MODELS OF COMPUTATION FOR PERFORMANCE ESTIMATION IN A PARALLEL IMAGE PROCESSING SYSTEM

By

YUE YIN

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2000

To My Parents

ACKNOWLEDGMENTS

I would like to thank Dr. Ritter for providing the opportunity to work for him on the AIM project, and for all the help, advice, and supervision he offered.

Special gratitude goes to Dr. Schmalz for his great guidance and continuous encouragement that are essential to this research. His tireless patience, intelligence, and diligence have been invaluable.

Further thanks go to Dr. Wilson for his patient advisement and support throughout this research, and Dr. Dankel for his graciously agreeing to serve on my committee and careful review of my thesis.

And many thanks go to my parents and my sister for their loving support and encouragement. Also, I would like to thank all my friends for their kind assistance and constant support.

TABLE OF CONTENTS

page

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

MODELS OF COMPUTATION FOR PERFORMANCE ESTIMATION
IN A PARALLEL IMAGE PROCESSING SYSTEM

By

Yue Yin

August 2000

Chairman: Dr. Gerhard X. Ritter
Major Department: Computer and Information Science and Engineering

Image and signal processing applications usually require high computational power. One common approach is to partition image data across several processors, and then to process each partition in parallel. Estimating the performance and power consumption of processors is necessary to obtain high performance.

In this thesis, we describe an infrastructure and methodology for estimating system performance by hierarchical modelling. We develop several models of computation for underlying hardware modules such as bus, memory and CPU, based on a thorough study of the characteristics of various hardware objects. The system model is an ensemble of these modules.

The hardware architecture is divided into three levels: the host, the unit, and the processing elements (PEs). These three hardware levels accept three sets of instructions: AIM Server Call (ASC), assembly-language-like instruction (ASM), and Bit Vector Call

(BVC), respectively. The implementation is realized by accomplishing ASC $\rightarrow$ ASM $\rightarrow$ BVC translation hierarchy and then performing a simulation on the system constructed from those modules based on the models of computation.

After running simulations with the EBLAST (Enhanced Blurring, Local Averaging, and Thresholding) compression algorithm, the simulation results are represented in the accumulated computed time vs. elapsed time graphs. The results show the differences between I/O at each level, data vs. instruction I/O at each level and across all levels, as well as computational cost vs. I/O cost at all levels.

CHAPTER 1
INTRODUCTION

1.1 Background and Justification

Image and signal processing (ISP) tasks are usually computationally intensive, because of (a) the large amount of data that must be processed and (b) the complexity of modern image processing algorithms. The space and computational requirements of image processing algorithms can be observed from the following example. For a 1,024 × 1,024 image, up to 3Mbytes of data must be processed for an RGB-encoded color image. For an N × N-pixel image, the computational complexity is at least of the order $\boldsymbol{O}(N^2)$ for simple operations such as pointwise arithmetic or local operations. The computational complexity is much higher for more complex tasks. For example, edge detection, with an M × M-pixel mask, an important operation, has a complexity of $\boldsymbol{O}(N^2M^2)$. The execution time of these tasks on a general purpose computing platform is very high compared to the desired real-time response. Thus, the usual performance limits of general purpose computers can be easily reached in image processing applications. Currently, parallel and distributed processing (PDP) provides a practical solution to these limits [BOU99].

There is a wide range of PDP architectures, which can be classified by the topology of the interconnection network, type of processing elements (PE), and the computational model employed. The topology of the network includes the logic layout of the PE nodes and the communication paths connecting PEs. Reconfiguration is an additional issue to be considered, which facilitates dynamically changing hardware

organization, designed to be flexible enough to accommodate different computational tasks without hardware redesign.

The Adaptive Image Manager (AIM) project [SCH98] developed at University of Florida's Center for Computer Vision and Visualization (UF/CCVV) is designed to distribute the computational process across multiple machines to take advantage of parallelization. AIM maps image and signal processing (ISP) algorithms across a heterogeneous network of processors. All or a portion of such hardware could be PDP hardware, for example, reconfigurable Single Instruction stream Multiple Data Stream (SIMD) processor and Symmetric Multi-Processor (SMP), as well as machines based on Field-Programmable Gate Arrays (FPGAs).

The performance of such architectures is a major concern during the design and development stages as well as after the completion of the system. Simulation, the most common approach to estimate the system performance, is the process of developing a model that captures the system characteristics, coding and running a program that behaves like the model, and observing the program's estimates of actual system performance. Simulation is a good choice for performance evaluation due to flexibility and possible application to a wide variety of situations, including existing and under-development systems. Also, via software simulation techniques to estimate hardware performance instead of building and testing hardware, significant cost savings are realized.

1.2 Key Assumptions

ISP Algorithms in AIM are specified using Image Algebra [RIT96]. Image Algebra provides a mathematical basis for specifying image processing algorithms in a

rigorous, concise notation that unifies linear and nonlinear mathematics in the image domain. Several lines of image algebra operations can replace many more lines of underlying source code (written in a programming language such as C++), so that describing algorithms in image algebra is easier than writing an equivalent C++ program. Algorithms in image algebra are also more readable and easier to maintain.

In AIM, based on the architectures of PDP hardware objects such as SMP, SIMD and FPGA, a generalized architecture model has been constructed for the purpose of estimating timing of various ISP operations. The model is divided into three levels, PDP hardware Host, PDP hardware Unit, and PEs, as shown in Figure 1. These three different hardware levels accept different sets of operations, AIM Server Call (ASC), assembly-language-like instruction (ASM), and Bit Vector Call (BVC). ASC is the high level instruction accepted by the host. ASMs are the assembler-like representation of hardware instructions. BVCs are the low-level bit vectors that actually control the PEs.

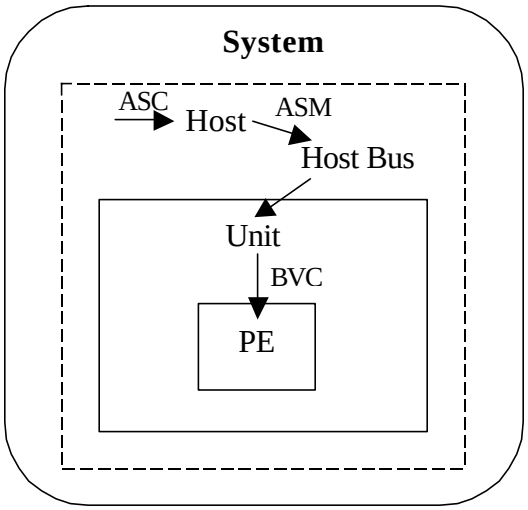

Figure 1: Hardware architecture

Symmetrically, image processing tasks are usually divided into a three-level hierarchy comprised of high, intermediate, and low-level instructions, based on the computational characteristics. High-level instructions consist of image or template operations. Operations over an image partition are intermediate-level. Low-level instructions are typically pixel-based. Therefore, this ASC-ASM-BVC instruction hierarchy can be thought of as reflecting the logic structure of the application.

The AIM models of computation are built on the proceeding hardware architecture and instruction hierarchy, and address the common-case effects and attributes. For example, a communication channel model considers the channel bandwidth, overhead, error rate, and switch type. In a memory module, the I/O protocol, capacity, and bandwidth are taken into account. The computational bandwidth, degree and type of parallelism, and I/O are studied for the processor.

Because the model of computation generalizes a SIMD or FPGA model, and because it both captures the main factors determining the system performance and considers the detailed design, the simulation based on the model of computation is designed to be realistic. The results of the simulation are expected to be within ±10 percent of physical reality.

1.3 Methodological Overview

In the AIM project, we have developed models of computation for SIMD and FPGA processors. It would be useful to extend the underlying hardware to simulate a wider range of systems, by connecting hardware objects in a network-like structure to simulate the structure of user-specifiable hardware systems. This thesis focuses on estimating the performance and power consumption of a system constructed from

heterogeneous components. We first develop software models for the performance of different hardware objects, such as bus, memory, and CPU, then compose these models and perform simulation on them to get performance estimation.

Timing is crucial in modelling a computer system. Also, performance is closely related to cost, that is, a computer system should have adequate performance at reasonable cost. High performance usually requires high cost, and low cost often sacrifices system performance [HAR90]. Additionally, power modelling is another important aspect of performance modelling.

In AIM, mapping of the algorithms to the hardware proceeds as follows. First, the system host accepts the ISP algorithms, i.e., a sequence of ASCs, and then transforms each ASC into a collection of ASMs, incurring a time delay. After being transmitted through the host bus and entering the unit, each ASM is transformed into a stream of bit vectors (BVCs) in the unit, also incurring a time delay. BVCs are transmitted to the next lower hardware level, the processing elements (PEs). Finally, an ALU or FPGA logic block executes the BVCs. This decomposition procedure is shown in Figures 1 and 2.

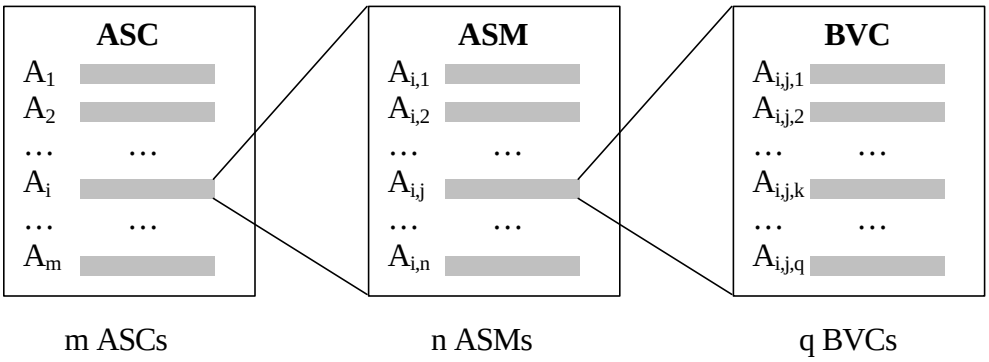

Figure 2: Instruction hierarchy

The models of computation are driven by the proceeding three-level instruction hierarchy (ASC-ASM-BVC). The decomposition of ASC to ASM to BVC involves detailed attributes and variables of the system such as the precision of instructions and data, instruction translation time, computation time, data transfer delay, parallel communication mechanism, channel and processor bandwidth, component failure rate, etc. As mentioned previously, the simulation procedure goes through the three hardware levels, Host, Unit and PEs, which is based on the decomposition procedure. Therefore, the models of computation are built on the three-level instruction hierarchy.

Another consideration for the models of computation is the interface requirement with the AIM debugger and display, such that simulation results must be displayed and analyzed. Because the models of computation are supported by the hardware architecture and reflect the logical structure of applications, the interface must be carefully designed to be congruent with all aspects of the model.

To perform simulations on computer systems is the goal of models. We study a wide range of hardware components, including various kinds of buses, memories, switches, and CPUs. Special attention is paid to the main computational hardware, such as FPGA and SIMD. The characteristics and performance attributes for each hardware object are obtained so that an accurate simulation can be performed. So, the models of computation are also designed to simulate salient functionality of hardware at Control Logic Block (CLB) of FPGA and ALU of SIMD levels.

This thesis is organized as follows. In Chapter 2, recent research on performance modelling of hardware objects is overviewed. Chapter 3 presents the models developed in this project. Chapter 4 describes how these models are implemented. Chapter 5

presents example simulation results and analysis. Finally, Chapter 6 contains conclusions, open issues, and possible future work.

CHAPTER 2
RELATED RESEARCH

Performance is the most important concern with a computer system. During system design and development, direct measurement of performance is not feasible, hence performance modelling is necessary. Modelling requires (1) capturing the main factors determining system performance, (2) representing them in a model, (3) determining performance measures in the model, and (4) using these measures from the model as performance estimates of the actual system.

There are basically two performance analysis techniques: analytical modeling and simulation. Analytical modeling formulates an analytical expression that solves a given model, then the performance data can be evaluated from this expression, whereas simulation involves the formulation of a computer program and its execution. Given the requirements imposed by the fast evolution of computer architectures, performance modelling and simulation is a major research area that has been in constant development over the past four decades. Salient research is reviewed, as follows.

2.1 Performance Modelling

Clement and Quinn [CLE97] developed a dynamic performance prediction methodology to analyze parallel systems. The model first inserts instrumentation code to gather execution statistics and to arrive at a symbolic equation for the execution time of each basic block. After analyzing the major architectural features, such as on-chip cache and page fault behavior, message startup time, and bandwidth characteristics, the

expressions for operation counts are computed as a function of the problem size and the number of processors. The total execution time can be predicted from the counts and cost of each operation. Statistical techniques are used to estimate the cost for each operation for a dynamic performance prediction model. The compiler-generated analytical model accounts for the effects of cache behavior, CPU execution time, and message passing overhead for realistic programs written in high level data-parallel languages. Experimental results show this dynamic performance prediction technique is effective in analyzing general and scalable applications of Massively Parallel Processing (MPP) systems.

Noh, Dussa-Zieger, and Agrawala [NOH99] suggested a heterogeneous mixed-mode (HeMM) model for the possibility of providing different processing power within a single tightly coupled computing system. The model consists of a data parallel component that is composed of a massive number of slow processors and a control parallel component that is composed of a small number of fast processors. Performance analysis includes computation time and communication time for both single and multiple level transfer, as follows:

1. The execution time of a parallel application on a HeMM system is bounded by

$$T_{cps} = (\beta i f_i T) / [\delta(n_f) n_f],$$

where β is the ratio of time to execute a unit of computation on one processor of control parallel component to the execution time on one processor of data parallel component, T is the execution time of a parallel algorithm on the data parallel component, i is the number of processors executing in parallel, f_i is the fraction of T such that i processors are executing in parallel, n_f is the

number of processors on the control parallel component, and δ is the efficiency factor function;

2. The communication time is given by

$$T_{dps} = T_{sd} + T_{rd},$$

where T_{sd} and T_{rd} are the times required to respectively send and receive a unit of data on a processor of the data parallel component.

Zaleski [ZAL96] presents a methodology by which the number of executed lines of C code associated with a given application are transformed into an estimate of processing time on the IBM RS/6000 990 processing platform. The total number of machine instructions processed per line of executed C code, $N_{I/C,}$ is computed by analyzing the specific code and determining the number of executed lines N_C, then dividing N_C into the total number of processed instructions N_I. The actual performance can vary depending on the specific C functions, the level of optimization, the number of users on a platform at a given time, and the number of background processes. A mathematical expression is derived which relates the MIPS (Millions of Instructions Per Second) and MFLOPS (Millions of Floating-point Operations Per Second) rating of a given processor to the CPU processing time t_{CPU} associated with a C program up to a million lines of code:

$$t_{CPU} = (N_{I/C})(N_{ELOC})[F/MFLOP + (1 - F) / MIP],$$

where N_{ELOC} is the number of executed lines of C code, F is the fraction of the code which is determined to be floating point, MFLOP is MFLOP rating of the target processor, and MIP denotes the MIP rating of the target processor.

Previous research has also investigated models that focus on selected hardware objects of an overall system, as shown in the following sections.

2.2 Bus and Memory Modelling

A network model can be constructed by aggregating several instances of a model of a single switch or stage of switches. For example, Harper and Jump [HAR90] first specify and solve a queuing network approximation of a 2×2 crossbar switch. Next, a resource model is developed and combined with a single switch to form the network model. To evaluate performance of the network, contention between multiple switch outputs for a common resource bus is modeled. Access to the bus is granted to each packet requesting the bus with equal probability without consideration of aging or priority schemes. Blocking for the output link consists of two parts: contention within the switch, and contention among the switch outputs for access to each resource bus. Experimental results suggest that a slower network could be used to reduce system cost without sacrificing system performance. Therefore, hybrid networks provide a simple method of incrementally trading network bandwidth for network cost.

2.3 Network Performance Modelling

Interconnection network types such as Mesh, Torus, and Hypercube are generalized as a k-ary n-cube in Xiao's research [XIA97]. Each node consists of a process, a switch, a router, and some channels associated with input queues and output buffers, for example, the MPP system supports message passing, store and forward, and wormhole flow control mechanisms. The Analysis model takes account of the network properties (such as topology and channel width), the effects of message size and message blocking. During simulation, the queue corresponding to each network output port is

treated as a queuing server. Simulation models are implemented in SLAM and C. The results show that when the bisection width is constant, which is the minimum number of wires that must be cut to separate the network into two equal halves, the network latency of three or four dimensional networks is shorter than that of other dimensional networks. When the messages that have not been accepted by their destinations increase, the network efficiency is sensitive to a threshold such as the number of outstanding requests.

The performance of a parallel application running on a network of workstations depends on the number and type of the workstations, local workload at each workstation, network connectivity, and scheduling algorithm used to allocate parallel tasks to the workstations.

In Menasce and Rao [MEN96], a hybrid analytic and simulation model is used to predict the performance of parallel applications specified by a task graph on a network of workstations. The model computes the total execution time of the parallel application given a task graph, a network of workstations, a scheduling policy, and owner job information. The model first accounts for owner job interference, then accounts for interference from other parallel tasks, and finally computes parallel task execution. The results show that the computing demand of the interactive workload has a non-linear effect on the execution time of the parallel application, that the slower workstations show an increasing degradation as the number of concurrent owner submitted commands increases. Additionally, simulation indicates that the execution time of the parallel application decreases dramatically as the sleep time of owner workload increases.

Karatza [KAR94] proposed a cyclic queuing network model of a multiprocessor system consisting of an I/O channel and two single-server homogeneous processors

linked in tandem (i.e., in series). Each processor is equipped with its own queue. Both the nonblocking case and the blocking case are compared, depending on the capacity of the second processor queue. The effects of blocking on the overall system and program performance in relation to nonblocking for various degrees of multiprogramming and coefficients of variation of the CPU service times are studied. The performance of the model is measured using CPU response time of a random job, mean cycle time, system throughput rate, and the utilization factors of the CPU processors. The results show that the blocking mechanism slightly degrades system performance in relation to nonblocking, and this deterioration depends on both the degree of multiprogramming and the coefficient of variation of the CPU service times.

A closed queuing network model [KAR97] can represent a distributed computing system that is comprised of several loosely-coupled workstations communicating over a network. The model consists of the CPU and the I/O unit. The CPU consists of four identical and independent processors each serving its own queue. A single I/O channel which has the same service capacity with the CPU is used. Each task is assigned to a processor queue according to the task routing policy. Of four task routing strategies, two are static: Probabilistic (P) and Round Robin (RR), and two are adaptive: Shortest Queue (SQ) and Shortest Response Time Queue (SRT). Performance of these strategies is estimated with discrete event simulation models using the method of independent replications. Every simulation model was repeated 30 times for 10,000 CPU service time requests in each time, with the same starting conditions but with different streams of random numbers. The results show that the two adaptive strategies are superior to the two static techniques, but their performance advantages are not completely exploited due

to subsequent resequencing delay of jobs. This delay is much higher with these policies than with the static ones.

2.4 CPU and Multiprocessor Performance Modelling

Obaidat and Abu-Saymeh [OBA93] presented a simulation model for the RISC-based multiprocessor system. The RISC processing elements are simulated using a detailed model that could be used to generate a memory reference trace by running benchmark programs on the processor models. All units and stages of pipelines are modeled as resources, with a first come first serve (FCFS) queue. Data buses, address buses and internal buses are also modeled as resources. To model data dependencies, all operands are modeled as resources. For multiprocessor systems, an interconnection network (IN) is simulated as a decision routine. Each processor is simulated using the previous model with a local instruction memory and a shared data memory. Multiprocessor system performance is studied using two types of INs: delta and crossbar switch. Theoretical benchmark programs are created with various alpha ratios. The main performance measures used are the speedup, mean waiting time, probability of blocked requests and relative speedup. The results show that the relationship between the speedup and the number of processors is linear for a given alpha ratio.

Wang, Wu, and Nelson [WAN90] investigated three MIMD computing surfaces, namely, torus, augmented torus, and grid-bus machine. In the augmented torus, each PE can use either links or buses for communication. In the torus, each PE can only use links for communication. In the grid-bus machine, each PE can only use buses for communication. All three machines operate as message-passing MIMD machines and each processing element has its own private memory. The generalized stochastic Petri

nets (GSPN) modeling scheme is used to probabilistically model the behavior of the three MIMD architectures. Each PE has three states: active, communicating, and queued. The processing power, defined as the average number of processors executing in their private memory, is used to measure the performance of the augmented torus. Solution of a 2-D n^2-point FFT problem exemplified the architectural merits and demerits of the three MIMD architectures in a deterministic method. The two modeling methods generate the same conclusion: the augmented torus has better performance than the torus and the grid-bus.

For reconfigurable systems, the performance can also be evaluated through simulation and analysis. Kwiat [KWI9] modeled and simulated the architecture of Dynamic Reconfigurability Assisting Fault Tolerance (DRAFT), which is composed of the following main components: a set of Computing Modules (CMs) (each containing a processing unit and memory), an FPGA, an FPGA controller, and ROM. The host communicates with DRAFT through the host interface unit. In simulation, a concurrent VHDL process model exhibits the following parallel behavior. Applications arrive at an input queue and are dispatched with delay D_1 to a subset of the N CMs, delay D_2 denotes the time required for message passing and output service. The analytic approach to performance modelling uses a Markov model for tracking the number of applications completed. The results obtained from the simulation compare closely with results obtained from analytic models.

Ratha's study [RAT97] emphasizes the Splash 2 architecture. Splash 2 is an FPGA processor board that is connected to the host through an interface board that extends the address and data buses. Each processing board has 16 processing elements

(PEs) in addition to a seventeenth PE which controls the data flow into the processor board. Each PE has 512KB of memory. The PEs are connected through a crossbar switch. There is a 36-bit linear data path (SIMD Bus) running through all the PEs. In simulation, the reconfigurable logic is specified in VHDL. The results of the VHDL simulation are compared with those obtained manually or by a sequential program. In synthesis, the main concern is to achieve placement of the logic in an FPGA that minimizes the timing delay. Three representative examples are used to demonstrate the mapping of vision algorithms onto Splash 2. As a low-level vision algorithm, a generalized 2-D convolution is implemented. For intermediate level vision, a texture-based segmentation algorithm is implemented. Point pattern matching represents a high-level vision process. All three examples come to the same result: when comparing the implementation on Splash 2 with implementations on different platforms such as SPARC-20, i-860, CM-5, etc, the execution time on Splash 2 is much less than the others. Besides the efficient mapping of algorithms, the examples also demonstrate the suitability and superiority of custom computing approach for all levels of vision algorithms.

CHAPTER 3
MODELS OF COMPUTATION

3.1 Overview of Model Strategy

A model is a representation of a system intended to enhance our ability to understand, analyze, predict and evaluate the behavior of the system. The ability to gain sufficient knowledge of a system by analyzing system models depends on how successful the model captures the essential features of the system in a scientific way. Construction of a clear, logical, and unambiguous model is thus an important, scientifically oriented process.

3.1.1 Comparison of Two Modelling Approaches

Based on the different viewpoints models represent, there are two primary methods in system modelling, namely, (1) the black box approach and (2) the componentwise approach.

The black box approach builds a model based on observations of system behavior and statistical analysis without actually explaining the properties of individual system components. It treats the system like a closed box with an internal structure that is unknown. The output is expressed in terms of inputs.

The black box approach is convenient and can estimate or predict system performance without one knowing the internal structure of a system. However, its applications are limited to reasonably stable and closely specified.

The componentwise approach, on the other hand, builds a model describing the system based on parameters of individual system components and knowledge of internal system structure.

Models built by the componentwise approach tend to be more flexible than models constructed with the black box approach, because there are more degrees of freedom for each component. The componentwise approach can be used in most applications, but it may be harder to express the model due to a complicated arrangement of objects within the model.

In this study, we choose the componentwise approach because

1. AIM requires detailed knowledge of each component, i.e., the functionality of each individual device. In contrast, the black box approach tells us little about intra-component functionality, and is thus only a general model.
2. The underlying hardware objects in AIM can be flexibly designed, since in a physical model, the hardware is user-specified. In contrast, when a device specification changes, the black box model must be recalibrated; but the componentwise model parameter(s) only require(s) modification.

3.1.2 Hierarchical Modelling

Computer systems like AIM often have high structural complexity. The performance evaluation of these systems, based on single-level unstructured models, may not be sufficiently adequate in terms of model definition or analysis. Instead, a structured or hierarchical modelling method can be applied to avoid a single-level view of the system.

The hardware components most frequently used in practice are buses, CPU, and memory, which are the most important components in computer system. In AIM, we decompose the system model (the top level) into three major models (secondary level): bus, CPU, and memory models. This hierarchical structure is shown in Figure 3.

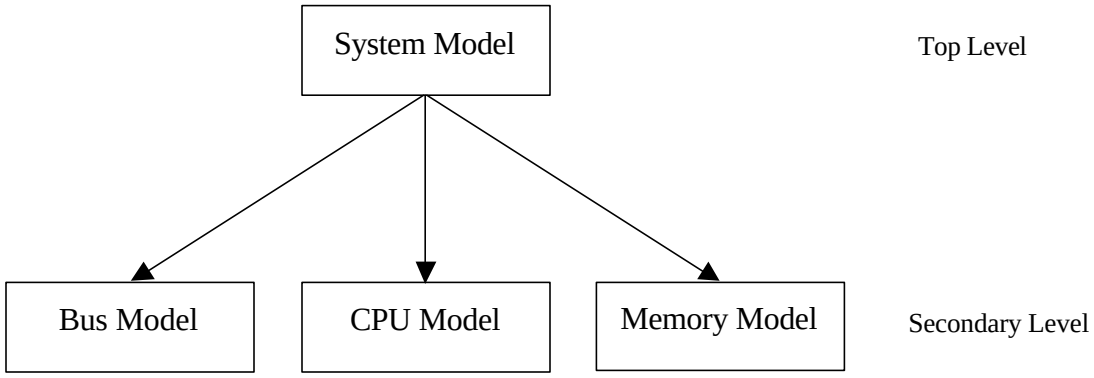

Figure 3: Hierarchical structure of AIM MOCs

Hierarchical modelling has several advantages:

1. *Consistency with hierarchical system structure*—A component in the system can be further decomposed into sub-modules. With hierarchical modelling, it is easy to partition a model into submodels. This partitioning procedure can be iterated downward through the model hierarchy to obtain a model at the appropriate level of detail.
2. *Physical fidelity and diversity*—The properties and performance of different hardware components often differ widely, and cannot be portrayed using the same model. In practice, a single hardware component may have different types, where each type has its own instance of a basic model. Therefore, the diversity of the components also supports hierarchical modelling.

3. *Modularity*—Each model can be built and analyzed in isolation with little effects to other models. If we want to change one type of hardware object to another (e.g., change a synchronous bus to an asynchronous bus), we only need to replace the original object model (e.g., synchronous bus model) with the new model (e.g., asynchronous bus model).

Thus, with hierarchical modelling, the system behavior can be easily observed and studied.

3.1.3 Model Validation

Model validation is the process of substantiating that the model within its domain of applicability is sufficiently accurate for the intended application. A model is of little practical use without validation, because one cannot determine whether or not the results obtained from the model are accurate.

To determine how accurately a model represents a system, simulation is carried out to obtain the resultant predicted behavior of the system. Additionally, observed performance data is obtained by measurement of physical system parameters. The predicted behavior (simulation results) is compared with that of the real system (measured or observed data). If the differences between model and system behaviors are within the range of accuracy with respect to the nature of the system, then the model is said to be validated. This validation process ideally occurs with a large sample of random input. If time or test resources are insufficient, a typical subset of input encountered in practice is often used. If this subset corresponds statistically to typical physical input, then validation is said to be physically accurate.

3.2 Bus Model

In a system, the various modules must be interfaced to each other. For example, the processor and memory/buffer need to communicate, as do the processor and the I/O devices. Such communication paths are provided by a bus. As the degree of system parallelism increases, the overall system performance and cost is limited by the performance and cost of its communication paths [HEN95]. Therefore, a bus model is key to accurate estimation of the performance of the overall system.

In AIM, various kinds of buses are used to connect different components. A local bus provides the communication between ALU and local memory in a single processor. A host bus provides the common communication path among host and all add-on units. Multiple-bus or switched networks provide dynamic interconnections for multiprocessors, for example, SIMD mesh interconnection. Figure 4 depicts the examples of these three kinds of buses.

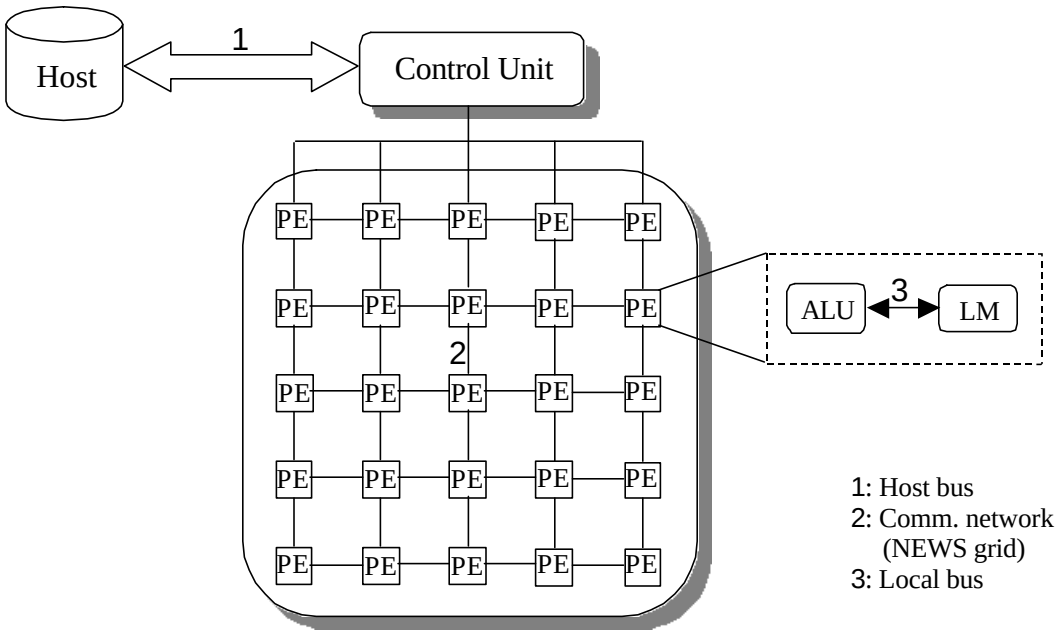

Figure 4: Bus system

To study the performance of different kinds of buses, we first specify some general attributes for the buses we consider in this study:

1) *Nominal bandwidth*: In an ideal situation, there is no bus failure or any kinds of communication overhead, the bus bandwidth equals the nominal bandwidth. Units are bits per second.
2) *Communication overhead*: In reality, there are costs for processing data, such as that incurred by error correction or encryption. These costs comprise communication overhead, which must be considered since each of these costs can degrade bus bandwidth.
3) *Bus failure rate*: Bus failure arises because of noise, intersymbol interference, etc. This is another key decrement to bus bandwidth.
4) *Synchronous or Asynchronous*: In synchronous timing, all bus transaction steps take place at fixed clock cycle according to the global clock used by all the devices on the bus. Asynchronous timing is based on a handshaking or interlocking mechanism. A synchronous bus is simple to control and costs less. It is suitable for connecting devices having relatively the same speed. The advantage of an asynchronous bus lies in the freedom of using variable length clock signals for different-speed devices. It offers better flexibility at the expense of increased complexity and costs.
5) *Switch*: For communication networks, switches are used to route data streams from one link to another. A network can transfer data using either circuit switching or packet switching. In circuit switching, once a device is granted use of a bus it occupies the bus for the entire duration of data transfer

[YAN90]. In packet switching, the data are partitioned into small packets and a bus is held only during the transfer of a packet. In AIM, only packet switching is used because communication links can be utilized more efficiently by packet switching.

In this study, we consider three kinds of bus models: (1) Linear Bus, (2) Packet Bus, and (3) Nonlinear Bus.

3.2.1 Linear Bus

A linear bus is the simplest way to connect several computational hardware modules. In its simplest form, a linear bus is comprised of one or more wires. In this study, we model basic effects, such as bus failure and overhead due to error correction or encryption, expressed as follows:

1) *Bus failure rate* (R_{BF}), denotes the probability of bus failure (open circuit). *Bus success rate* (R_{BS}) is computed from R_{BF}:

$$R_{BS} = 1 - R_{BF}.$$

2) *Security overhead* (O_S), expressed as a factor > 1, usually accrues from encryption and is computed as a penalty factor expressed in terms of the bus bandwidth. For example, if O_S reduces the bus bandwidth by 50%, then O_S = 2.

3) *Bandwidth fluctuation* (F_{BW}) ≥ 0 with respect to a nominal-bandwidth. For example, if the bandwidth fluctuates from 80 percent to 120 percent of nominal bandwidth, then $F_{BW} = \pm 0.2$.

Note that R_{BF}, O_S, and F_{BW} can vary with time, but not within the time slice specific to a given low-level instruction.

The *nominal bandwidth* (BW_{NOM}) of a bus is decreased by R_{BS}, O_S, and F_{BW} to yield the *modified bandwidth* (BW_{MOD}) as follows:

$$R_{BS}: BW_{MOD} = BW_{NOM} \cdot R_{BS},$$

$$O_S: BW_{MOD} = BW_{NOM} / O_S, \text{ and}$$

$$F_{BW}: BW_{MOD} = BW_{NOM} \cdot (1 \pm F_{BW}).$$

This leads to the simple linear model:

$$BW_{MOD} = BW_{NOM} \cdot (1 - R_{BF}) \cdot (1 \pm F_{BW}) / O_S .$$

The linear bus is useful as a model of simplest and least costly connection between a small number of modules, for example, in local bus. When the number of modules increases, the bus is often heavily loaded, has long propagation delay, and consumes significant amounts of energy.

The following bus models have been proposed especially for communication networks to (1) increase utilization, (2) offer high-communication bandwidth, and (3) reduce contention between partitions of information that must be retransmitted.

3.2.2 Packet Bus

For more effective use of communication resources, the idea of packet transmission was introduced. In packet transmission, a data stream is divided into fixed-length packets, which are sent out individually over the network through multiple paths, then marshalled and reassembled at the final location prior to being delivered to the user at the receiving end [SHE85].

Each packet must contain a destination address to facilitate routing. The sender's address and other control information are also appended to the original data to identify which part of which message the packer belongs, so that it can be reassembled at the

receiving end [NAP90]. The dividing and reassembling procedures cause *transmission overhead*. To ensure transmission reliability, mechanisms are included to detect and correct errors for each packet, which leads to *error detection/correction overhead*. These two overheads vary per packet.

In packet transmission, when two or more stations transmit a packet along the same channel during the same time period, this causes interference and destruction of these packets, called *collision*. The colliding packets must be retransmitted. Also, individual buses may fail.

Based on these properties of the packet bus, we propose the following attributes:

1) *Number of partitions or packets* (N_P): In packet transmission, the effective bandwidth is changed by dividing a data stream into partitions. N_P approximates the data stream length (N bits) divided by the packet size (K bits):

$$N_P = \lceil N / K \rceil.$$

In the absence of errors or overhead, the relationship between the nominal bandwidth (BW_{NOM}) and the effective bandwidth (BW_{EFF}) for packet bus is:

$$BW_{EFF} = BW_{NOM} \cdot N / (N_P \cdot K).$$

2) *Collision probability* (P_C): The probability of packet collision is denoted by P_C. For example, if 10 percent of packets collide and must be retransmitted, then $P_C = 0.1$. The effect of collision increases the number of packets as follows:

$$N_P' = \lceil N_P / (1 - P_C) \rceil.$$

3) *Failure probability* (P_F): The probability of bus failure is denoted by P_F. For example, if a bus fails 10 percent of time, then $P_F = 0.1$. Bus failure affects the bandwidth as follows:

$$BW' = BW \cdot \ (1 - P_F).$$

The aggregate effect of collision and failure is described as follows:

$$BW_{EFF} = \frac{BW_{NOM} \cdot (1 - P_F) \cdot N}{\left\lceil \frac{N_P}{(1 - P_C)} \right\rceil \cdot K}$$

4) *Transmission overhead* (O_T): O_T is a function on packet size K. In this study, f(K) is computed as a penalty factor. Here, we assume a linear model for f(K):

$$f(K) = C_1K$$

where C_1 is a constant. This effect reduces the bandwidth as follows:

$$BW' = BW / C_1K.$$

5) *Error detection/correction overhead* (O_D): O_D is, also, a function on packet size K, denoted by g(K), is computed as a penalty factor. Again, we assume a linear model for g(K):

$$g(K) = C_2K$$

where C_2 is a constant. Error detection/correction overhead reduces the bandwidth as follows:

$$BW' = BW / C_2K.$$

Combining the preceding expressions, we obtain the effective bandwidth for a packet bus:

$$BW_{EFF} = \frac{BW_{NOM} \cdot (1 - P_F) \cdot N}{\left\lceil \frac{N_P}{1 - P_C} \right\rceil \cdot K \cdot C_1K \cdot C_2K}$$

The primary benefit of packet transmission is sharing the data burden evenly and effectively throughout a communication network. The use of packets reduces the traffic contention and has the capability for transmission of large volumes of data with less propagation delay. Because of the multiple routing path capabilities, individual line failures will not cause the loss of a complete message path.

3.2.3 Nonlinear Bus

In the packet bus model, transmission overhead and error detection/correction overhead vary linearly with packet size. However, for some bus configurations, these relations are not linear. In this section, we refine the packet bus model to produce a nonlinear bus model. In this model, all the bus parameters except packet size, nominal bandwidth, and success/failure rates, are dependent on packet size. In addition, error correction overhead is dependent on the probability of bus failure.

In the nonlinear bus model, the number of packets (N_P), packet collision probability (P_C), and bus failure probability (P_F) are formulated the same as in the packet bus model. The following parameters are different from those in the packet bus model:

1) *Transmission overhead* (O_T): a nonlinear function on packet size K, denoted by $f(K)$. This effect reduces the bandwidth as follows:

$$BW' = BW / f(K).$$

2) *Error detection overhead* (O_D): a nonlinear function on packet size K, denoted by $g(K)$. It reduces the bandwidth as follows:

$$BW' = BW / g(K).$$

3) *Error correction overhead* (O_C): a nonlinear function on packet size K and bus failure probability P_F, denoted by $h(K, P_F)$. Error correction overhead reduces bus bandwidth as follows:

$$BW' = BW / h(K, P_F).$$

This leads to the following expression for effective bandwidth:

$$BW_{EFF} = \frac{BW_{NOM} \cdot (1 - P_F) \cdot N}{\left\lceil \frac{N_P}{1 - P_C} \right\rceil \cdot K \cdot f(K) \cdot g(K) \cdot h(K, P_F)}$$

3.3 Memory / Buffer Model

Memory is a portion of a computer system that is used for the storage and subsequent retrieval of data and instructions. A computer system is usually equipped with a hierarchical memory subsystem due to a tradeoff among the three key characteristics of memory, namely *cost*, *capacity*, and *access time*. A typical memory hierarchy includes buffer, cache, main memory, and secondary (external) memory [HEN95].

3.3.1 Buffer Model

A buffer is a temporary storage area shared by hardware devices that operate at different speeds. The buffer allows each device to operate without being delayed by another device. It exists not so much to accelerate the speed of an activity as to support the coordination of separate activities. A buffer is kind of a “dumb” memory, because it needs no memory management unit (MMU). Thus, a buffer is simpler and costs less than memory.

For a buffer to be effective, the size of the buffer and the algorithms for moving data into and out of the buffer need to be considered by the buffer designer. FIFO (first-in, first-out) and LIFO (last-in, first-out) are two approaches to move data into and out of buffers.

In this study, we consider the following parameters for our buffer model:

1) *Buffer width* (W_{BUF}): is the number of bits read out of or written into buffer at a time.
2) *Cycle time* (C_{BUF}): in unit of seconds, is the time between the start of one buffer access to the time when the next access can be started. It consists of the access time plus any additional time required before a second access can commence.
3) *Buffer failure rate* (R_{BF}): denotes the probability of buffer failure.

Then, we obtain the effective bandwidth for buffer model:

$$BW_{EFF} = (1 - R_{BF}) \cdot \ W_{BUF} / C_{BUF}.$$

3.3.2 Basic Memory Model

Memory is more complex than a buffer, requiring a MMU to manage access to the memory. All requests for data are sent to the MMU, which determines whether the data is in memory or needs to be fetched from the secondary storage device. If the data is not in memory, the MMU issues a page fault interrupt. The MMU holds the Translation Look-aside Buffer (TLB), which matches virtual address to physical addresses. Thus, the MMU has finite, nonzero overhead.

Memory is the destination of input as well as the source for output. A key design goal is to broaden the effective memory bandwidth so that more memory can be accessed

per unit time. Memory interleaving is used to take advantage of the potential parallelism of having multiple memory banks. When different addresses are presented to different memory banks, parallel access of multiple words can be performed simultaneously. Each memory bank is accessed once per cycle, so if two addresses to the same memory bank are presented at the same time, then a conflict results. Thus, a given access should wait until the other one is finished.

For purposes of simplicity, we model memory using the following parameters without consideration of cache effects:

1) *Memory width* (W_{MEM}): is the number of bits read out of or written into memory at a time;
2) *Cycle time* (C_{MEM}): is the time between the start of one memory access to the time when the next access can be started;
3) *Memory size* (S_{MEM}): is the total number of bytes or words in memory;
4) *TLB size* (S_{TLB}): is the length of the TLB, in bits;
5) *MMU overhead* (O_M): a function of TLB size S_{TLB}, is denoted by $f(S_{TLB})$. MMU overhead reduces the bandwidth as follows:

$$BW' = BW / f(S_{TLB}).$$

6) *Conflict probability* (P_{CF}): denotes the probability of memory access conflict. This increases the effective cycle time as follows:

$$C'_{MEM} = C_{MEM} / (1 - P_{CF}).$$

7) *Memory access failure rate* (P_{MF}): denotes the probability of memory access failure.

The following effective bandwidth for basic memory model is obtained:

$$BW_{EFF} = \frac{(1 - P_{CF}) \cdot (1 - P_{MF}) \cdot W_{MEM}}{C_{MEM} \cdot f(S_{TLB})}$$

3.3.3 Memory Model with Associative Cache

Cache is a special high-speed storage mechanism that a processor can access more quickly than it can access regular memory. A cache is made of smaller, faster, and more expensive hardware than memory.

Memory caching is effective because of the principle of locality. The cache contains a copy of a portion of main memory. As the processor attempts to read data from memory, the MMU looks first in the cache. If it finds the data there (from a previous reading of data), it does not have to do the more time-consuming reading of data from larger memory. If the data is not in the cache, a block of main memory, consisting of some fixed number of words is read into the cache [STA96].

Three approaches are used to place a block in a cache:

- *Direct*: each block has only one place it can appear in the cache.
- *Fully associative*: a block can be placed anywhere in the cache.
- *Set associative*: a block can be placed in a restricted set of locations in the cache. If there are *n* blocks in a set, the cache placement is called *n-way set associative*.

Because set associative cache captures the advantages of both the direct and fully associative approaches, it is used in the majority of cache applications.

The cache is controlled by the MMU [HWA93], which translates the virtual address to a physical address. Then the tag field of the address is checked and validated to determine a hit, which incurs address checking overhead.

When a cache miss occurs, a block must be replaced with the desired data. Four of the most common strategies employed for replacement are: (1) least recently used (LRU), (2) first-in-first-out (FIFO), (3) least frequently used (LFU), and (4) random.

To study cache performance, we consider the following parameters affecting cache speed and hit rate:

1) *Block size* (S_{BLK}): the number of bytes in a block. Larger block sizes will reduce compulsory misses but may increase conflict misses and even capacity misses. Larger blocks also increase the miss penalty.
2) *Cache size* (S_{CA}): the number of bytes or words in cache. A larger cache size can certainly reduce miss rate, but is more expensive.
3) *Associative degree* (D_{WAY}): the number of blocks in a set. Greater associativity can reduce miss rate at the cost of increased hit time.
4) *Miss rate* (R_m): the fraction of accesses that are not in the cache, a function also denoted by $r(S_{BLK}, S_{CA}, D_{WAY})$.
5) *Cycle time* (C_{CA}): the time in seconds needed for one cache access.
6) *Address checking overhead* (O_{AC}): in unit of seconds, the time needed to check a cache address for a hit. It is a function of associative degree D_{WAY} and cache size S_{CA}, denoted by $O_C(D_{WAY}, S_{CA})$. The hit time T_h for cache access is

$$T_h = C_{CA} + O_C(D_{WAY}, S_{CA}).$$

7) *Replacement overhead* (O_R): the time in seconds needed to replace a block if there is a miss, which is dependent on replacement strategy. When a replacement strategy is determined, replacement overhead is a function of block size S_{BLK}, denoted by $O_R(S_{BLK})$. The miss penalty time T_m is replacement overhead plus the time to access memory:

$$T_m = O_R(S_{BLK}) + C_{MEM} / [(1 - P_{CF}) \cdot (1 - P_{MF})].$$

Then, the effective memory access time T_{EFF} is given by:

$$T_{EFF} = T_h + R_m \cdot T_m$$

$$= C_{CA} + O_C(D_{WAY}, S_{CA}) +$$

$$r(S_{BLK}, S_{CA}, D_{WAY}) \cdot \{O_R(S_{BLK}) + C_{MEM} / [(1 - P_{CF}) \cdot (1 - P_{MF})]\}$$

8) *Word width* (W_{WD}): is the number of bits per memory word. A word is the basic unit of one cache access.

9) *Cache access failure rate* (P_{CAF}): denotes the probability of cache access failure.

Combining the preceding expressions, we obtain the effective bandwidth for memory model with associative cache as follows:

$$BW_{EFF} = (1 - P_{CAF}) \cdot W_{WD} / [T_{EFF} \cdot f(S_{TLB})]$$

$$= \frac{(1 - P_{CAF}) \cdot W_{WD}}{\{C_{CA} + O_C(D_{WAY}, S_{CA}) + r(S_{BLK}, S_{CA}, D_{WAY}) \cdot [O_R(S_{BLK}) + \frac{C_{MEM}}{(1 - P_{CF}) \cdot (1 - P_{MF})}]\} \cdot f(S_{TLB})}$$

3.4 CPU Model

3.4.1 Switch

The CPU contains the logic circuitry that executes the instructions in a computer program. Since instructions and data are stored in memory, switches are used to transfer information between the CPU and its memory. There are three principal types of switches: monostable, bistable, and crossbar.

A crossbar switch allows any CPU to communicate with any memory in one pass. In the crossbar shown in Figure 5, each processor has a (horizontal) wire linking it to all memories, or equivalently, each memory has a (vertical) wire linking it to all processors.

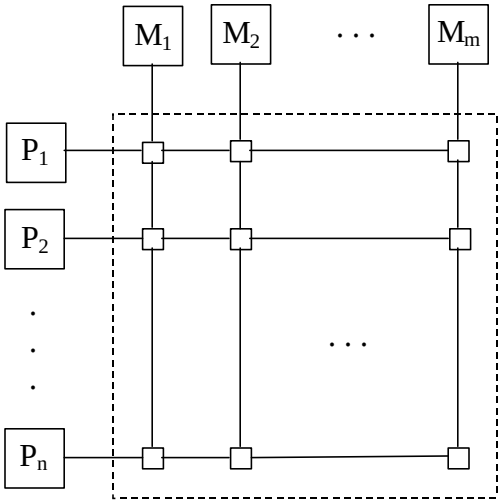

Figure 5: Crossbar switch

An $n \times m$ crossbar allows up to min{n, m} transactions to take place simultaneously, so it can relieve contention problems encountered in the von Neumann architecture discussed later in this chapter. To model the switch, we only need to consider the latency and the faults caused by intermittent connections or connections that are open-circuit or frozen to ground:

1) *Switch latency* (L_S) in unit of seconds/word, is the time needed to transfer one word through the switch. The *bandwidth of a switch* (BW_{SW}) would be:

$$BW_S = W_{WD} / L_S.$$

2) *Switch failure rate* (P_{SF}) denotes the probability of a switch failure. Switch failure affects the bandwidth as follows:

$$BW' = (1 - P_{SF}) \cdot \quad BW.$$

The time used to transfer a word through a switch is as follows:

$$T_S = L_S / (1 - P_{SF}).$$

3.4.2 ALU

Two major structural components of a CPU are

- The *arithmetic and logic unit (ALU)*, which performs arithmetic and logical operations, and
- The *control unit*, which extracts instructions from memory and decodes and executes them, calling on the ALU when necessary.

The ALU is that part of the computer that actually performs arithmetic and logical operations, such as addition, multiplication, and comparison, on operands in the computer instruction words [HAY98, PAT97]. Typically, the ALU has direct I/O access to the processor controller, main memory, and input/output devices, where connectivity is provided by buses. The input consists of an instruction word that contains an operation code, one or more operands, and an optional format code. The operation code tells the ALU what operation to perform. The operands are input to the operation. For example, two operands might be added together or compared. The output consists of a result of the

operation that is placed in a storage register, with settings that indicate whether the operation was performed successfully.

In general, the ALU includes storage elements for input operands, parameters, an accumulated partial result, and shifted results [STA96]. The arithmetic and logical operations performed on the operands are controlled by precisely clocked circuits. In this section, we view the ALU as a "black box", i.e., we ignore how these storage elements and gated circuits are implemented in digital logic.

To model an ALU as a black box, we consider the following parameters:

1) *ALU throughput* (BW_{ALU}) in unit of bits/second;
2) *Bits of precision* (PR_{ALU}) accuracy of computational precision in bits;
3) *Error rate* (P_{ALUE}) the probability of ALU error;
4) *Datapath width* (W_{DP}) the width of a word along the datapath, in bits; and
5) *Error burst duration* (T_{EB}) in seconds, the time an error burst lasts.

Thus, we get the ALU execution time:

$$T_{ALU} = (W_{DP} / BW_{ALU}) + P_{ALUE} \cdot \ T_{EB}.$$

Although the ALU is obviously a critical part of the CPU, other elements of the computer system, such as, switches, registers, memory, bus, and I/O processor, bring data into the ALU for it to process and output. A key example of this technique is the von Neumann architecture [STA96], which is built from the following components:

- A *main memory*, which stores both data and instructions;
- An *ALU* capable of operating on binary data;
- A *control unit*, which interprets the instructions in memory and causes them to be executed; and

- *Input and output (I/O)* equipment operated by the control unit.

The data and instruction flows in a von Neumann architecture are shown in Figure 6.

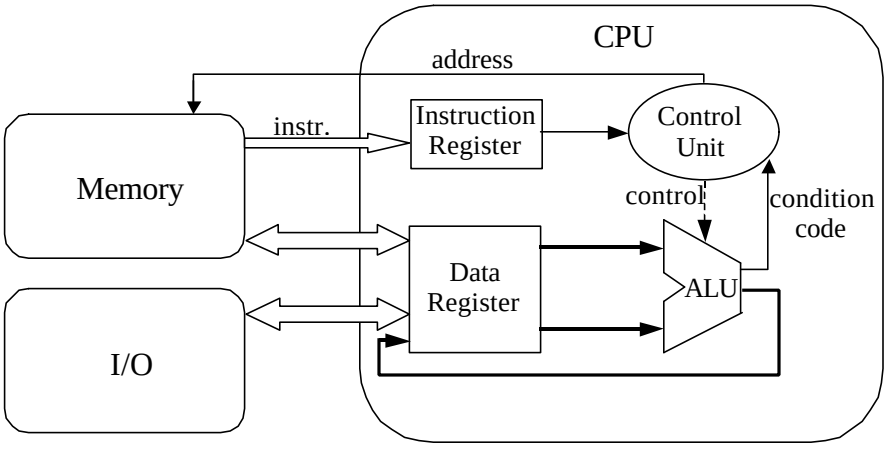

Figure 6: Data and instructions flows in a von Neumann architecture

In contrast, in a SIMD computer, an array of PEs is typically managed by one control unit [CYP90, SUN90, HWA93]. The instruction set is decoded by the control unit. The PEs are passive ALUs executing instructions broadcast from the control unit. Program and data are loaded into the control memory through the host computer. An instruction is sent to the control unit for decoding. Partitioned data sets are distributed to all the local memories attached to the PEs through a data bus. The PEs are interconnected by a communication network (NEWS grid) which performs inter-PE data communication such as shifting, permutation, and other routing operations. The communication network is also managed by the control unit. In other words, the same instruction stream is used to operate on separate data streams by the PEs. Figure 7 shows the data and instructions flows in SIMD computers.

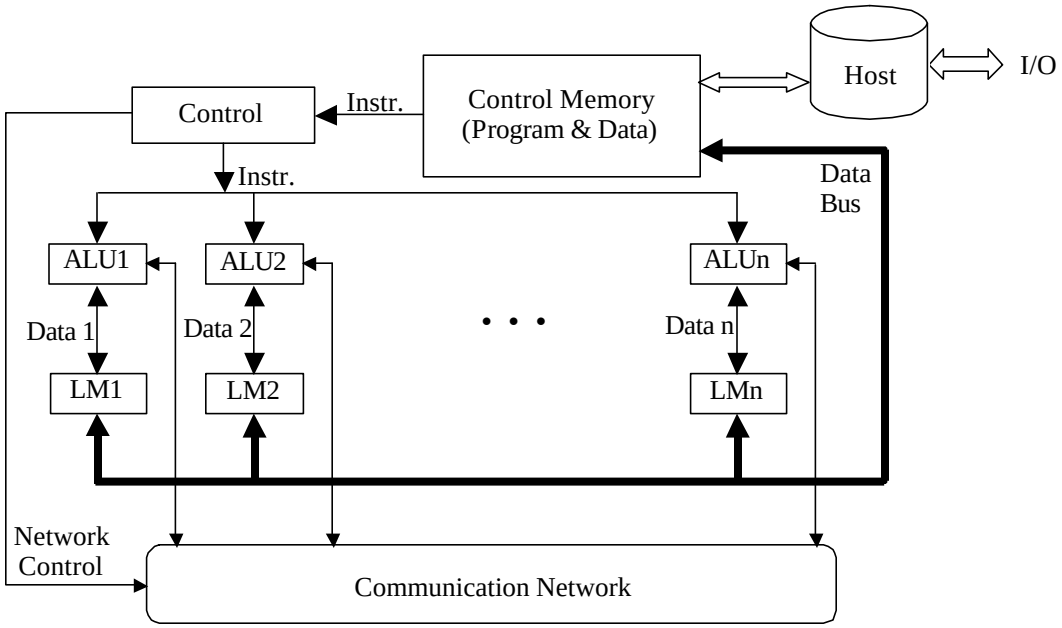

Figure 7: Data and instructions flows in SIMD

3.4.3 CPU Model

As the CPU comprises the ALU and the control unit, and I/O is connected to the CPU by switches, the CPU model is a composition of the ALU model, the control unit model (with delay T_{CU}), and switch model.

When the control unit is decoding the instructions, operands may be loaded into the ALU or results may be stored to memory through switch at the same time to gain speedup. With this parallelism, the CPU model is as follows:

$$T_{CPU} = \max(2T_S, T_{CU}) + T_{ALU}.$$

If there is no such parallelism, the CPU model would be:

$$T_{CPU} = 2T_S + T_{CU} + T_{ALU}.$$

CHAPTER 4
IMPLEMENTATION AND SIMULATION RESULTS

We have presented several models of computation in Chapter 3. In this Chapter, we discuss implementations of the models of computation (MOCs) that estimate system performance. Based on the architecture shown in Figure 1, we can see how, as the instructions (ASC $\rightarrow$ ASM $\rightarrow$ BVC) go through the three hardware layers, the time delay on each hardware object can be computed.

Before the simulation procedure begins, we need a translator to generate ASMs and BVCs from ASCs. Thus, two major components are required in our implementation:

I. *The translator*: scanning in AIM Server Calls (ASCs), translating them into assembly instructions (ASMs), and then translating ASMs into Bit Vector Calls (BVCs); and

II. *The simulator*: running ASCs, ASMs and BVCs through the three simulated hardware layers and gathering performance information.

4.1 The Translator

The translator has two stages, in which the first one is to translate ASCs into ASMs and the second one is to translate ASMs generated from the first stage into BVCs.

4.1.1 Translating ASCs into ASMs

In this stage, there are three phases: (1) lexical analysis, (2) syntactic analysis, and (3) semantic analysis and code generation. The program first scans the source code file containing a set of ASC instructions. It decomposes characters into meaningful tokens

such as keywords, numbers, and special symbols. In the meantime, it ignores the comment lines and in-line comments. This is the lexical analysis phase.

The next phase is syntax analysis. There are two kinds of statements in an ASC file. One is the `MOCSET` construct that sets variable values, and the other statement is an ASC call. Because the ASC grammar is simple, there is no need to construct a parse tree. The translator checks the tokens from the lexical analysis according to the grammar. If the token is not what it expects, then a syntax error occurs.

The semantic analysis and code generating phase is much more complicated than the first two phases. The flow chart of this phase is shown in Figure 8.

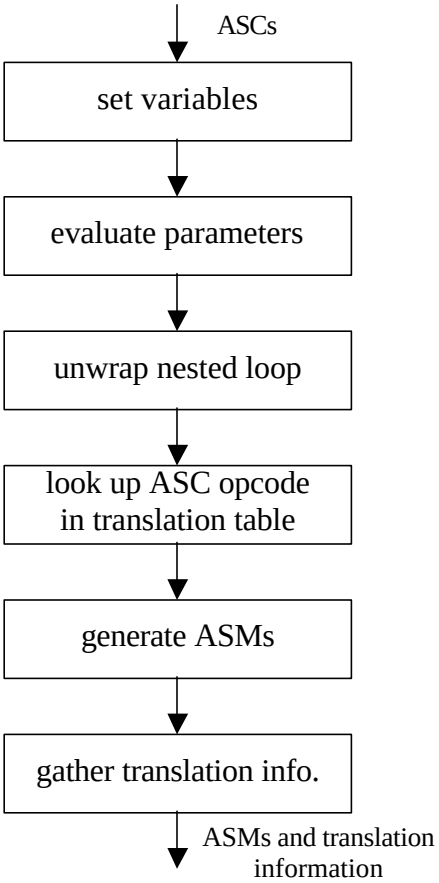

Figure 8: Flow chart of translating ASCs to ASMs

In order to support the MOCSET construct, which has the form:

MOCSET <variable-name> = <value>[, <variable-name> = <value>]*

a variable name-value table is established to maintain the binding between the variable names and their values. Variable names are case sensitive. So, when a variable name is detected in the following instructions, the translator looks in the table to obtain the value associated with the variable name.

ASC instructions are expressed in the following format [SCH00], in forms of a vector representation:

<opcode> (D_1, M_{11}, M_{12}, . . ., M_{1D_1}, S_1, N_1, T_{11}, T_{12}, . . ., T_{1D_1},
D_2, M_{21}, M_{22}, . . ., M_{2D_2}, S_2, N_2, T_{21}, T_{22}, . . ., T_{2D_2},
γ, P_γ, o, P_o)

whose parameters are defined in Table 1:

Table 1: List of ASC Parameters and their definitions.

Parameter	Definition
<opcode>	ASC operation code
D_1	Dimensionality of first operand
M_{11}, M_{12}, ..., M_{1D_1}	Size of each dimension 1..D_1 of domain of first operand
S_1, N_1	Number of signal and noise bits per pixel in first operand
T_{11}, T_{12}, ..., T_{1D_1}	Origin of first operand
D_2	Dimensionality of second operand
M_{21}, M_{22}, ..., M_{2D_2}	Size of each dimension 1..D_1 of domain of second operand
S_2, N_2	Number of signal and noise bits per pixel in second operand
T_{21}, T_{22}, ..., T_{2D_2}	Origin of second operand
γ, P_γ, o, P_o	Operators γ and o, with bits of precision

The majority of these parameters are currently represented by expressions in the ASC file. The expressions use the basic arithmetic operators: +, –, *, /, and specify the order of evaluation by using parentheses. If there are variables in the expression, then the

translator first examines the name-value table to retrieve the value associated with the variable. During the evaluation of an expression, a number stack and a symbol stack are used to implement proper operator priority and proper evaluation order.

The translator accepts nested loops in the ASC description. These are a composite representation of instructions having form:

{ ... { ... { ... }*m ... }*n ... }*k

where ... denotes a list of comma-delimited ASC instructions, and m, n, k are expressions that denote how many times the instruction loops repeat. Nested loops make the algorithms concise, as well as improve conciseness and readability. However, for purposes of translation, nested loops must be unwrapped using a stack. For example, each time the translator detects an open brace, it pushes the index of the current ASC call onto the stack. When the translator detects a closed brace and detects the repetition factor (e.g., m, n, k), it pops the stack to get the last { position. The ASCs ranging from the last { position to the current ASC are replicated per the given repetition factor.

In the next step, the translator looks up the ASC opcode in a translation table, which is specific to a given device. This file contains translation information for each ASC opcode, for example, (a) a series of ASM opcodes into which an ASC is translated, (b) the number of ASMs in this series, (c) the number of operands for each ASM instruction, (d) translation time, expressed in machine cycles, and (e) type of opcode. After finding the ASC opcode, the translator obtains a series of ASMs and associated translation information. This ASM sequence can also be represented in nested loops using the same general approach as ASC specification and translation.

The translator then generates ASM instructions. For each ASM, the translator first generates the i.d. tags for the specified registers. From parameters in each ASC, the translator computes image and template size, and determines the type of image algebra operations based on the outer (and inner) operators γ (and o). Then the translator computes the register sizes from image size, template size, signal bits, type of operation, and precision bits of the outer and inner image algebra operators γ and o according to the following rules:

1. When γ is null and o is non-null, a pointwise operation occurs, and the sizes for register 1, 2 and 3 are:

$$\text{reg-size-1} = \text{reg-size2} = \max(S_1 \text{ and } S_2) \cdot I, \text{ and}$$

$$\text{reg-size-3} = P_o \cdot I.$$

   where I denotes the image size. S_1, S_2 and P_o are defined in Table 1.

2. When γ is non-null and o is null, a global reduce operation occurs, and the size for register 1, 2 and 3 are:

$$\text{reg-size-1} = S_1 \cdot I,$$

$$\text{reg-size-1} = 0, \text{ and}$$

$$\text{reg-size-3} = P_\gamma \cdot I.$$

   where P_γ is defined in Table 1.

3. When γ and o are both non-null, an image-template or template-template operation occurs, and the sizes for registers 1, 2 and 3 are:

$$\text{reg-size-1} = \text{reg-size2} = \max(S_1 \text{ and } S_2) \cdot I, \text{ and}$$

$$\text{reg-size-3} = \max(P_\gamma, P_o) \cdot I.$$

In this way, a list of ASM instructions is built, each of which has the format:

```
<opcode> <reg-id-1> <reg-size-1> <reg-id-2> <reg-size-2>
         <reg-id-3> <reg-size-3>
```

where the variables signify:

`<opcode>`	ASM operation code,
`<reg-id-`*i*`>`	Register id for i-th operand, and
`<reg-size-`*i*`>`	Register size for i-th operand.

The final but the most important step is to gather translation information. The numbers of data bits and instruction bits are computed from dimensionalities of ASC operands, image size, template size and signal bits. The number of result bits is estimated from the image size and from the type of image algebra operations (based on operators γ and o), as shown in the derivation of register 3's size in the preceding rules. In addition, the translation information obtained in translating ASCs into ASMs can be used for performance evaluation. For convenience, these data are stored in tables that are prespecified as ASCII files.

4.1.2 Translating ASMs into BVCs

Because the ASMs are generated by the translator and are less complicated than ASCs, the lexical and syntax analysis is straightforward. For example, there are no variables to be set. Also, in generating ASMs, the translator has computed the register size and unwrapped the nested loops, so there is no need to evaluate parameters or unwrap. Therefore, this stage is similar to but simpler than the first one. The flow chart of this stage is shown in Figure 9.

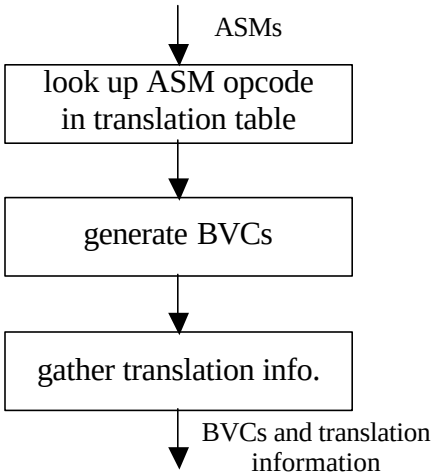

Figure 9: Flow chart of translating ASMs to BVCs

For each ASM call generated in the first stage, the translator also locates the ASM opcode in the translation table file for a given device and obtains a series of BVC opcodes into which an ASM is translated for that device. Associated translation information includes the number of BVCs in this sequence, the number of operands for each BVC instruction, translation time, and type of opcode.

To generate BVC instructions, the translator first generates the register i.d. tags for each BVC. Based on the register sizes of ASM, operators γ and o, as well as device information, the translator computes the register sizes for the BVCs. Then, a list of BVC instructions is built. The BVC instructions generated by the translator are in a format similar to ASMs.

In the next step, the translator computes the numbers of data bits, instruction bits, and result bits for each ASM. These data are computed from register sizes specified in the ASM and device tables described previously. In addition, the numbers of data bits, instruction bits and result bits for each BVC are also are computed based on the BVC

register sizes. Thus, a list of ASCs is translated into ASMs and BVCs, and key information about the translations is obtained.

4.2 The Simulator

The second part of implementation is to perform the simulation. Obtaining the results of the translator, the simulator runs ASCs, ASMs and BVCs through each hardware object and calculates the performance information based on the models of computation we developed in Chapter 3. The following subsections describe in detail the implementation components.

4.2.1 Bus Model Implementation

In AIM models of computation, there are various kinds of buses. In this subsection, we give typical implementations of the linear bus model and the packet bus model. Additional types of buses are similarly implemented.

A host bus provides the communication path between the host and the control unit. ASMs and data are transferred from the host to the control unit for execution, and the results are returned from the control unit to the host via the host bus. The host bus is modeled using the linear bus model. Given the choice of a simulated device for the host bus, we can get the parameters, nominal bandwidth (BW_{NOM}), bus failure rate (R_{BF}), security overhead (O_S), and bandwidth fluctuation (F_{BW}) for this device. These parameters are the inputs to the simulator. The simulator computes the effective bandwidth (BW_{EFF}) based on the linear bus model (in Section 3.2.1). Then, for every transmission on the host bus, the simulator gets the number of bits in this event from the results of the translator. If one ASM instruction and its associated data are transferred on the bus, then the number of transmission bits (b_{xhbu}) is the sum of the ASM instruction

bits and the data bits. If a result is transmitted along the bus, then the number of transmission bits (b_{xhbu}) equals the number of result bits. The simulator computes the transmission time (Δt_{xhbu}) on the host bus as follows:

$$\Delta t_{xhbu} = b_{xhbu} / BW_{EFF}.$$

The interconnections among multiprocessors can be modeled using a linear or packet bus. For example, when simulating a SIMD processor, we can implement the communication network (NEWS grid) among PEs as follows. In the beginning, the simulator gets the input parameters, number of partitions or packets (N_P), collision probability (P_C), and failure probability (P_F), for the NEWS grid. Then, for every transmission on the NEWS grid, the simulator gets the number of transmission bits from the results of the translator. If the data are transferred from the I/O buffer to the PE for execution, the number of transmission bits equals the number of the data bits (variable *dbtsbvc*) in a BVC instruction. If the result is returned from the PE to the I/O buffer, then the number of transmission bits is the number of result bits (*resbts*).

The simulator then computes the transmission overhead (O_T) and the error detection/correction overhead (O_D) for each packet, because these two parameters vary with the packet size. And the effective bandwidth (BW_{EFF}) is then computed based on the packet bus model (in Section 3.2.2). Finally, the simulator computes the transmission time (Δt_{ibcm} for transmission from the I/O buffer to the interconnection mesh, Δt_{cmib} for transmission from the mesh to the I/O buffer) in the NEWS grid as follows:

$$\Delta t_{ibcm} = \text{dbtsbvc} / BW_{EFF}$$

and

$$\Delta t_{cmib} = \text{resbts} / BW_{EFF}.$$

4.2.2 Memory / Buffer Model Implementation

There are diverse memories or buffers in a computer system. In AIM, the host has its own memory, there are instruction memory and frame buffer(s) in the control unit, I/O buffer in the chip, local memory in SIMD PE, and several buffers in FPGA Functional Block.

A buffer is simpler than memory, so simulation of a buffer is also straightforward. For each buffer, in the beginning the simulator inputs buffer width (W_{BUF}), cycle time (C_{BUF}), and buffer failure rate (R_{BF}), and then computes the effective bandwidth (BW_{EFF}) according to the buffer model (in Section 3.3.1). In the simulation, when there is I/O to the buffer, the simulator computes the delay time (Δt_{bf}) as:

$$\Delta t_{bf} = b / BW_{EFF}$$

where b denotes the number of I/O bits.

To implement the basic memory model, the simulator takes the following parameters as inputs: memory width (W_{MEM}), cycle time (C_{MEM}), memory size (S_{MEM}), Translation Lookaside Buffer (TLB) size (S_{TLB}), conflict probability (P_{CF}), and memory access failure rate (P_{MF}). Memory management unit (MMU) overhead (O_M) is computed from TLB size (S_{TLB}). Based upon the values of these parameters, the simulator computes the effective bandwidth (BW_{EFF}) according to the memory model (in Section 3.3.2). For every memory access, the simulator computes the memory access delay (Δt_{mem}) as:

$$\Delta t_{mem} = b / BW_{EFF}$$

where b denotes the number of data bits stored in (or retrieved from) memory with each memory access.

For the memory model with associative cache (in Section 3.2.3), the simulator needs more parameters such as the block size (S_{BLK}), cache size (S_{CA}), associative degree (D_{WAY}), miss rate (R_m), cycle time (C_{CA}), word width (W_{WD}), and cache access failure rate (P_{CAF}). The simulator computes the address checking overhead (O_{AC}) upon D_{WAY} and S_{CA}, replacement overhead (O_R) upon S_{BLK}, and then the effective bandwidth (BW_{EFF}) according to the model. For every cache/memory access, the simulator computes the access delay (Δt_{ca}) as

$$\Delta t_{ca} = b / BW_{EFF}$$

where b denotes the number of bits in the cache/memory access.

4.2.3 CPU Model Implementation

The CPU model is a composition of the ALU model, control unit model, and switch model.

Initially, the simulator gathers the inputs for these three models. The switch model parameters are switch latency (L_S) and switch failure rate (P_{SF}). The simulator computes the time used to transfer a word through the switch (T_S) based on these parameters according to a switch model. The ALU model parameters are ALU throughput (BW_{ALU}), bits precision (PR_{ALU}), error rate (P_{ALUE}), datapath width (W_{DP}), and Error burst duration (T_{EB}). The simulator calculates the ALU execution time T_{ALU} according to the ALU model. Also, the simulator takes the control unit delay T_{CU} as input.

During the simulation of an instruction being executed in the CPU, the simulator obtains the number of words in the operands (NW_{OP}) and the number of words in the result (NW_{RST}). Then it computes the switch time delay (Δt_{sop}) for loading operands into ALU as

$$\Delta t_{sop} = NW_{OP} \cdot \ T_S,$$

the ALU execution time (Δt_{alu}):

$$\Delta t_{alu} = NW_{OP} \cdot \ (W_{WD} / W_{DP}) \cdot \ T_{ALU},$$

and the switch time delay (Δt_{srst}) for storing the result into memory:

$$\Delta t_{srst} = NW_{RST} \cdot \ T_S.$$

Finally, the simulator computes the time delay for CPU execution (Δt_{cpu}) according to whether there is parallelism between the control unit and I/O to the CPU or not, as

$$\Delta t_{cpu} = max(\Delta t_{sop} + \Delta t_{srst}, T_{CU}) + \Delta t_{alu}$$

with parallelism, or

$$\Delta t_{cpu} = \Delta t_{sop} + T_{CU} + \Delta t_{alu} + \Delta t_{srst}$$

without parallelism.

4.2.4 Composition of Models

The simulator is a composition of the implementations of the selected sub-models. In our MOCs, the simulation begins at the host, which accepts a sequence of ASCs. For each ASC, the simulator obtains a sequence of ASMs into which an ASC is translated, the ASC $\rightarrow$ ASM translation time delay, the number of data bits and the number of instruction bits from the translator. Then, it simulates the transmission of data and instructions to the unit through the host bus, and uses the bus model to compute the transmission delay.

In the unit, for each ASM, the simulator obtains a sequence of BVCs into which an ASM is translated, the ASM $\rightarrow$ BVC translation time delay, the number of data bits and the number of instruction bits from the results of the translator. The BVC

instructions are stored in the instruction memory, and the data are stored in a frame buffer. The access times for the memory and the buffer can be computed according the memory/buffer model.

The instructions and data are transferred to every PE through a bus or communication network. The transmission delay is obtained using a linear or packet bus model. In each PE, the execution time is computed using the CPU model, and the access time to its own local memory is computed using a memory model.

The results are returned from the PE to the unit through the communication network and then from the unit to the host through the host bus. These transmission delays are computed from the appropriate bus models.

4.3 Example Simulation Outputs

To illustrate the operation of our MOC simulation process, we simulate a SIMD processor, which is a PC host system with a PCI-bus connecting SIMD Unit(s), using the EBLAST compression algorithm [SCH99] in ASCs. The system is configured as follows: the PC host is a 500MHz Pentium, the Host $\leftrightarrow$ Unit PCI bus is 32bits parallel at 33MHz, the IOC card controller is also a 500MHz Pentium, and the SIMD mesh size is 8 $\times$ 8 PEs, with each PE running at 200MHz and mesh data transfer rate at 1Gbyte/sec. These configuration parameters are input to the MOC simulator. Figures 10 and 11 exemplify time graphs from the simulation results.

Figure 10 illustrates the accumulated times for different functional modules at the Host level. Line 1 in the Figure shows the accumulated execution time (Δt_{simd}) for the SIMD processor, including computational cost and I/O cost, line 2 shows the accumulated execution time (Δt_{unit}) for the unit, and line 3 shows the accumulated

transmission time (Δt_{xubh}) for the host bus. Δt_{simd} is the summation of Δt_{xasc}, Δt_{unit}, and Δt_{xubh}. Note that Δt_{xasc}, which is the ASC $\rightarrow$ ASM translation time, is too small to be shown in this Figure. In Figure 10, if we add line 2 and line 3 together, we obtain line 1.

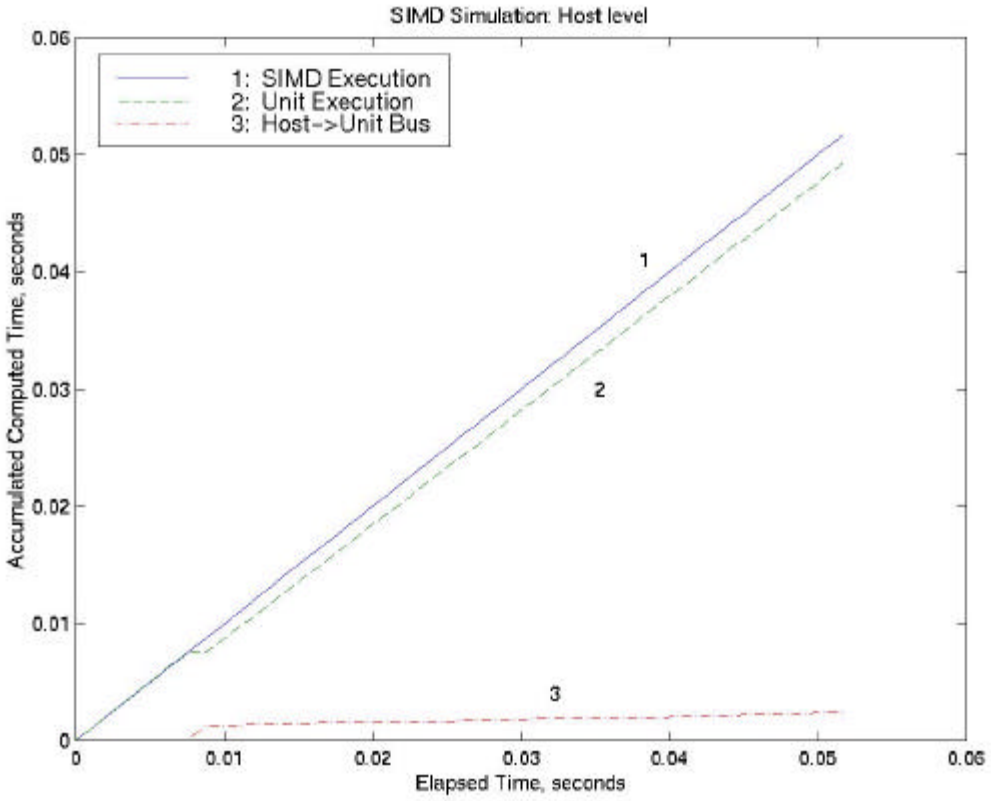

Figure 10: Time graph at Host level

Figure 11 illustrates the accumulated times for different functional modules at the IPC Card level. Line 1 in the Figure shows the accumulated I/O time (Δt_{ibcm}) for the transmission from the I/O Buffer to the communication network (NEWS grid). Line 2 shows the accumulated I/O time (Δt_{cmau}) for the transmission from the NEWS grid to the ALU. Line 3 shows the accumulated I/O time (Δt_{lmio}) for the transmission from each PE's local memory (LM) to the ALU. Line 4 shows the accumulated computation time

(Δt_{aluf}) for the ALU, which is net computational cost. This is not like the total SIMD execution time Δt_{simd} (Line 1 in Figure 10), which includes computational cost and I/O cost, and is thus small.

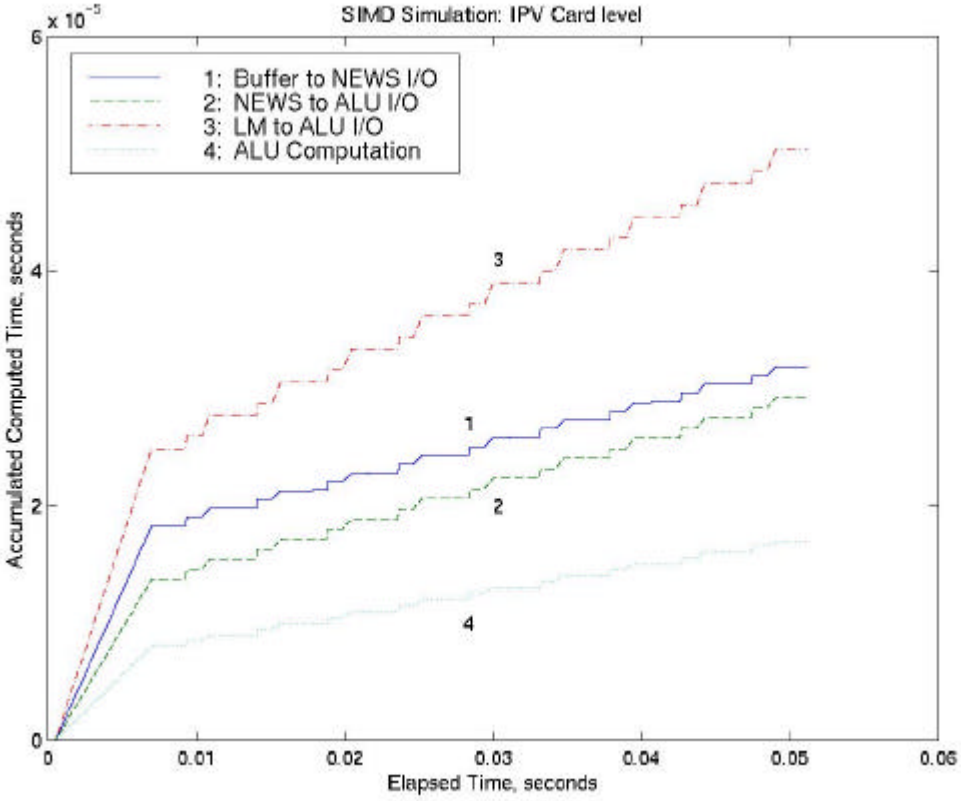

Figure 11: Time graph at IPV Card level

At the IPV level, data and instructions are brought from the IOC and local memory through different paths to the ALU, where the main computation takes place. Thus, the structure at this level is more complex than at the Host level. Additionally, there are more I/O channels in the IPV card, as we can see from Figure 10 and 11.

CHAPTER 5
SIMULATION RESULTS AND ANALYSIS

In this chapter, we present the time graphs and analysis for all simulation variables for the EBLAST compression algorithm.

5.1 I/O Costs

I/O is usually the bottleneck of a system. Thus, I/O bandwidth is a key performance measure.

Instruction and data transmission comprises I/O, which are sometimes assigned to different channels. We first discuss the difference between instruction and data I/O for each level.

Figure 12 exemplifies the instruction and data I/O for the transmission from the Host to the Unit. Line 1 in the Figure is the accumulated instruction I/O time on the Host $\rightarrow$ Unit PCI bus. Line 2 is the accumulated data I/O time on that bus. As we expected, the data I/O cost is more than the instruction I/O cost. In our simulation example, the image size is 10×10-pixel. Thus, the difference between data and instruction I/O costs is not great. When image size is large, the data I/O cost will be much more than the instruction I/O cost.

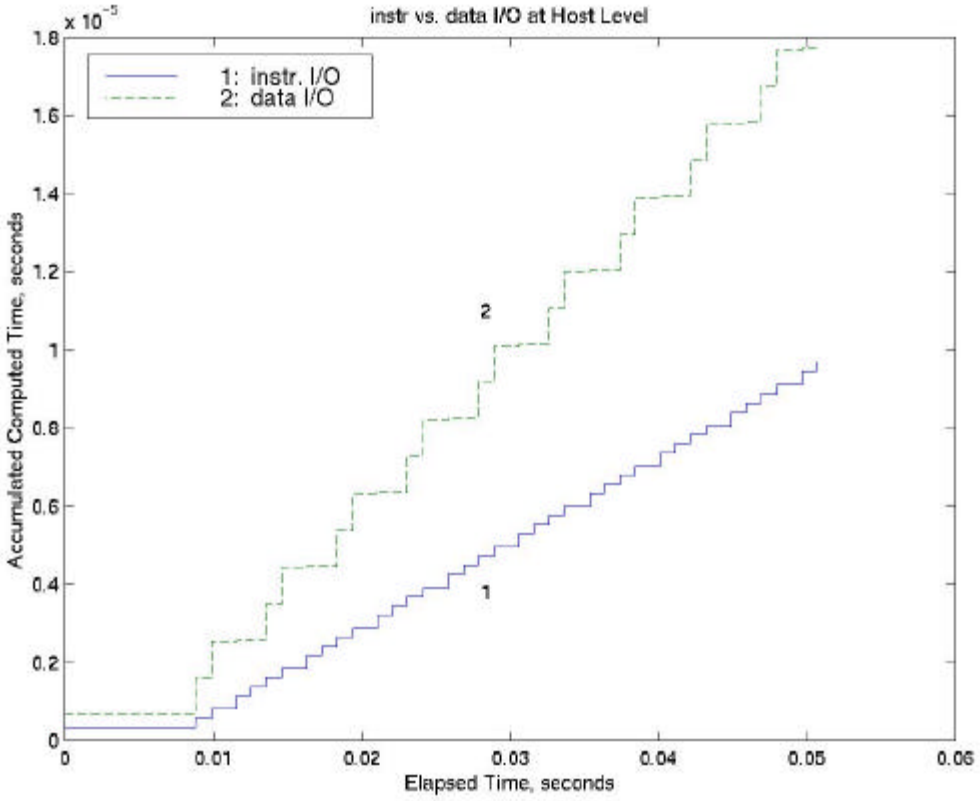

Figure 12: Instruction vs. data I/O at Host level

Within the IOC card, data are stored in Frame Buffer and instructions are stored in Instruction Memory. Figure 13 shows the difference between instruction and data I/O in the IOC card. Line 1 in the Figure is the accumulated instruction I/O time (Δt_{isio}) to the Instruction Memory. Line 2 is the accumulated data I/O time (Δt_{hfio}) to the Frame Buffer. It's easy to see that the data I/O cost is much more than the instruction I/O cost. When the Frame Buffer and Instruction Memory can be accessed in parallel, the instruction I/O cost is reduced.

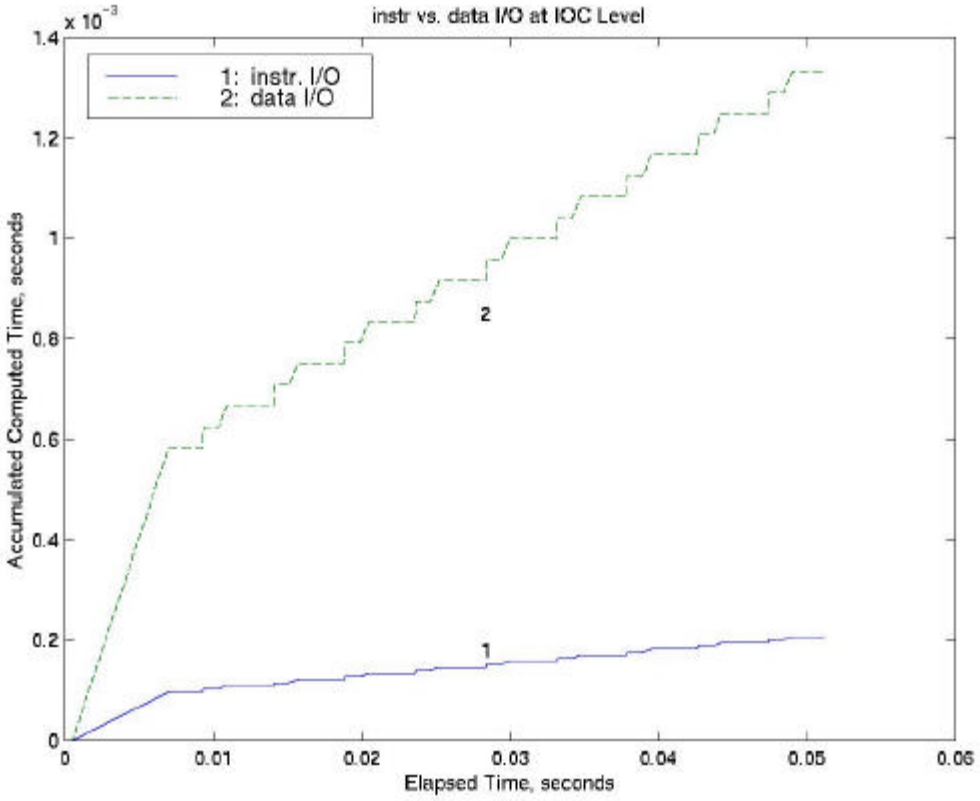

Figure 13: Instruction vs. data I/O within IOC card

Instructions stored in the Instruction Memory of the IOC card are transferred directly to the ALU in the PE. Data stored in the Frame Buffer are transferred to the I/O Buffer in the IPV card. Figure 14 shows the difference between instruction and data I/O from IOC to IPV. Line 1 in the Figure denotes the accumulated instruction I/O time (Δt_{imau}), and Line 2 denotes the accumulated data I/O time (Δt_{fbib}). When instructions and data can be transferred in parallel, the instruction I/O cost is also saved.

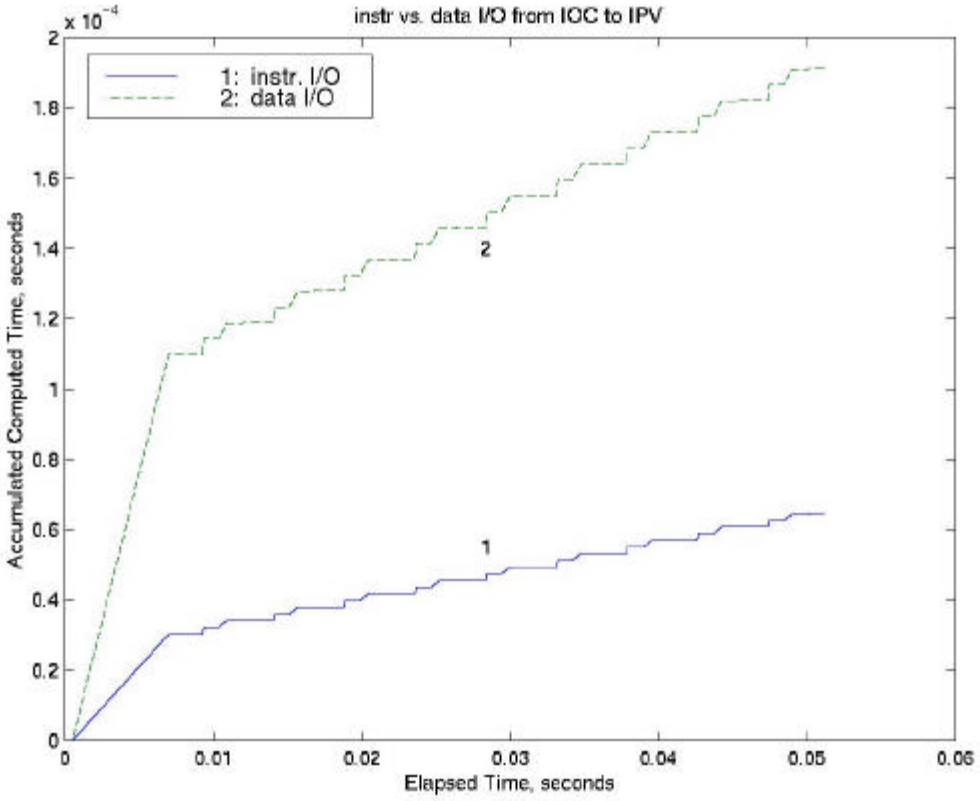

Figure 14: Instruction vs. data I/O from IOC to IPV

The I/O overheads within the IPC card are shown in Figure 11 (in Section 4.3). Data are input to the ALU through the communication network (NEWS grid) with transmission time $\Delta t_{ibcm} + \Delta t_{cmau}$. The register contents in the PE's local memory are transferred to the ALU with transmission time Δt_{lmio}. Instructions are transferred directly to the ALU from the Instruction Memory of the IOC card with transmission time Δt_{imau}. When the I/O from the I/O Buffer via the communication network to the ALU can be performed in parallel with the I/O from local memory to the ALU, considerable I/O cost savings result.

From the above figures, we can see that it takes much more time to transfer data than instructions because the size of the data is much larger than the size of the instructions.

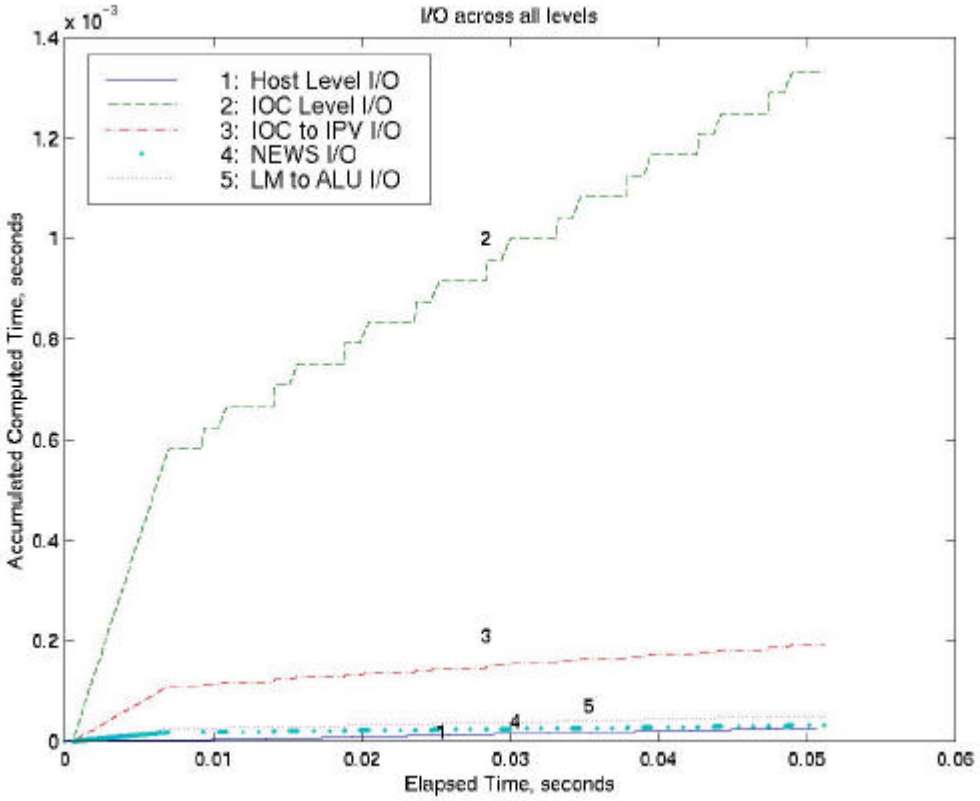

Figure 15: I/O across all levels

Figure 15 shows the difference between total I/O at each level. Line 1 in the Figure is the accumulated I/O time (Δt_{xhbu}) at the Host level for the transmission from the Host to the Unit, which is the summation of Δt_{xhbui} and Δt_{xhbud} (in Figure 12), because instructions and data are transferred together in the PCI bus. Line 2 is the accumulated I/O time (Δt_{hfio}) for the Frame Buffer at the IOC level. From this level downward in the SIMD MOC, instruction I/O and data I/O are separated. Data I/O cost is the major I/O

cost within IOC and instruction I/O cost may be ignored with parallelism, so only the data I/O cost is selected to represent the I/O cost at this level. Line 3 is the accumulated I/O time (Δt_{fbib}) from the IOC card to the IPV card. Line 4 is the accumulated I/O time through the communication network (NEWS grid) in the IPV card. Line 5 shows the accumulated I/O time (Δt_{lmio}) for the transmission from each PE's local memory (LM) to the ALU. As it can be seen, the most costly I/O is in the IOC, and the second most costly I/O is from the IOC to the IPV. This is due to the storage and retrieval of intermediate data during the computation. If register-register transfer is employed, this figure can be significantly reduced. The I/O within the IPV is small comparing with the I/O in the IOC, because the data size to each PE is small.

5.2 Computational Cost vs. I/O Cost

Computation is the most important task of a computer system, which requires analysis of computational cost. We compare computational cost with I/O cost to visualize system performance.

Figure 16 shows the accumulated computational time vs. the I/O time at the Host level. At the Host, the computation is to translate ASCs to ASMs, and the I/O is via the Host bus between the Host and the Unit. The lines show that the I/O costs nearly twice as much as the computation.

Figure 17 shows the accumulated computational time vs. the I/O time at the IOC level. In the IOC card, the computation translates ASMs to BVCs, and the I/Os are the data I/O to the Frame Buffer and the instruction I/O to the Instruction Memory. As stated earlier, the I/O cost predominates in the IOC card. The lines show that at this level the I/O costs over ten times more than the computation, as expected.

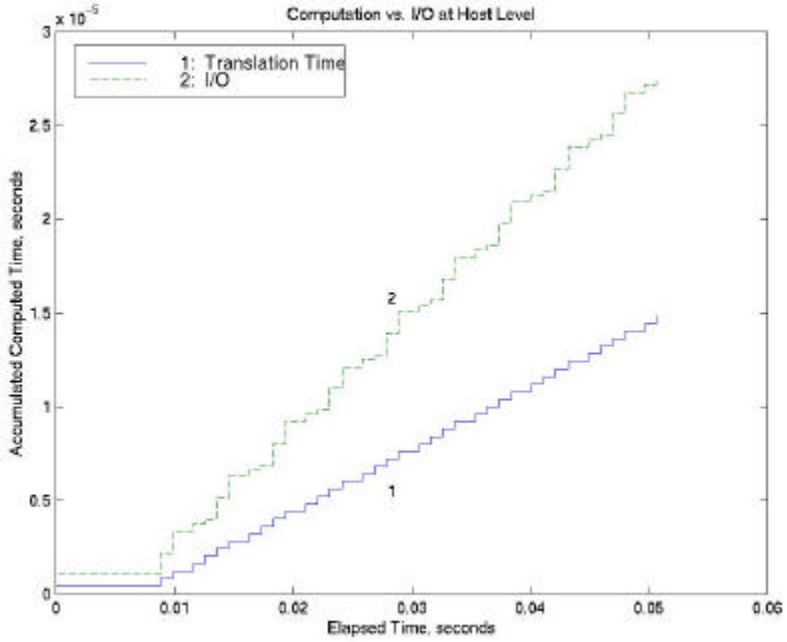

Figure 16: Computational cost vs. I/O cost at Host Level

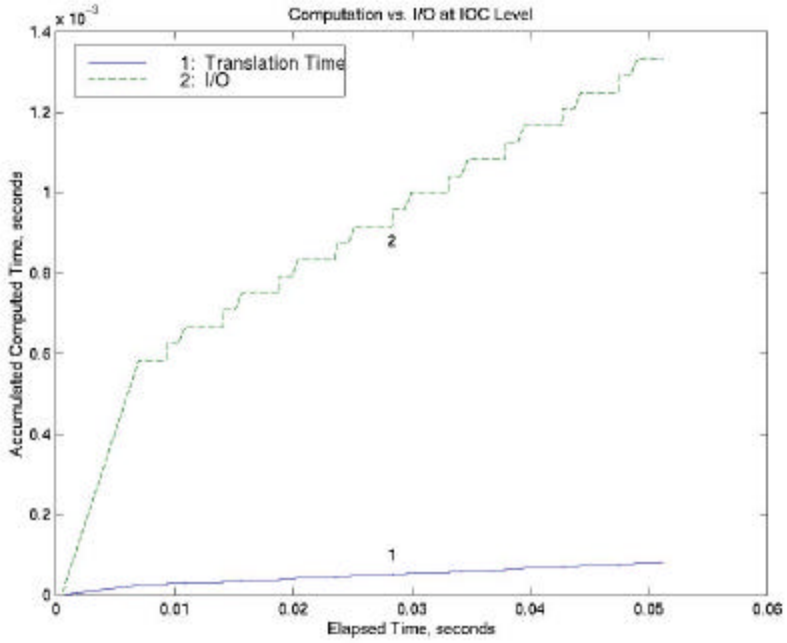

Figure 17: Computational cost vs. I/O cost at IOC Level

Figure 18 shows the accumulated computational time vs. the I/O time at IPV level. Within the IPV card, the computation is the ALU computation, and the I/Os are in the communication network and between the ALU and the local memory. The lines show that at this level the I/O also costs much more than the computation.

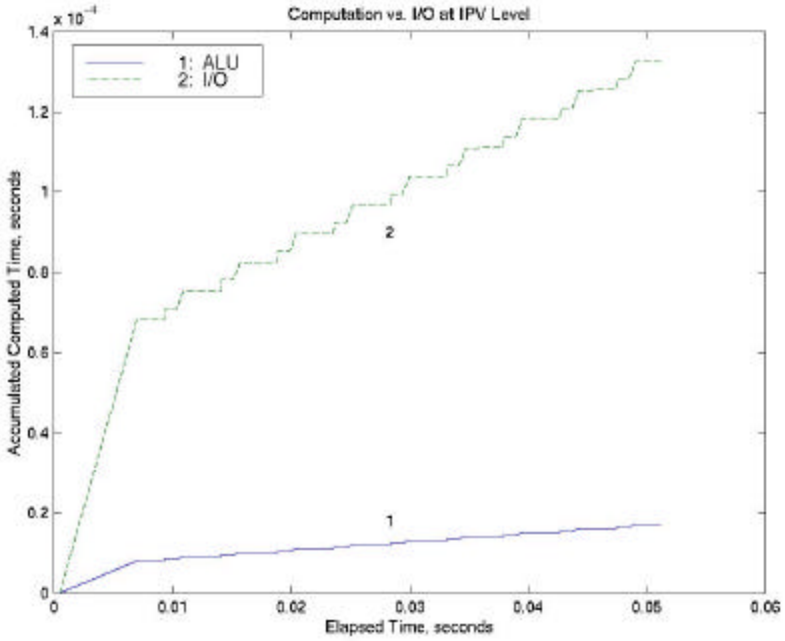

Figure 18: Computational cost vs. I/O cost at IPV Level

Figure 19 shows the ratios of the I/O time vs. the computational time at each level of the system and for all levels. Line 1 is the ratio of the I/O time vs. the translation time at the Host level. Line 2 shows the ration at the IOC level. Line 3 shows the ratio of the I/O time vs. the ALU time at the IPV level. And Line 4 is the ratio of the total I/O time vs. the total computational time for all levels. It is oscillatory, because the ratio is large when the system performs I/O operations, and the ratio is small when the system performs computational operations.

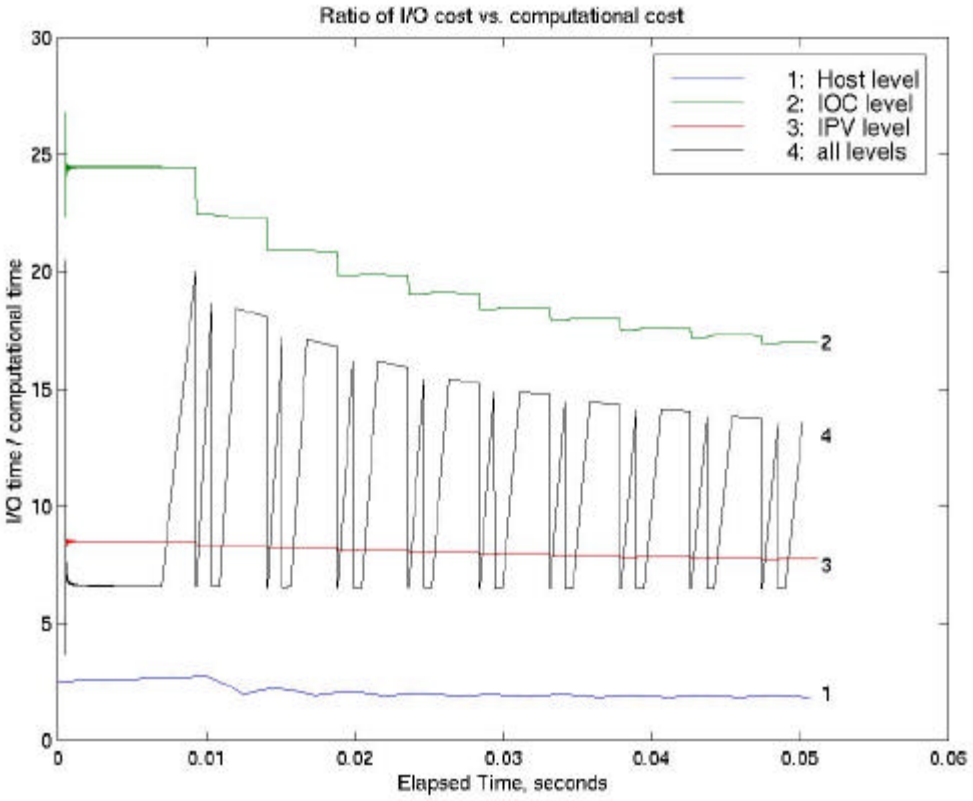

Figure 19: Ratios of I/O cost vs. computational cost

5.3 Effects of Mesh Size

To show the effects of mesh size on system performance, we perform simulation with a pointwise addition image algebra operation (see the ASC instruction in Appendix A).

Figure 20 shows the effect of mesh size on the ALU computational time. Line 1, 2 and 3 are the accumulated ALU computational times on mesh size 64×64 with image size 64×64, 80×80, and 100×100 correspondingly. Line 4 is the accumulated ALU computational time on mesh size 100×100 with image size 100×100. As we can see from Line 1, the first two ALU working periods are occupied with loading the two

operands, the third period is occupied by the addition operation, and the fourth period involves storage of the result. When mesh size and image size both equal 64×64, there is only one ALU working period for computation since all the pixels can go to the mesh for computation together. When image size becomes larger than mesh size, all the pixels cannot go to the mesh at a time, so it needs more ALU computation periods as shown in Line 2 and 3. However, when the mesh size increases to 100×100, image size equals mesh size again, the computation time decreases to one ALU period as shown in Line 4. In conclusion, the computation time for pointwise operations increases when the image size increases, but decreases when the mesh size increases.

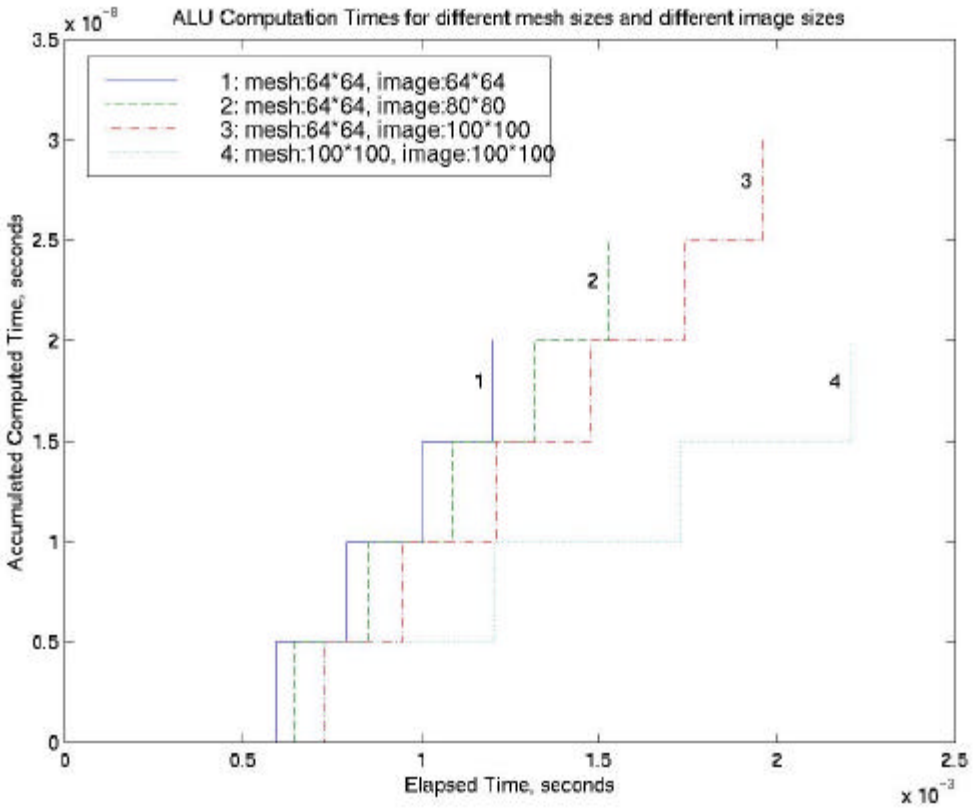

Figure 20: ALU computational times for different mesh sizes and different image sizes

Figure 21 shows the effect of mesh size on the ASM translation time. Similarly to Figure 20, Line 1, 2 and 3 are the accumulated ASM translation times on mesh size 64 × 64 with image size 64 × 64, 80 × 80, and 100 × 100 correspondingly. Line 4 is the accumulated ASM translation time on mesh size 100 × 100 with image size 100 × 100. The conclusion is also similar to the previous case of pointwise addition: if the mesh size is fixed, the translation time increases when the image size increases; and if the mesh size increases with all other parameters held constant, then the translation time decreases.

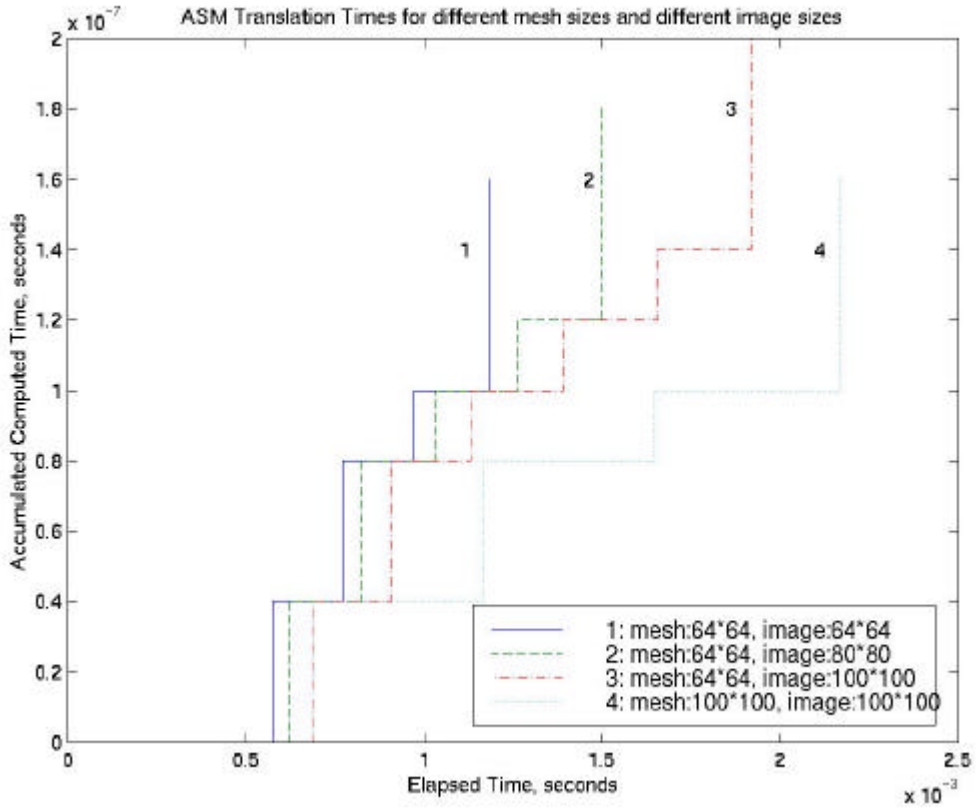

Figure 21: ASM translation times for different mesh sizes and different image sizes

In Figure 22, Lines 1 and 2 are the accumulated data I/O and instruction I/O for image size 64 × 64 on mesh size 64 × 64. Line 3 and 4 are the accumulated data I/O and

instruction I/O for image size 100 × 100 on mesh size 64 × 64. It is easy to see that I/O operations for larger image size cost more than for smaller image size.

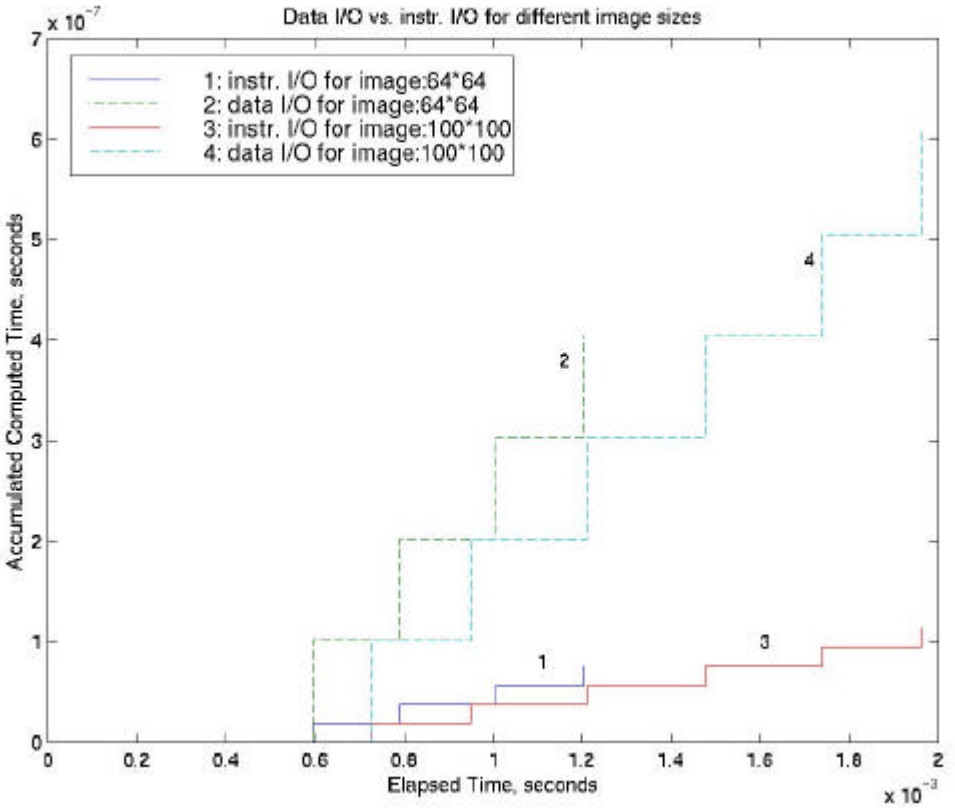

Figure 22: Data I/O vs. instruction I/O for different image sizes

In Figure 23, Line 1 and 2 are the accumulated data I/O and instruction I/O on mesh size 64 × 64 for image size 80 × 80. Line 3 and 4 are the accumulated data I/O and instruction I/O on mesh size 64 × 64 for image size 100 × 100. In this case, I/Os for larger mesh size cost less than for smaller mesh size.

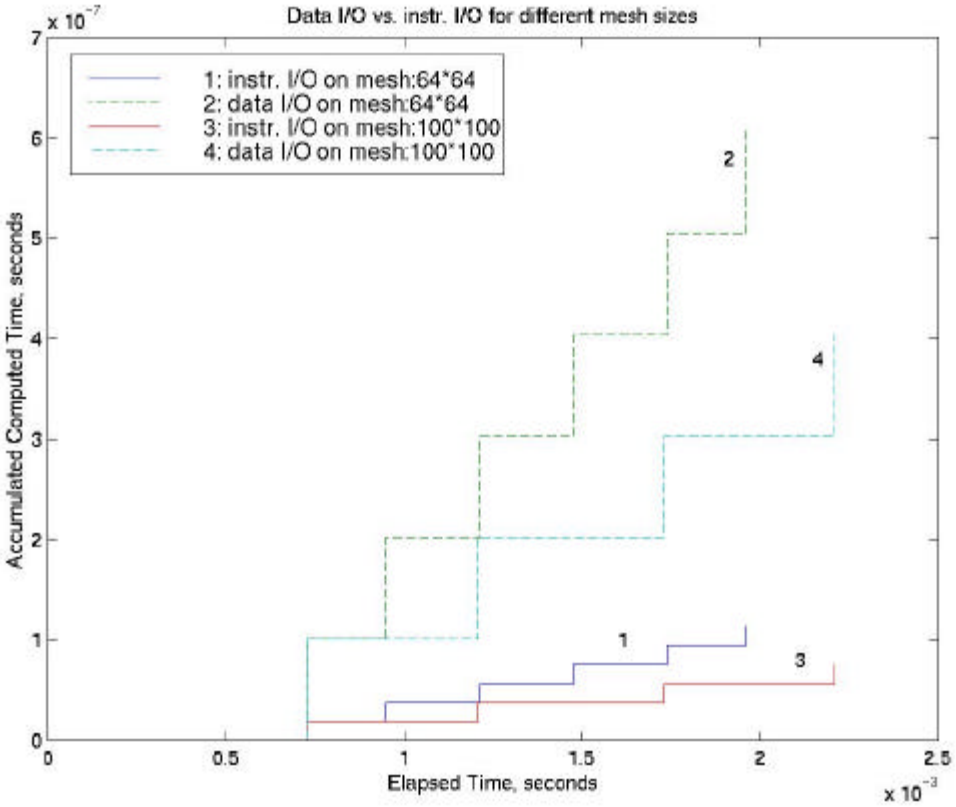

Figure 23: Data I/O vs. instruction I/O for different mesh sizes

CHAPTER 6
CONCLUSIONS AND FUTURE WORK

In this chapter, we present a summary in Section 6.1, as well as suggestions for future work in Section 6.2.

6.1 Summary

This thesis presents a methodology for estimating the performance and power consumption of parallel systems, e.g., UF's AIM project, which maps image and signal processing algorithms across parallel and distributed processing hardware architecture. To obtain high performance is a primary goal of parallel systems.

After investigating the requirements and system architecture of the AIM project and reviewing several approaches used in related research projects, we decided to adopt a hierarchical modelling with a componentwise approach to build performance estimation model. We developed several models of computation for the underlying hardware modules:

- *Bus Model*, which has three alternatives: linear bus model, packet bus model, and nonlinear bus model;
- *Memory Model*, which has three alternatives: buffer model, basic memory model, and memory model with associative cache; and
- *CPU Model*, which is a composition of switch model and ALU model.

These models are built based on a thorough study of the characteristics of various hardware objects. A mathematical equation is derived for each model. The system model is an ensemble of these modules.

The image processing algorithms are specified using the Image Algebra expressed in AIM Server Calls. Before simulation, the algorithms expressed in ASCs needed to be translated into ASMs and then further translated into BVCs. The implementation of the system model is realized by performing simulation through the three-level hardware hierarchy: Host, Unit, and PEs, with the corresponding ASC, ASM, and BVC instruction hierarchy. Each hardware object is simulated according to the model of computation for this hardware.

We performed the simulation of a SIMD processor with the EBLAST compression algorithm. The main performance measures that we analyzed are computational cost and I/O cost, which are represented in the Accumulated Computed Time vs. Elapsed Time graphs. The results revealed the differences between I/O at each level, data vs. instruction I/O at each level and across all levels, as well as computational cost vs. I/O cost at all levels.

6.2 Future Work

The models of computation for performance estimation we developed in this research can be used in performance debugging of parallel applications and in making architectural improvements for system hardware. However, extensions and improvements to this research remain as follows:

First, we plan to extend the range of the hardware model. Though we have developed models of computation for the hardware objects currently employed in the

AIM project, there is still a wide range of hardware that is not included but may be used in the future. We need to build the models of computation for them to support further design and implementation.

Second, in the implementation, the translator accepts nested loops in the ASC description and in the ASM sequence from the translation table, so an algorithm is translated into a large number of ASM calls. When the mesh size is large, the iteration times for the nested loops are very large, millions of ASM calls and more BVCs will be generated. And when running the simulator, every performance variable will be computed a value for each instruction. So, massive memory is needed to store them, but the run time memory space is limited. To make them useful for a large mesh size, we suggest developing a database for the instructions and performance variables.

Third, improving the accuracy of the model is another major part of the future work. Accuracy is the key concern of performance estimation. The accuracy of this system is dependent upon extensive knowledge about the system architecture and the physical characteristics of every hardware object. The models we developed in this study may need to be refined to better represent computational hardware. With higher accuracy, the performance estimates will be more useful for algorithm and system designers.

APPENDIX A
ASC GRAMMAR

The grammar for an AIM Server Call (ASC) in Extended BNF format is listed as follows:

<program> :: [<MOCSET constr>]* [<ASC instr>]$^{+}$

<MOCSET constr> :: MOCSET <variable-name> = <value>[, <variable-name> = <value>]*

<ASC instr> :: <opcode> (D_1, M_{11}, M_{12}, . . ., M_{1D1}, S_1, N_1, T_{11}, T_{12}, . .., T_{1D1}, D_2, M_{21}, M_{22}, . . ., M_{2D2}, S_2, N_2, T_{21}, T_{22}, . . ., T_{2D2}, γ, P_γ, o, P_o)

where * indicates zero or one or more repetitions, and $^{+}$ indicates one or more repetitions.

For example, the following ASC instruction:

```
ITGCNX (2,M,N,P,Q,X0,Y0, 2,Kt,Lt,R,S,Xt0,Yt0, +,16,x,16)
```

has a 2-D operand of M × N pixels with P bits of noise and Q bits of signal and origin at (X0,Y0), a 2-D operand of Kt × Lt pixels with R bits of noise and S bits of signal and origin at (Xt0,Yt0), outer operation of 16-bit addition, and inner operation of 16-bit multiplication. This implies 16-bit linear convolution of an M × N-pixel image with a Kt × Lt-pixel template.

And the following ASC instruction:

```
IADDX (2,K,L,P,Q,X0,Y0, 2,K,L,P,Q,X0,Y0, -1,-1,+,16)
```

is a pointwise addition of two K × L pixels images. Each image has P bits of noise and Q bits of signal, with origin at (X0,Y0).

APPENDIX B
EBLAST

EBLAST (Enhanced Blurring, Local Averaging, and Thresholding) [SCH99] is a high-compression image transformation.

B.1 Compression and Decompression Templates

EBLAST performance is dependent upon two template convolution operations. A compression template **s** preconditions the source image **a** to be more compressible, and to better withstand distortions inherent in EBLAST's block averaging process. The decompression template postconditions the compressed image to be more visually attractive or preserves statistics of **a**, by approximately inverting the effect of **s** on **a**.

B.1.1 Assumption

Let finite source domain $\mathbf{X} \subset \mathbf{R}^2$ and compressed domain $\mathbf{Y} \subset \mathbf{X}$, with source image $\mathbf{a} \in \mathbf{R}^{\mathbf{X}}$ and compression template $\mathbf{s} \in (\mathbf{R}^{\mathbf{X}})^{\mathbf{X}}$. Let decompression template $\mathbf{t} \in (\mathbf{R}^{\mathbf{X}})^{\mathbf{X}}$ be customarily space-variant. Let $h : \mathbf{X} \rightarrow \mathbf{Y}$ assign a source point **x** to an encoding block index **y**, and let the dual operation $h^* : \mathbf{Y} \rightarrow 2^{\mathbf{X}}$ return the domain in **X** of the **y**-th encoding block.

B.1.2 Compression Algorithm

Given Assumption B.1.1, EBLAST compression proceeds as follows:

Step 1. Subdivide **X** into encoding blocks $\mathbf{b}_\mathbf{y}$, $\mathbf{y} \in \mathbf{Y}$, where $domain(\mathbf{b}_\mathbf{y}) = h^*(\mathbf{y})$.

Step 2. Clamp the values in **a** to the interval $[g_b, g_s]$, which is between the perceived black and saturation levels of an image display.

Step 3. Convolve the compression template **s** with **a** to precondition the source image, then average each preconditioned source block, thereby yielding an image **c** on **Y** of block means.

Step 4. Quantize the means in c to yield an m-bit compressed image $\mathbf{a_c}$ on **Y**.

B.1.3 Decompression Algorithm

Given Algorithm B.1.2, EBLAST decompression involves the following steps:

Step 1. Dequantize the pixel values of compressed image $\mathbf{a_c}$.

Step 2. Project each of the |**Y**| pixel values produced in Step 1 to the corresponding source block domain. For example, the **y**-th pixel value is projected onto an intermediate image $\mathbf{d}|_{D(\mathbf{y})}$.

Step 3. Apply the decompression template **t** to **d**.

Step 4. [Optional] Inject noise or pseudorandom variance into the result of Step 3, to disguise blocking effect and simulate natural image variance or noise effects.

B.1.4 Image Algebra Formulation

Given Assumption B.1.1, let **s** and **t** be space-invariant templates, and let block averaging function $f_{\mathbf{L}} : \mathbf{R}^{\mathbf{KL}} \rightarrow \mathbf{R}$ such that the **y**-th encoding block $\mathbf{b_y}$ is averaged and quantized as

$$f_{\mathbf{L}}(\mathbf{b_y}) = q(\sum \mathbf{b_y}) ,$$

where q is a quantization function. Similarly, let block projection function $f_{\mathbf{B}} : \mathbf{R} \rightarrow \mathbf{R}^{\mathbf{KL}}$ dequantize a value $\mathbf{a_c}(\mathbf{y})$ in the compressed image and project it to the domain D(**y**), as follows:

$$\mathbf{d}|_{D(\mathbf{y})} = f_{\mathbf{B}}(\mathbf{a_c}(\mathbf{y})) = q^*[\mathbf{a_c}(\mathbf{y})] .$$

The EBLAST compression transform is described in image algebra as

$$\mathbf{a_c} = f_{\mathbf{L}}(\mathbf{a} \oplus \mathbf{s}),$$

and the decompression transform is given by

$$\mathbf{d} = f_{\mathbf{B}}(\mathbf{a_c}) \oplus \mathbf{t},$$

which does not include Step 4 of Algorithm B.1.3.

B.2 EBLAST Compression Algorithm in ASC

The EBLAST compression algorithm expressed in ASC is listed as follows:

```
## AIM Project Demonstration algorithm -- EBLAST compression
## AIM Server Call (ASC) format

## Set mesh dimensions corresponding to image partition
MOCSET M=8, N=8

## Set image origin and block dimensions
MOCSET X0=0,Y0=0
MOCSET K=10,L=10

## Set template dimensions Kt,Lt and origin
MOCSET Kt=3, Lt=3
MOCSET Xt0=1,Yt0=1

## Set block partition index limits
MOCSET NbX=3, NbY=3

## Set number of signal & noise bits in image (P,Q) and template (R,S)
MOCSET P=8, Q=2
MOCSET R=8, S=2

#ASC:: Perform KxL convolution over MxN image partition at 16 bits
#precision
ITGCNX (2,M,N,P,Q,X0,Y0, 2,Kt,Lt,R,S,Xt0,Yt0, +,16,x,16)

#ASC::For each block (U,V loops start here)
{{

#ASC::Apply quantization lookup table (I/O ops)
ILUTX (2,K,L,P,Q,X0,Y0, -1,-1,-1,-1,-1,-1,-1, -1,-1,M,16)

#ASC::Compute the block sum
ISUMX (2,K,L,P,Q,X0,Y0, -1,-1,-1,-1,-1,-1,-1, +,16,-1,-1)

#ASC::Scalar divide by 1/KL to get block mean
IMULX (2,1,1,16,0,0,0, 2,1,1,16,0,0,0, -1,-1,x,16)

#ASC::Output block mean
```

```
IOUTX (2,1,1,P,Q,0,0, -1,-1,-1,-1,-1,-1,-1, -1,-1,M,16)

#ASC::End Block Processing Loops
}*NbX }*NbY
```

LIST OF REFERENCES

[BOU99] A. Bouridane, D. Crookes, P. Donachy, K. Alotaibi, and K. Benkrid, "A high level FPGA-based abstract machine for image processing," *Journal of Systems Architecture*, Vol. 45, No. 10, pp. 809-824, 1999.

[CLE97] Mark J. Clement and Michael J. Quinn, "Automated performance prediction for scalable parallel computing," *Parallel Computing*, Vol. 23, No. 10, pp. 1405-1420, 1997.

[CYP90] R.E. Cypher, J. L. C. Sanz, and L.Snyder, "Algorithms for image component labeling on SIMD mesh-connected computers," *IEEE Transactions on Computers*, Vol. 39, No. 2, pp. 276-281, 1990.

[HAR90] D. T. Harper III and J. R. Jump, "Evaluation of reduced bandwidth multistage networks," *Journal of Parallel and Distributed Computing*, 9, pp. 304-311, 1990.

[HAY98] John P. Hayes, *Computer Architecture and Organization, Third Edition*, McGraw-Hill, Inc., Boston, 1998.

[HEN95] John L. Hennessy and David A.Patterson, *Computer Architecture: A Quantitative Approach, Second Edition*, Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1995.

[HWA93] Kai Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw-Hill, Inc., New York, 1993.

[KAR94] Helen D. Karatza, "Simulation study of a system with two processors linked in tandem," *Journal of Systems and Software*, Vol. 26, No. 3, pp. 285-292, 1994.

[KAR97] Helen D. Karatza, "Task routing and resequencing in a multiprocessor system," *Journal of Systems and Software*, Vol. 41, No. 3, pp. 189-197, 1998.

[KWI97] Kevin Kwiat, "Performance evaluation of a dynamically reconfigurable multiprocessing and fault-tolerant computing architecture," in *Proceedings of the 1997 Summer Computer Simulation Conference*, pp. 91-96, San Diego, CA, 1997.

[MEN96] D.A. Menasce and A. Rao, "Performance prediction of parallel applications on networks of workstations," *CMG Proceedings*, Vol. 1, pp. 299-308, 1996.

[NAP90] Leonard M. Napolitano, "The design of a high performance packet-switched network," *Journal of Parallel and Distributed Computing*, Vol. 10, pp. 103-114, 1990.

[NOH99] Sam H. Noh, Klaudia Dussa-Zieger, and Ashok K. Agrawala, "HeMM execution model for massively parallel system," *Journal of Parallel and Distributed Computing*, Vol. 56, No. 1, pp. 2-16, 1999.

[OBA93] M.S. Obaidat and Dirar S. Abu-Saymeh, "Performance of RISC-based multiprocessors," in *Computers & Electrical Engineering*, Vol. 19, No. 3, pp. 185-192, 1993.

[PAT97] David A.Patterson and John L. Hennessy, *Computer Organization and Design, the Hardware/Software Interface, Second Edition*, Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1997.

[RAT97] Nalini K. Ratha and Anil K. Jain, "FPGA-based computing in computer vision," in *CAMP Proceedings*, pp. 128-137, IEEE Computer Society Press, Los Alamitos, CA, 1997.

[RIT96] Gerhard. X. Ritter and Joseph. N. Wilson, *Handbook of Computer Vision Algorithms in Image Algebra*, CRC Press, Inc., Boca Raton, FL, 1996.

[SAU81] Charles H. Sauer and K. Mani Chandy, *Computer Systems Performance Modeling*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981.

[SCH98] Mark S. Schmalz, http://www.cise.ufl.edu/~mssz/AIM/Top-Level.html, 1998.

[SCH99] Mark S. Schmalz, Gerhard. X. Ritter and F. M. Caimi, "EBLAST – An efficient, high-compression image transformation ," in *Proceedings SPIE - The International Society for Optical Engineering*, Vol. 3814, pp. 73-85, 1999.

[SCH00] Mark S. Schmalz, http://www.cise.ufl.edu/~mssz/AIM/MOC-vars.ps, 2000.

[SHE85] Ken Sherman, *Data Communications, A Users Guide, Second Edition*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1985.

[STA96] William Stallings, *Computer Organization and Architecture, Designing for Performance, Fourth Edition*, Prentice-Hall, Inc., Upper Saddle River, NJ, 1996.

[SUN90] Myung Hoon Sunwoo and J. K. Aggarwal, "Flexibly coupled multiprocessors for image processing," *Journal of Parallel and Distributed Computing*, Vol. 10, pp. 115-129, 1990.

[WAN90] C. J. Wang, C. H. Wu, and V. P. Nelson, "A Comparative architectural study of three MIMD computing surfaces," *IEEE Proceedings Computers and Digital Techniques*, Vol. 137, No. 4, pp. 261-268, 1990.

[XIA97] XiaoQiang Xiao, HuaPing Hu, and ShiYao Jin, "Performance analysis of k-ary n-cube interconnection networks," in *Proceedings of the 1997 Summer Computer Simulation Conference*, pp. 107-111, San Diego, CA, 1997.

[YAN90] Qing Yang and Laxmi N. Bhuyan, "Perforamnce of multiple-bus interconnections for multiprocessors," *Journal of Parallel and Distributed Computing*, Vol. 8, pp. 267-273, 1990.

[ZAL96] John R. Zaleski, "RS/6000 processor performance modeling a methodology for IBM RISC 6000 model 990 modeling and performance evaluation," *CMG Proceedings*, Vol. 2, pp. 1002-1011, 1996.

[ZEI90] Bernard P. Zeigler and Guoqing Zhang, "Mapping hierarchical discrete event models to multiprocessor systerms: concepts, algorithm, and simulation," *Journal of Parallel and Distributed Computing*, Vol. 9, pp. 271-281, 1990.

BIOGRAPHICAL SKETCH

Yue Yin was born in Wuxi, P.R.China, in October 1976. She received a B.S. degree in computer science from Nanjing University, P.R.China, in July 1998. She expects to receive her Master of Science degree in computer engineering from the University of Florida in August 2000.