a single clock cycle. It was also assumed that the fetch and execute stages operate in pipelined fashion.

In a similar fashion, Figure 7 displays the simulation results for the execution of various FIR filters on the MC68020/68881 processor models with different clock rates. Thus, using this modeling environment, algorithms can be evaluated on different architectures.

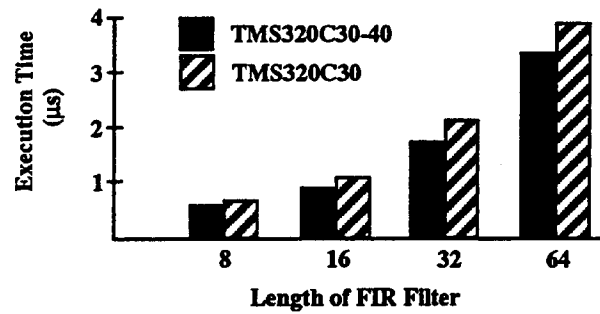


Figure 6. FIR Simulation Results (TMS320C30)

Although not shown, the execute stages of both processor models contain abstract resources, modeled as uninterpreted elements [7] lacking function. For example, the digital signal processor includes two abstract resources: a multiplier and an arithmetic logic unit. Both of these resources are modeled as delay elements which do not perform any functional transformations on the input operands. However, in order to capture the complete state of the computation as it proceeds, interpreted (functional) components are necessary.

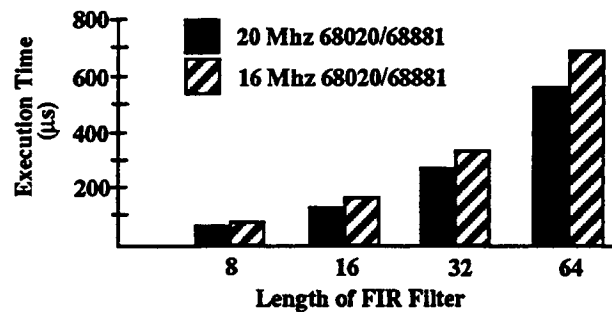


Figure 7. FIR Simulation Results (68020/68881)

5.2 Software Bottleneck Analysis

In this section, the modeling constructs and analysis techniques used within ADEPT to identify software bottlenecks, computations which require a relatively large amount of execution time, are described. The identification of software bottlenecks reveals functions which require further improvement, either in the form of more efficient algorithms or hardware support. Although the techniques are demonstrated on a software module, these techniques are applicable to hardware elements as well. The example presented in this section utilizes descriptions at different levels of detail.

Improvements in the execution time of critical computations can dramatically improve overall

performance. As shown in equation (5), the operator sensitivity metric S_i [28] can be used to quantify the portion of the overall execution time T taken by a software node i , where a node can be a single operation or an entire module. Software nodes with high sensitivity values are good candidates for further improvement.

$$S_i = \frac{T_i}{T} \quad (5)$$

The calculation of an operator sensitivity is performed with the aid of the *MONITOR* primitive. This ADEPT module is similar to a “voltmeter”, allowing the modeler to probe the inputs and outputs of one or more ADEPT modules. As the simulation proceeds, the *MONITOR* generates a file which contains the input-output latencies of the portion being probed. A post-processing program is then used to determine the operator sensitivities for the nodes of interest.

The process of performing bottleneck analysis is demonstrated on a portion of a “best-fit ellipse” feature extraction algorithm used in a system for aluminum defect classification [29]. This algorithm first calculates the orientation (axis angle θ) of the best-fit ellipse for a defect, based on the object’s center of mass and central moments. Using this information, the major and minor axes of the ellipse can be derived. These three pieces of information, the major axis, the minor axis, and the orientation angle, provide gross shape information which can be used for classification purposes.

The orientation procedure performs one center of mass calculation, three central moment calculations, and one computation of θ . It was determined through simulation that the operator sensitivity for a single central moment calculation was approximately between 26%-34% of the overall orientation execution time, making this module a candidate for further improvement.

Using ADEPT, hardware and software engineers can collectively decide how best to speed up a computation. For example, faster resources may be incorporated at some additional cost. Alternatively, special-purpose hardware can be employed. Using timings based on the MC68020/68881, Figure 8 shows the percent improvement in the execution time of the orientation code assuming a 20% improvement in the speed of various primitive operations: floating point multiply (FPM), floating point divide (FPD), and floating point arctan (FPA). The disparity in improvement is a consequence of the operation frequency.

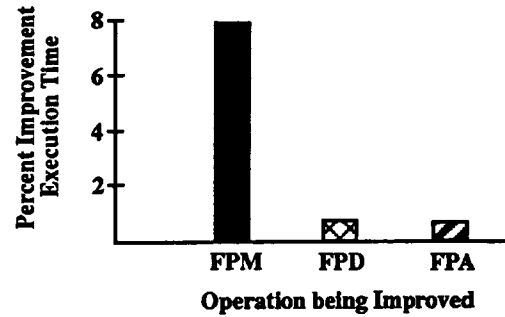


Figure 8. Improvement in Orientation Algorithm

5.3 Evaluating Hardware/Software Alternatives

A system [30] is being developed to track light movement across a screen. The system is an example of a real-time application whose major processing steps are illustrated in Figure 9. A position detector outputs four analog current values, up (U), down (D), left (L), and right (R), based upon the position of a light emitter on a screen. These four current “directions” are sampled N_s times at a rate of f_s , corresponding to a sampling interval of T_s . After performing analog to digital (A/D) conversion of the sampled data, the values are transferred to memory. Next, a spectral analysis is performed on these values using a fast fourier transform (FFT), once for each direction. The amplitudes of the FFT data are used to update the (x,y) position of the light on the screen. A constraint imposed on the system is that the position update, that is, the entire process shown in Figure 9, must be performed every 1/30 sec, or 33 msec.

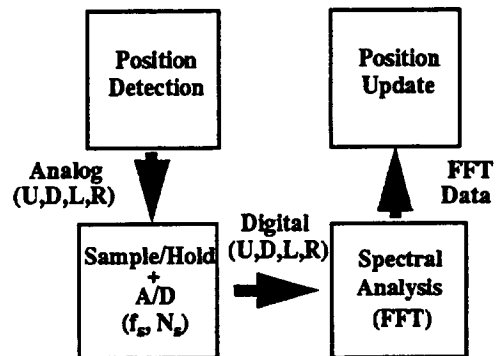


Figure 9. Processing steps in Tracking System

The most time consuming portions of the system are the sampling and the FFT computation. Of these two computations, the FFT is the most intensive. The remaining portions of the system take a relatively small amount of time and, as a result, can be neglected in the performance analysis.

Because the FFT is the bottleneck, it is analyzed in greater detail. Given f_s and N_s , a timing constraint can be established for the FFT. Assuming that $f_s=100\text{kHz}$ and $N_s=1024$, the time to perform the FFT computation, T_{FFT} , is approximately 20 msec.

There are several possible ways of implementing the FFT function. These alternatives [6] span a spectrum of different mixtures of hardware and software from general purpose to application specific solutions. An ADEPT model of the system was constructed to analyze some alternatives for the FFT. A portion of the top level description of the model is shown in Figure 10. The two parameterized blocks shown in the figure correspond to the sample and hold/analog to digital conversion and the FFT functions. To aid in the analysis, an ADEPT primitive called the *TIMECHKR* module was used to determine if any timing constraints were being violated.

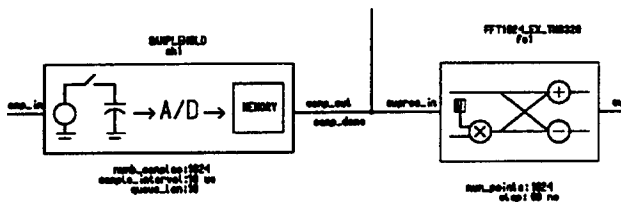


Figure 10. Portion of Top Level ADEPT Model

As an example, a hardware/software model of a decimation-in-time FFT algorithm executing on a general purpose processor was developed. The processor selected was the 20 Mhz MC68020/68881. In this model, no concurrency was exploited since the software model consisted of only floating point operations within a butterfly computation, which could only be executed serially with the MC68881 floating point unit. Through simulation, an approximate analysis revealed that a 1024 point FFT would take 220 ms, violating the time constraint.

The performance of the FFT function can be improved in several ways. One way is to use a specialized processor, such as a digital signal processor. Because the FFT computation consists of several butterfly operations, another possibility is to utilize an even more specialized butterfly processor [31]. At the extreme, a "hardware" (with perhaps some microcode) implementation of the FFT can be developed. Yet another alternative is the use of multiple processors.

The simulation results for some of these different hardware/software alternatives are provided in Figure 11. The execution times represent the time to perform 4 1024 point FFTs, where each FFT consists of only

butterfly computations. For these alternatives, a more detailed description of the FFT is required to assess violations of real time constraints.

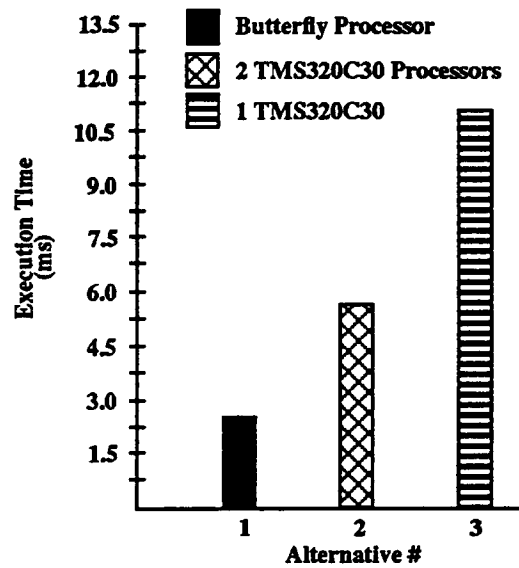


Figure 11. Simulation Results - FFT Alternatives

5.4 Analysis using Functional Models

In some cases, functional models are necessary to provide adequate information for analysis. This section presents the simulation results for a hardware/software framework [32] used to support parallel discrete event simulation (PDES) [33]. In this example, functional descriptions, independent of hardware or software, and appropriate abstractions are used in the analysis. Although the abstract hardware/software model is not used as in earlier examples, the model can be incorporated if desired.

Referring to Figure 12, the PDES framework consists of host processors (HPs), such as SUN SPARC workstations, which communicate via a host communication network (HCN) using messages. The HPs execute a discrete event simulation algorithm and interface to auxiliary processors (APs) using a dual-ported RAM (DPRAM). The APs execute synchronization algorithms and exchange information with the aid of a parallel reduction network (PRN) through a register interface (IN/OUT). The PRN is a synchronization network, consisting of a binary tree of pipelined ALUs, which can rapidly compute and disseminate information to the HPs. The ALUs are programmed to perform binary, associative operations, such as sum, minimum, maximum, logical AND, and logical OR.

The PDES framework incorporates an interesting hardware/software trade-off. The ability to quickly compute and disseminate global synchronization information is important in reducing the total time required to perform a PDES. One option is to perform global operations, such as minimum or maximum, using software running on the HPs. However, the time to perform global reductions on existing parallel architectures can be costly. For example, the time to perform a global reduction operation using barriers for a 32 processor Intel iPSC/2 is on the order of 10 milliseconds [34]. Another approach, represented by the PDES framework, is to perform the global operations in special purpose hardware (the PRN).

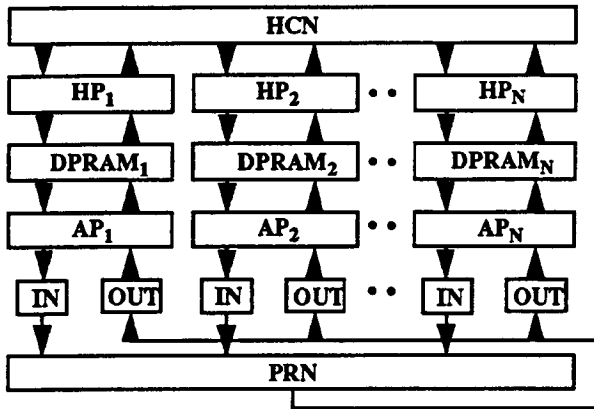


Figure 12. PDES Hardware/Software Framework

An eight node ($N=8$ HP/AP pairs) shared bus model for this framework was developed. Except for the HPs and the APs, ADEPT modules were used to model all portions of the framework. The HPs and APs were described as functional, concurrent processes written in custom VHDL, independent of hardware or software. Delays were inserted within these descriptions to model the time to perform various computations. Also, code was included that allowed them to be interfaced with other ADEPT modules.

The PRN model (see Figure 13) was simplified, supporting only the minimum operation. The model consists of three pipelined stages (levels) since $N=8$. Each ALU within the PRN performs a minimum operation on its inputs and has a stage delay that can be parameterized. In the model, this delay was set to 150 nanoseconds. Note that these ALUs perform functional transformations on the inputs, and therefore, are not the same as the abstract resources used in earlier examples.

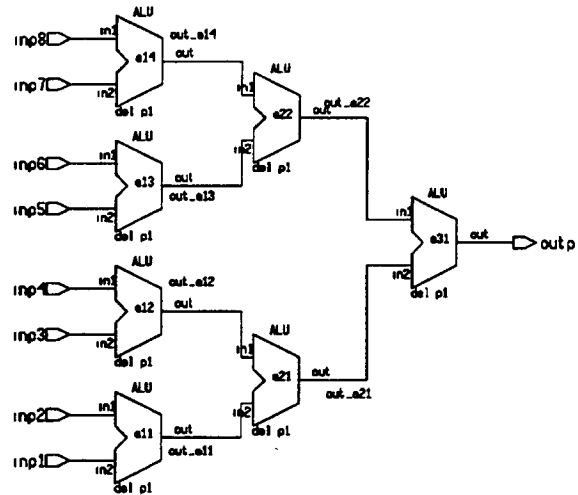


Figure 13. Parallel Reduction Network

To reduce message traffic in the HCN, the APs and the PRN are utilized for performing acknowledgments of messages sent between HPs. A two phase acknowledgment protocol, requiring two global reductions through the PRN, is used for message acknowledgments. When more than one AP tries to perform acknowledgments simultaneously, message acknowledgments become serialized through the PRN. In the worst case, all APs perform acknowledgments simultaneously. Thus, an important issue is analyzing the effect of this serialization on the performance of the framework.

The model was used to analyze the effect of serialization on the framework. For the worst case scenario, the simulation results in Figure 14 display the serialized acknowledgment times for messages sent between HPs in "chained" fashion (1 to 2, 2 to 3, and so on). In other words, after executing an event, an HP would send a message to its neighbor on the right.

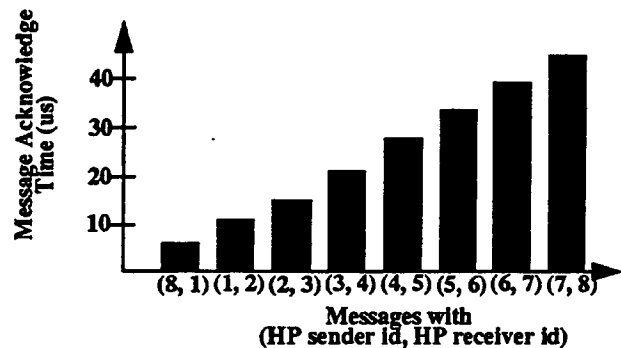


Figure 14. Serialized Message Ack. Times

6. Summary

To address the separation between the hardware and software domains, this paper has presented an abstract hardware/software model employing a unified representation using concepts that are familiar to both domains, namely, those based on data/control flow. A unified representation supports a common view of hardware and software, one that can be understood by both hardware and software developers. In addition, by utilizing a unified representation, techniques from one domain can be applied to the other, such as those utilized for performance analysis, reliability analysis, and formal verification of correctness. For example, by modeling hardware components using data abstraction, the rich theory associated with verifying the correctness [35] of implementations can be applied to hardware.

Because the execution of instruction set level descriptions can take substantial amounts of time, this paper has developed hardware/software abstractions to support early performance evaluation and trade-off exploration. These abstractions allow systems to be evaluated at different levels of detail with respect to multiple metrics. The amount of detail within a model is influenced by both the level of interpretation (uninterpreted versus interpreted) and the level of abstraction. The techniques discussed in this paper have been incorporated into the ADEPT environment.

An important issue in the modeling of complex systems is deciding how much detail is necessary. In the ADEPT environment, the modeler determines what level of detail is appropriate. Further research is required to better understand how the level of detail within a model influences the model's fidelity, as well as the trade-off between simulation time and model fidelity.

Acknowledgments

The authors would like to acknowledge the partial support provided by the Semiconductor Research Corporation under contract number 93-DJ-152 and the National Science Foundation under grant number CDA-8922545.

A number of dedicated students have contributed to the development of the ADEPT environment. These students include Ramesh Rao, Sanjay Srinivasan, Eric Cutright, Gnanasekaran Swaminathan, Richard MacDonald, Maximo Salinas, Charles Choi, Ambar Sarkar, Moshe Meyassed, and Bob McGraw. Thanks also to Robert Klenke, Ron Waxman, and Joanne Dugan for stimulating codesign discussions and their support of this effort. A special thanks is extended to Peter Schaefer for discussions regarding the stylus tracking system.

References

- [1] Franke, D. W., M. K. Purvis, "Hardware/Software Codesign: A Perspective," *Proceedings of the 13th International Conference on Software Engineering*, May 13-16, 1991, pp. 344-352.
- [2] Roman, G., et al., "A Total System Design Framework," *IEEE Computer*, May 1984, pp. 15-26.
- [3] Smith, C. U., L. G. Williams, "Software Performance Engineering: A Case Study including Performance Comparison with Design Alternatives," *IEEE Transactions on Software Engineering*, Vol. 19, July 1993, pp. 720-741.
- [4] Kumar, S., J. H. Aylor, B. W. Johnson, W. A. Wulf, *The Codesign of Embedded Systems - A Unified Hardware/Software Representation*, Kluwer Academic Publishers, Boston, Massachusetts, 1996.
- [5] De Micheli, G., "Extending CAD Tools and Techniques," *IEEE Computer*, January 1993, pp. 84-87.
- [6] Kumar, S., J. H. Aylor, B. W. Johnson, W. A. Wulf, "A Framework for Hardware/Software Codesign," *IEEE Computer*, December 1993, pp. 39-45.
- [7] Aylor, J. H., R. Waxman, B. W. Johnson, R. D. Williams, "The Integration of Performance and Functional Modeling in VHDL," in *Performance and Fault Modeling with VHDL*, J. Schoen, ed., Prentice-Hall, Englewood Cliffs, N. J., 1992.
- [8] Kumar, S., R. H. Klenke, J. H. Aylor, B. W. Johnson, R. D. Williams, R. Waxman, "ADEPT: A Unified System Level Modeling Design Environment," *Proceedings of the 1st Annual RASSP Conference*, Arlington, Virginia, August 15-18, 1994, pp. 114-123.
- [9] IEEE, "IEEE Standard VHDL Language Reference Manual," New York, NY, IEEE Standard 1076-1987, March 31, 1988.
- [10] Jensen, K., "Colored Petri Nets: A High Level Language for System Design and Analysis," in *High-level Petri Nets: Theory and application*, K. Jensen and G. Rozenberg (Eds.), Berlin: Springer-Verlag, 1991, pp. 44-119.
- [11] Gupta, R. K., C. N. Coelho Jr., G. De Micheli, "Synthesis and Simulation of Digital Systems Containing Interacting Hardware and Software Components," *29th Design Automation Conference*, June 1992, pp. 225-230.
- [12] Gupta, R. K., G. De Micheli, "System-level Synthesis using Re-programmable Components," *Proceedings of the European Design Automation Conference*, March 1992, pp. 2-7.
- [13] Ernst, R., J. Henkel, T. Benmer, "Hardware-Software Cosynthesis for Microcontrollers," *IEEE Design and Test*, December 1993, pp. 64-75.
- [14] Buchenrieder, K., "Codesign and Concurrent Engineering," in *Hot Topics of IEEE Computer*, January 1993, pp. 84-87.
- [15] Kumar, S., J. H. Aylor, B. W. Johnson, W. A. Wulf, "Object-Oriented Techniques in Hardware Design," *IEEE Computer*, June 1994, pp. 64-70.
- [16] Franke, D. W., M. K. Purvis, "An Overview of Hardware/Software Codesign," *International Symposium on Circuits & Systems*, May 1992, pp. 2669-2671.

- [17] Smith, C. U., R. R. Gross, "Technology Transfer between VLSI Design and Software Engineering: CAD Tools and Design Methodologies," *Proceedings of the IEEE*, Vol. 74, No. 6, June 1986, pp. 875-885.
- [18] Huck, J. C., M. J. Flynn, *Analyzing Computer Architectures*, IEEE Computer Society Press, Washington, D. C., 1989.
- [19] Flynn, M. J., "Directions and Issues in Architecture and Language," *IEEE Computer*, October 1980, pp. 5-22.
- [20] Stankovic, J. A., "The Types and Interactions of Vertical Migrations of Functions in a Multilevel Interpretive System," *IEEE Transactions on Computers*, Vol. C-30, No. 7, July 1981.
- [21] Hoffman, R., "A Classification of Interpreter Systems," *Microprocessing and Microprogramming*, 12, 1983, pp. 3-8.
- [22] Flynn, M. J., "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers*, Vol. C-21, No. 9, September 1972, pp. 948-960.
- [23] Frank, G. A., et al., "An Architecture Design and Assessment System for Software/Hardware Codesign," *Proceedings 22nd Design Automation Conference*, 1985, pp. 417-424.
- [24] Linger, R. C., H. D. Mills, B. I. Witt, *Structured Programming Theory and Practice*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1979.
- [25] Jensen, R. W., "Structured Programming," *IEEE Computer*, March 1981, pp. 31-48.
- [26] Aho, A. V., R. Sethi, J. D. Ullman, *Compilers Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.
- [27] Meyyassed, M., R. McGraw, J. Aylor, R. Klenke, R. Williams, F. Rose, and J. Shackleton, "A Framework for the Development of Hybrid Models," *Proceedings 2nd Annual RASSP Conference*, pp 147-154, Arlington, VA, July, 1995.
- [28] Sholl, H. A., T. L. Booth, "Software Performance Modeling using Computation Structures," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 4, December 1975, pp. 414-420.
- [29] Miller, W. H., B. W. Johnson, "Automatic Classification of Aluminum Defects," Department of Electrical Engineering, University of Virginia, Technical Report No. UVA/5-38459/EE94, October 31, 1994.
- [30] Schaefer, P., R. D. Williams, "The Stylus Tracking Project," Department of Electrical Engineering, University of Virginia, Technical Report, 1995.
- [31] Owen, R. E., "A 15 Nanosecond Complex Multiplier-Accumulator for FFTs," *ICASSP '87*, 1987, pp. 527-530.
- [32] Reynolds, Jr., P. F., C. M. Pancerella, S. Srinivasan, "Design and Performance Analysis of Hardware Support for Parallel Simulations," *Journal of Parallel and Distributed Computing*, Vol. 18, August 1993, pp. 435-453.
- [33] Fujimoto, R. M., "Parallel Discrete Event Simulation," *Communications of the ACM*, Vol. 33, No. 10, October 1990, pp. 30-53.
- [34] Reynolds, Jr., P. F., "An Efficient Framework for Parallel Simulations," *International Journal in Computer Simulation* 2, 1992, pp. 427-445.
- [35] Wulf, W., M. Shaw, P. N. Hilfinger, L. Flon, *Fundamental Structures of Computer Science*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1981.