Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

THE VIRTUAL-TIME DATA-PARALLEL MACHINE

Shioupyn Shen

January 1992
CSD-920001

UNIVERSITY OF CALIFORNIA

Los Angeles

The Virtual-Time Data-Parallel Machine

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Shioupyn Shen

1991

#7

The dissertation of Shioupyn Shen is approved.

Kirby Baker

Christopher Anderson

Milos Ercegovac

Jack Carlyle

Leonard Kleinrock, Committee Chair

University of California, Los Angeles

1991

To my family,

for providing me with the love, support and freedom

to indulge me in pursuing my best.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

Many people played a part in this dissertation. I would like to express my appreciation to all of them.

First and foremost is Professor Leonard Kleinrock. His immeasurable encouragement and knowledgeable guidance have been of tremendous value both academically and personally. His overwhelming charm makes him the ultimate model for me to follow. It is really a pleasure to be his advisee.

Many thanks to Jack Carlyle, Milos Ercegovac, Christopher Anderson, and Kirby Baker for serving on my committee. Their patience and suggestions have been indispensable to the completion of this dissertation. I also own my thanks to David Jefferson, Rajive Bagrodia, and Eliezer Gafni for their lectures, which provided the background and sparked the key ideas in my research.

This research has been supported by the Advanced Research Projects Agency of the Department of Defense under contract MDA 903-87-C0663, Parallel Systems Laboratory. The PSL is a wonderful place to do research. I offer my sincere thanks to former classmates Willard Korfhage, Farid Mehovic, Joy Lin, and current classmates Chris Ferguson, Bob Felderman, Simon Horng, and Jonathan Lu for making the environment stimulating and productive. To Bob especially, I owe a great deal. He is always there to help me. Special thanks to Lily Chien for handling everything in this research group.

To Waishan Wu, I dedicate this dissertation. Her love and understanding made the past four years of my life happy and productive. Finally, I want to thank my parents for providing me the best environment to pursue my dream.

VITA

1962	Born, Taipei, Taiwan, Republic of China
1985	B. S., Electrical Engineering National Taiwan University
1985–1986	Research Assistant under the Wisconsin Alumni Research Foundation, University of Wisconsin, Madison
1986	M. S., Electrical Engineering University of Wisconsin, Madison
1987	M. S., Computer Science University of Wisconsin, Madison
1988–1991	Graduate Student Researcher under the Parallel Systems Laboratory contract, University of California, Los Angeles

PUBLICATIONS

Shioupyn Shen, "Dispersive Pulse Propagation in a Multiple-Resonance Medium", Master Thesis, University of Wisconsin, Madison, December 1986.

Kurt E. Oughstun and Shioupyn Shen, "Velocity of Energy Transport for a Time-Harmonic Field in a Multiple-Resonance Lorentz Medium", *Journal of the Optical Society of America B*, vol. 5, no. 11, pp. 2395–2398, November 1988.

Shioupyn Shen and Kurt E. Oughstun, "Dispersive Pulse Propagation in a Double-Resonance Lorentz Medium", *Journal of the Optical Society of America B*, vol. 6, no. 5, pp. 948–963, May 1989.

Abstract of the Dissertation

The Virtual-Time Data-Parallel Machine

by

Shioupyn Shen
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 1991
Professor Leonard Kleinrock, Chair

We propose the "Virtual-Time Data-Parallel Machine" to execute SIMD (Single Instruction Multiple Data) programs *asynchronously*.

First, we illustrate how asynchronous execution is more efficient than synchronous execution. The inefficiency of synchronous execution of SIMD programs comes from unnecessary blocking. Processors that finish the current instruction early are blocked until all processors finish executing this instruction, even though the operands they need in order to execute the next instruction may be available. The Virtual-Time Data-Parallel Machine lets a processor execute the next instruction as soon as its operands are ready. We show, for a simple model, that asynchronous execution outperforms synchronous execution roughly by a factor of ($\ln N$), where N is the number of processors in the system.

Second, we explore how to execute SIMD programs asynchronously without violating the SIMD semantics. The new problem introduced by asynchronous execution is that processors are not allowed to overwrite their variables because the previous values may be needed by other processors. *"No overwrite"* can be solved by keeping a *memory history* that stores the previous values of variables in every processor. For the memory history, we design cost-effective hardware support which implements incremental backup for the main memory so that previous values can be retrieved on demand.

Third, we analyze the performance of this Virtual-Time Data-Parallel Machine under some simple assumptions. The performance measure is the average progress rate (r) of instruction execution. Several models are developed to give an upper bound, a lower bound, and an approximation to r, and as well as the limiting value of r for a large number of processors.

Fourth, we generalize these simple assumptions and then evaluate the performance by simulation. We also address some directions to further improve the efficiency of asynchronous execution. In all these cases we find, as in the simple model, that asynchronous execution is superior to synchronous execution. In summary, the Virtual-Time Data-Parallel Machine effectively converts the SIMD computation from (synchronous) control-flow to (asynchronous) dataflow.

Finally, an architecture simulator and an assembler are developed. Statistics of running a real program on the simulator are collected which support the argument that for computation intensive data-parallel programs, *the Virtual-Time Data-Parallel Machine can achieve LINEAR SPEED-UP for a large number of processors.*

CHAPTER 1

Introduction

1.1 High Performance Computers

Human beings will always find ways to utilize more and more computing power. More computing power is mandatory to solve more complicated problems with better accuracy, to simulate more complex systems with better confidence, to move background computations to real-time interactive computing and most importantly, to open the door to the previously unsolvable. High performance computers have become the vital enabling force in many areas of science and engineering research, for example, in climate modeling, fluid turbulence, quantum chromodynamics, VLSI design, superconductor modeling, structure biology, vision and cognition. The future of these fields is highly correlated to the progress of high performance computers. The absolute processing power of high performance computers is of such strategic value that the cost-effectiveness of high performance computers is not merely judged in terms of \$/MIPS.

For the past twenty years, solid state technology has been much more successful in reducing the cost of VLSI chips than in increasing the peak speed of ECL circuits. Even though the priceless strategic value of high performance computers still exists, it does not pertain to supercomputers any longer. Twenty years ago, when the microprocessor first came out, supercomputers were about one thousand times faster and ten thousand times more expensive than microprocessor-based personal computers. Now, supercomputers are only ten times faster but still one thousand times more expensive than RISC microprocessor-based workstations[1]. The strategic value of classic supercomputers is fading away slowly but definitely. The rising star for next generation supercomputing is parallel processing. The key to high performance is the high degree of parallelism instead of the high speed of a single processor.

1.2 The Parallel Processing Challenge

Parallel processing is by no means a new concept. It is simply not yet in the main stream of the computer industry.

During the 1960s, architecture design was already pretty mature. Machines such as the CDC 6600 [Tho61], the IBM 360/91 [Tom67] and the CRAY-1 [Rus78] were so beautifully designed that their achievements overshadowed the research in computer architectures for the next twenty years. The research emphasis in the 1970s and early 80s was in some sense to find the optimal subset of the supercomputer approaches that was best fitted to the VLSI technology at that time. As VLSI technology advances, more and more transistors can be put into a single chip and new architectures are invented to utilize these extra transistors. Many "new" architectures were derived from re-discovering the old ideas of the early 1960s. Technological advancements have made approaches which were expensive and peculiar in the past affordable and popular now. Those who knew how to take advantage of the new technology the best took

[1]The latest RISC workstation from HP is rated at over 70 mips for under ten thousand dollars.

the easy ride of *the new technology on old architectures* in developing faster and fancier uniprocessor systems. On the other hand, research on new parallel architectures was stunted because pioneering research was not making much money for its sponsors.

The parallel processing challenge in the past was how to compete with the easy ride on established architectures. Parallel processing lost the war as everyone predicted. In fact, parallel processing is destined to failure as long as the easy ride continues. This will not end for a while because current state-of-the-art VLSI microprocessors haven't yet caught up with the complexity of the supercomputers of the 1960s[2]. When we can implement a classic supercomputer on a single VLSI chip, the easy ride will be over because choosing the optimal subset is no longer interesting if we can have the whole thing.

As the chip density keeps on increasing, soon there will be more transistors in a single chip than uniprocessor architectures can use effectively. What should be done next? Anyone who wants to take the lead in high performance computing in the future has to prepare a solution and pursue it when the time is right. Everybody knows that parallel processing is the right direction in which to go because it is a direct way to make use of a large number of transistors. However, as mentioned at the beginning of this section, parallel processing is by no means a new idea. The problems of parallel processing in the past will still be problems in the future. Even though the major competitor of parallel processing, i.e., the use of modern VLSI technologies on ancient uniprocessor architectures, is weakening, parallel processing is not yet strong enough in itself. Parallel processing did not attract sufficient attention when other approaches

[2]There are approximately one million transistors in the Intel 80486 but there were two million transistors in the CRAY-1.

were more profitable. As it is now becoming clear that parallel processing is inevitable in the near future, we find that more federal funding is becoming available, and that more computer scientists are devoting themselves to parallel processing.

The challenge of parallel processing is how to express and utilize the parallelism effectively; that is, how to write parallel programs easily and how to implement parallel hardware efficiently. These two problems prevailed in the past and there were no satisfactory solutions. Recently, we have seen some partial solutions. The Connection Machine [Hil85] and Virtual Time [Jef85] are two of the most promising approaches. The Connection Machine provided a simple and efficient way to develop parallel programs. Virtual Time provided a transparent and effective way to improve hardware efficiency. They will be described in detail in the next section. In short, for the grand challenge of parallel processing, both the Connection Machine and Virtual Time are only partial solutions, and their merits are complementary. We choose to use the concept of Virtual Time to improve the efficiency of the Connection Machine, and thus combine the merits of both of these partial solutions.

1.3 Related Work

The concepts of the Virtual-Time Data-Parallel Machine are mainly derived from "The Connection Machine" [Hil85] and "Virtual Time" [Jef85]. In the next two sub-sections, we give a brief description of the Connection Machine and of Virtual Time.

1.3.1 The Connection Machine

Figure 1.1 shows the architecture of the Connection Machine. The characteristics of the architecture of the Connection Machine are as follows:

SIMD: A front-end broadcasts the instruction stream to all processing elements. All processors execute the same instruction stream on their own data.

Distributed Memory: Memory is distributed to the processing elements. Every processor has its own local memory and can access remote memory on the other processors through an interconnection network.

Massively Parallel: The number of processors is very large, and usually approaching the intrinsic parallelism of the problem being solved. The potential speed-up of the machine is therefore very large.

Programmable Connections: The interconnection network is a hypercube with embedded 2-D and 3-D grids. The *logical* interconnection can be programmed dynamically to support the communication between arbitrarily selected processors such that the fixed *physical* wiring scheme is invisible for the software.

The Connection Machine was proposed by Daniel Hillis as an alternative to the von Neumann architecture [Hil85]. In a large von Neumann computer, almost all the transistors are in the memory section of the machine, and only a few of those memory locations are accessed at any given time. A well designed von Neumann computer keeps the silicon devoted to the processor pretty busy, but the rest of the silicon devoted to the memory sits idle most of the time,

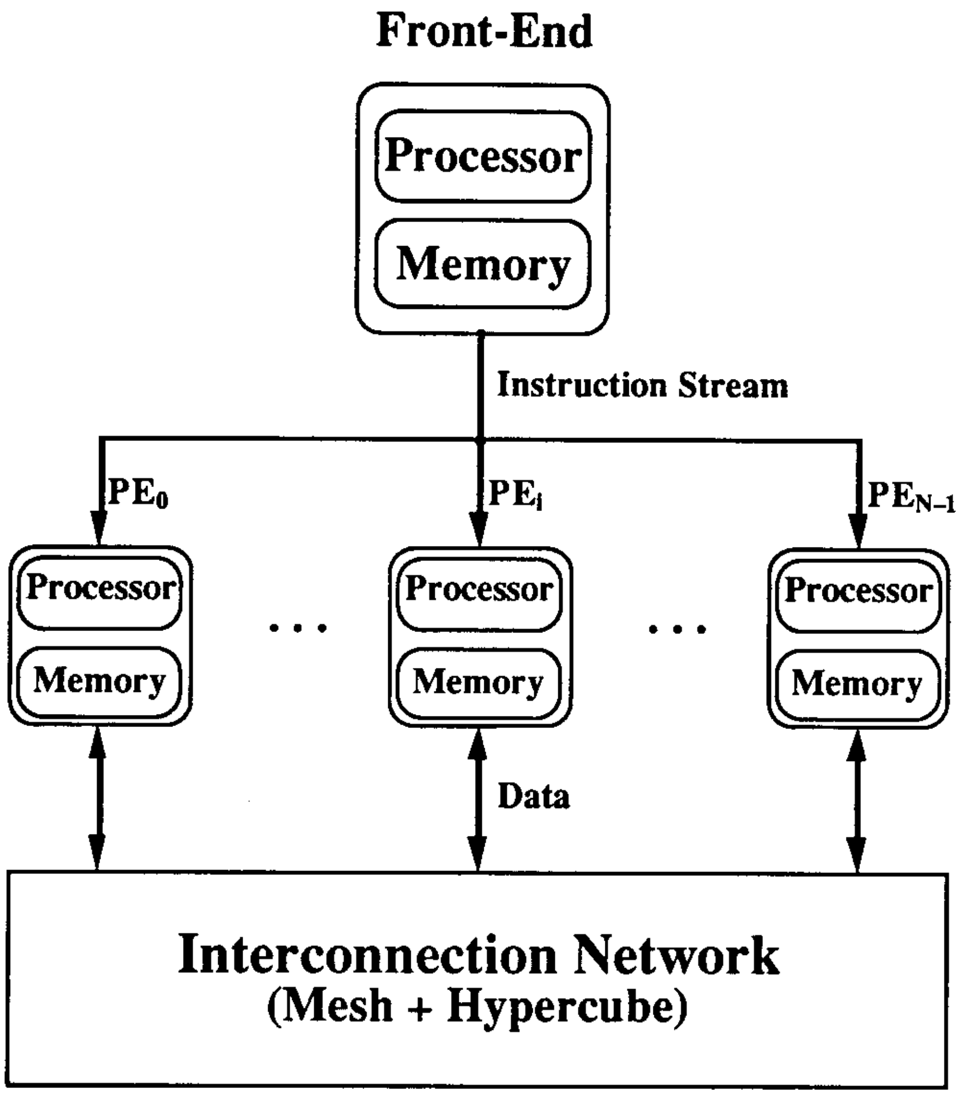


Figure 1.1: The Architecture of the Connection Machine.

which results in a vast wasted resource. The solution to this problem is to break the boundary between the processor and memory by distributing processing power closer to the memory so that the utilization of the silicon area is more balanced. As a consequence, the Connection Machine consists of a large number of tiny processor–memory pairs[3]; its architecture is thus named "Massively Parallel", reflecting the characteristic of having a very large number of processors. Compared to a von Neumann computer with the same number of transistors, the Connection Machine spends more transistors on the processor so that it has greater raw aggregate computing power. Moreover, the memory–processor bandwidth of the Connection Machine is also larger because many memory locations can be accessed concurrently by the large number of processors. Another important issue of this parallel architecture is the choice of SIMD (single instruction multiple data) vs. MIMD (multiple instruction multiple data). The Connection Machine chose SIMD based on the argument that for well-structured problems with regular patterns of control, SIMD machines have the edge, because more of the hardware is devoted to operations on data instead of control. The Connection Machine's interconnection network in fact has two distinct parts. The *general-purpose* network is a hypercube, which is used for communications among arbitrarily selected processors. The hypercube topology is chosen to maintain a low maximum distance between processors with a reasonable cost. The *special-purpose* network is a mesh network, which is embedded in the hypercube[4]. The mesh network is especially designed for those applications that mainly fetch data from immediate neighbors.

[3]For example, the CM-2 has 64K tiny bit-serial processors.

[4]The links of a mesh network are a subset of that of a hypercube.

SIMD machines have been available for more than twenty years, but none has been as successful as the Connection Machine. One of the most significant pioneering works on SIMD architectures was the ILLIAC IV [Bar68] [Kuc68] [Bou72] developed at the University of Illinois. Two comparable approaches to the Connection Machine are the Distributed Array Processor (DAP) [Fla77] [Par90] developed by Active Memory Technology and the Massively Parallel Processor (MPP) [Bat80] [Pot85] developed by Goodyear Aerospace Corporation. A competitor of the Connection Machine is the MasPar MP series[5], but MasPar only competes on the basis of price instead of on pioneering research. Refer to [Mar91] for more up-to-date research in massively parallel computing, and [Kuc77] [Hay82] [Hor90] for general surveys of highly parallel architectures and SIMD supercomputers.

The data-parallel computational model [Chr83] [Baw84] [Hil86] [Ble90] of the Connection Machine distinguishes the Connection Machine from other SIMD machines, and accounts for its big success. The revolutionary data-parallel computational model makes good use of as many processors as the intrinsic parallelism of the problem being solved. For well-structured large problems, the intrinsic parallelism is usually on the order of ten thousand to one million, or even more. A Connection Machine with tens of thousands of processors has no difficulty utilizing them in such well-structured large problem. This is in contrast to other kinds of parallel machines, which have a hard time dealing with as few as one hundred processors.

Let us now give some examples of data-parallel programming. In all of these examples, elements of big arrays are distributed among the processor–memory

[5]MasPar Computer Corporation, 749 N. Mary Avenue, Sunnyvale, CA 94086.

cells, e.g., the i-th element of an array is stored on the i-th processor–memory cell.

Example a)

```
for all i, x[i] = y[i] + z[i];
```

Every processor fetches operands from its local memory and stores the result of the operation in its local memory. The whole operation is executed inside each processor–memory cell. Since there is no data exchange through the interconnection network, the execution time of this instruction is quite deterministic. Suppose there are N processor–memory cells, there are N computations and up to $2N$ memory references executed in parallel. The aggregate throughput and memory bandwidth of the Connection Machine is much larger than traditional von Neumann supercomputers with the same number of transistors.

Example b)

```
for all i, x[i] = y[i-1] + z[i+1];
```

Every processor needs one operand each from its left and right neighbors. The requested data goes through the special mesh interconnection network, and the execution time of this instruction is somewhat nondeterministic. There are on the order of N operations executed concurrently.

Example c)

```
for all i, x[i] = y[z[i]];
```

Every processor needs one remote operand, and the location of the remote operand is random. The requested data goes through the general hypercube interconnection network, and the execution time of this instruction is rather random. Nevertheless, there is still a large amount of parallelism.

Example d)

```
for all i, if (is_even(i)) x[i] = x[i] + x[i+1];
```

Processors execute the instruction conditionally. Only the processors satisfying the condition execute the instruction. If few processors meet the condition, then the utilization of the system can be dreadfully low.

There are other paradigms for data-parallel programming. The purpose of these examples is to illustrate the flavor instead of the details of the data-parallel computational model.

Data-parallelism makes it worthwhile to go through the trouble of parallel programming. A speed-up of tens of thousands is very appealing. Currently, data-parallelism is probably the only feasible approach for teraflop machines by the end of the decade. In addition, data-parallelism is extremely easy to program. The SIMD semantics of the data-parallel computational model make it simple and effective to develop massively parallel programs. The SIMD machine eliminates the problem of explicit synchronization, and alleviates the cost of global synchronization as well.

This section has given a concise introduction to the massively parallel architecture and data-parallel programming of the Connection Machine. More information about the Connection Machine can be obtained from Thinking

Machines Corporation[6].

1.3.2 Virtual Time

The concept of Virtual Time [Jef85] originates from research in parallel discrete event simulation. Correct implementations of discrete event simulation must satisfy the causality constraint. The causality constraint says that if event A has an earlier simulation time than event B, then event A must be executed before event B because the execution of event A *may* change the environment of event B. A direct implementation of discrete event simulation is to sort all outstanding events in the order of increasing simulation time and execute them in sequence. There are many approaches to parallelize discrete event simulation. For simulating continuous-time systems, time-stepped simulation can be used by simulating the system at fixed time intervals [Pea79]. However, this kind of simulation is not typically "discrete" because the simulation time is continuous.

Conservative parallel discrete event simulations are based on the pioneering works of K. Mani Chandy, Jayadev Misra and R. E. Bryant [Bry77] [Cha79] [Cha81] [Mis86], and numerous follow-up work which enhances the basic algorithm [Bai88] [Fuj89a] [Gro88] [Gro89] [Lub89] [Nic84] [Nic88] [Su89]. Conservative methods concurrently execute those events whose environments will definitely not change. It is difficult in general to *make sure* that the environment of an event will not change, even though we know that the environment is *very unlikely* to change.

The *optimistic* approach takes advantage of "likelihood" instead of "certainty". The Time Warp synchronization mechanism, based on the Virtual

[6]Thinking Machines Corporation, 245 First Street, Cambridge, Massachusetts 02142-1214

Time paradigm, is the most well-known optimistic protocol [Jef85]. Here, virtual time is a synonym for simulation time. Optimistic methods concurrently execute those events whose environments are very unlikely to be changed by assuming that the environments will not change, and then undo the execution for those (hopefully few) events for which the assumptions have been violated.

Figure 1.2 shows the key concept of Virtual Time. In the task graph, nodes correspond to events and links correspond to causality constraints. In this example, the environment of event X *may* depend on events A, B ,D, E and G (i.e., the *ancestors* of event X). However, not all ancestors actually modify the environment of event X. After all ancestors are finished, we find out that only events B and D change the environment of event X. These ancestors are called "relevant" ancestors and the rest of the ancestors (i.e., events A, E and G) are called "irrelevant" ancestors. The execution of an event cannot proceed until all its relevant ancestors are finished. In the conservative approach, the execution of event X has to wait until *all* the ancestors of event X are finished. This constraint can be relaxed to the less strict condition that all the ancestors of event X either are finished or give consent to event X that they will not modify the environment of event X. The performance of this approach is not satisfactory because of the extra waiting for the irrelevant ancestors either to finish or to give consent. Waiting for an irrelevant ancestor (e.g., event X waits for event A) is called a "false alarm" because the waiting turns out to be unnecessary. The optimal approach is to wait for the relevant ancestors only, but the optimal approach is not realizable because whether an ancestor is relevant or not is often unknown in advance. The optimistic approach suggests not to wait for any *non-local* ancestors[7].

[7]In parallel discrete event simulation, events are distributed among processors. Local an-

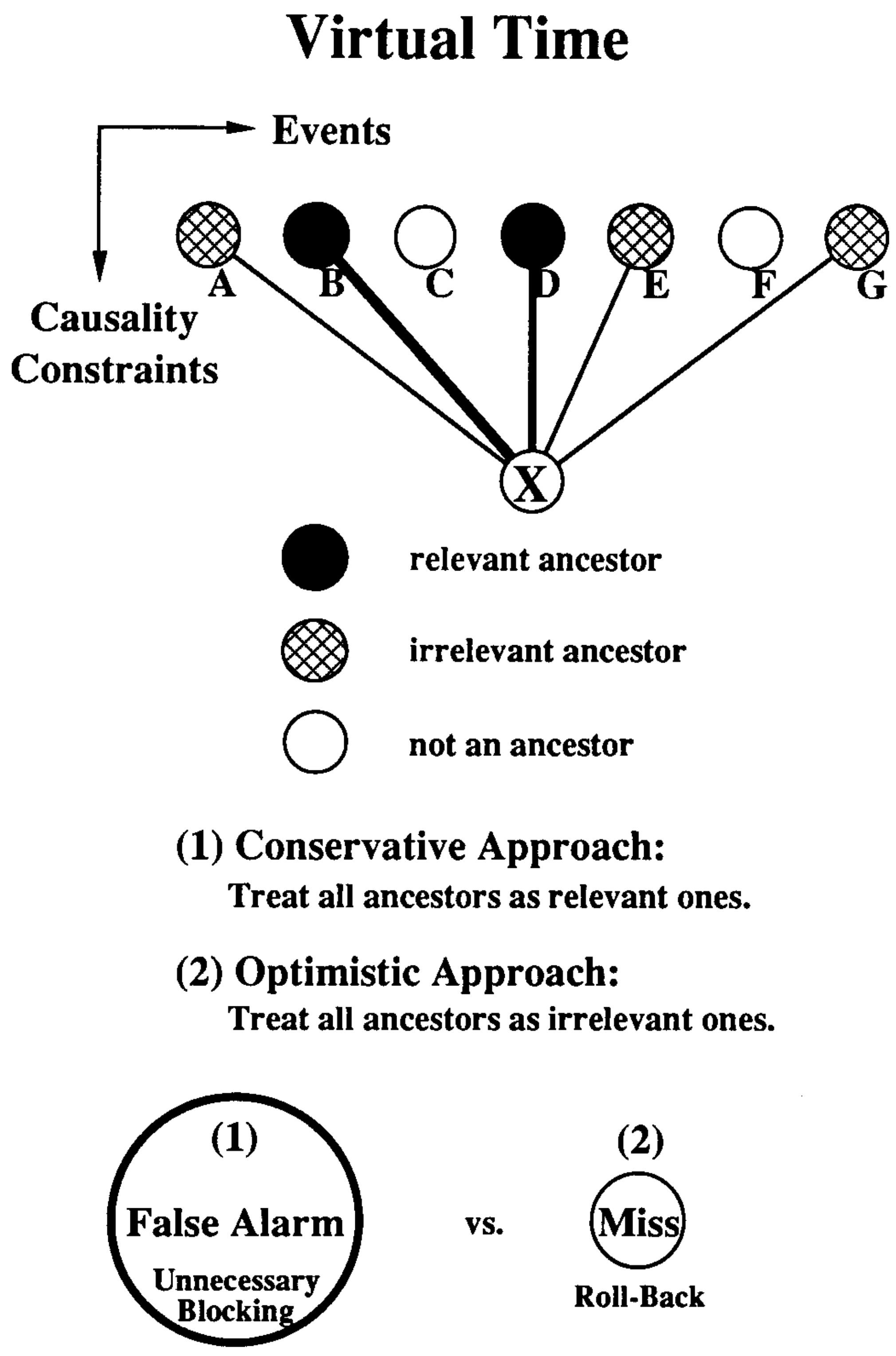


Figure 1.2: Concepts of Virtual Time – No False Alarms.

The performance of this approach is good but the result of executing an event without waiting for its relevant ancestors is wrong because the environment of the event is incorrect (i.e., not up-to-date). Not waiting for a relevant ancestor (e.g., event X does not wait for event B) is called a "miss" because the execution is incorrect. In order to get a correct result, a miss has to be handled properly by roll-back. Since we cannot tolerate any miss, a miss, once it is identified, must be rolled-back to the point before the execution of the miss.

State-saving is necessary for roll-back to be possible. The major overhead of roll-back is on the periodic state-saving rather than the roll-back itself. When roll-back actually happens, it does not take much time for the context switch in order to restore the previous state. What is lost during a roll-back is the (false) gain in progress of the optimistic approach. Indeed, after the roll-back, the processor state is equivalent to having been blocked there in the first place. Thus, rolling-back does not cause additional performance loss because otherwise (i.e., for the conservative approach), the processor would have been blocked and the performance would have been lost anyway. The performance loss of a roll-back refers to the loss of the performance *gain*, i.e., what is gained from the optimistic approach is lost. The cost (i.e., the overhead and performance loss) of rolling-back one miss is usually larger than the time wasted in waiting for one false alarm. However, in many applications, there are usually few relevant events (candidates for miss) but many irrelevant events (candidates for false alarm). The optimistic approach is preferred when the cost of roll-back is relatively low and the number of relevant events is relatively small. Refer to [Fuj90] and [Mad91] for more up-to-date research in parallel discrete event simulation.

cestors are the ancestors which happens to be on the same processor.

The implementation details of the Time Warp synchronization mechanism are not discussed here. Instead, a high level abstraction of Virtual Time is presented in this section. The strategy of Virtual Time is that by saving the current state, processors are always free to go ahead instead of being blocked frequently. In general, if a processor is blocked for some reason that can be solved by state-saving, then it may be worthwhile to spend some extra memory space on state-saving in exchange for the unblocking of the processor.

1.4 The Virtual-Time Data-Parallel Machine

The difficulties of parallel processing are two-fold. The first problem is that the computational model is hard to use (i.e., it is hard to develop programs on parallel processing systems, and to port programs from one system to another), and the second problem is that the hardware efficiency is poor for a large number of processors. We propose the "Virtual-Time Data-Parallel Machine" to solve these two problems at once. The Virtual-Time Data-Parallel Machine is derived from the pioneering works of Daniel Hillis (The Connection Machine), and David Jefferson (Virtual Time).

The Connection Machine introduced the data-parallel computational model. The SIMD semantics of the data-parallel computational model make it easy[8] to develop parallel programs and make it capable of expressing fine-grain parallelism. Though the Connection Machine achieves significant speed-up for large numbers of processors, hardware efficiency is poor because of the synchronous execution. Virtual Time introduced the Time Warp synchronization mecha-

[8]Parallel programming is hard in general. However, my own experience, and that of others, has shown that data-parallel computational model is much easier than other approaches.

nism for parallel discrete event simulation. The optimistic approach of Time Warp synchronization mechanism eliminates unnecessary blocking, and therefore makes better use of the hardware. However, it is hard to generalize Virtual Time to other parallel processing applications. Thus, we propose to use Time Warp to execute data-parallel programs asynchronously in hopes of exploiting more parallelism and obtaining better efficiency.

The inefficiency of synchronous execution of SIMD programs comes from unnecessary blocking. Processors that finish the current instruction early are blocked until all processors (i.e., the last one) finish this instruction, even though the operands they need to execute the next instruction may be available. The Virtual-Time Data-Parallel Machine lets a processor execute the next instruction as soon as the operands it needs are ready.

The key to the operation is to store a history of the values of each variable at every point in time during the execution of the program. Then, every (address, virtual-time) pair effectively represents a write-once variable. The (address, virtual-time)-pairs of the operands of an instruction specify the essential data dependencies of this instruction. The execution of the next instruction can proceed independently of the executions of the current instructions on the other processors as long as its own data dependencies are satisfied. This effectively converts the computation from control-flow to data-flow. Data-flow execution is favorable because it is less sensitive to long and unpredictable network delays.

The contribution of the Virtual-Time Data-Parallel Machine is to combine of the merits of the Connection Machine and Virtual Time. The main idea is the adoption of the Time Warp synchronization mechanism to execute SIMD programs asynchronously. This improves the efficiency of the data-parallel machine, and at the same time, preserves its SIMD semantics.

1.5 Asynchronous SIMD

In this section, we illustrate how asynchronous execution of SIMD programs is more efficient than synchronous execution of SIMD programs.

Figure 1.3 is an example task graph of a SIMD program. The nodes correspond to tasks (i.e., instructions) and the links correspond to data dependencies. The rows of tasks represent the fact that instructions are executed on all processors in parallel. The columns of tasks and the vertical links represent the fact that a processor must execute instructions in sequence. In this example, there are four processors and each processor has three instructions to execute. Note that each task takes one remote operand and, therefore, the number of ancestors of each task is two. Figure 1.3.a shows the pure data dependencies of the program, and ignores any artifacts due to the execution model.

When synchronous execution is enforced, it is equivalent to adding more links to the task graph so that every task depends on all the tasks one row above it. Figure 1.3.b shows the data dependencies of the program in the synchronous execution model. Originally (Fig. 1.3.a), every task had two ancestors, which means every instruction must wait for two processors. For synchronous execution (Fig. 1.3.b), every task has four ancestors, where four is the number of processors in the system. That means every instruction must wait for all the processors in the system to finish the previous instruction.

We know that adding links to the task graph degrades the performance of the system, and removing links from the task graph improves the performance

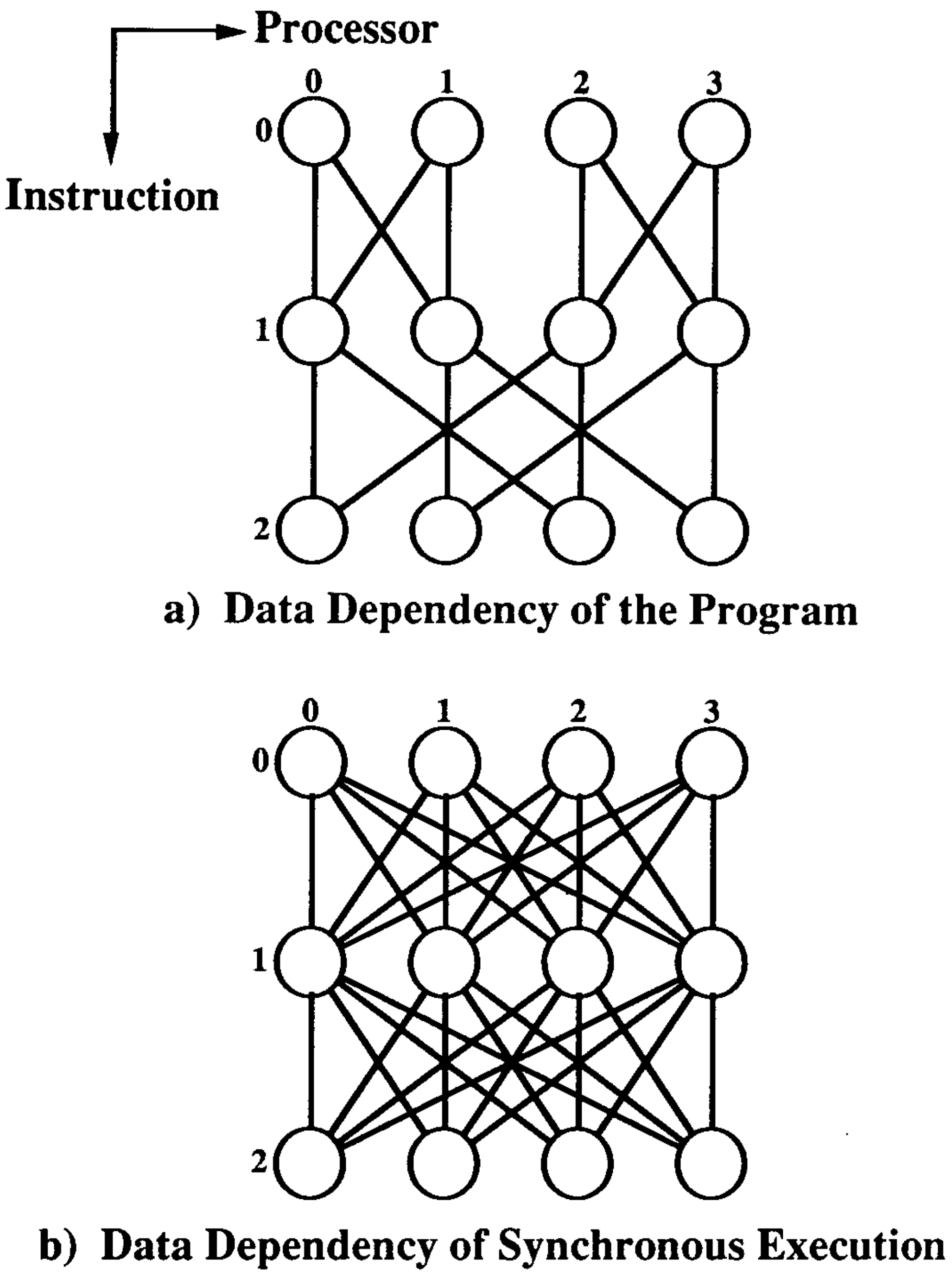
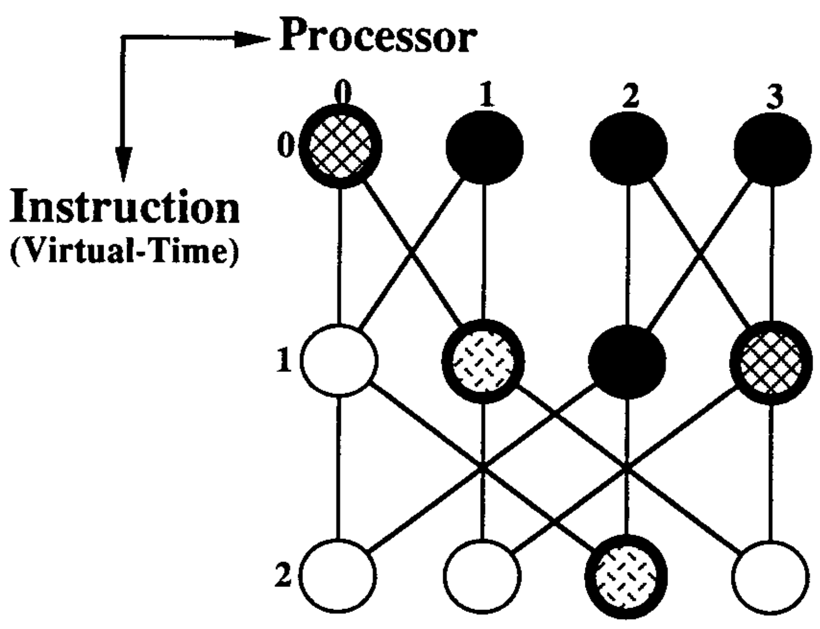

Processor
Instruction
0
1
2
3
0
1
2
a) Data Dependency of the Program
0
1
2
3
0
1
2
b) Data Dependency of Synchronous Execution

Figure 1.3: The Task Graph Representation of Data Dependencies.

of the system. The goal of the Virtual-Time Data-Parallel Machine is to promote asynchronous execution of SIMD programs by removing the extra links associated with synchronous execution and, at the same time, to preserve the original data dependencies of the program. In summary, the synchronous execution of SIMD programs changes the task graph from Fig. 1.3.a to Fig. 1.3.b by adding redundant links to enforce synchronous execution, and the asynchronous execution of SIMD programs changes the task graph from Fig. 1.3.b back to Fig. 1.3.a by removing those redundant links.

Figure 1.4 is a snapshot of an SIMD task graph during asynchronous execution of the program. Let us define some terminology. Virtual-time (`vt`) is defined to be the sequence number of the instruction stream. The virtual-time of a processor executing the i-th instruction is i. For example, in Fig. 1.4, processor 0 is at `vt` 0, processor 1 and 3 are at `vt` 1, and processor 2 is at `vt` 2. For each processor, all the instructions prior to the current instruction are finished. We say that these tasks are in the *finished* state. All the instructions after the current instruction are unreached, and we say that those tasks are in the *unreached* state. The current tasks in execution can be either running or blocked. If all the ancestors of a current task are in the finished state, then the task is in the *running* state because all the operands needed by the instruction are available. If one or more ancestors is not in the finished state, then the task is in the *blocked* state because some operands needed by the instruction are still unavailable.

Let T_N be the execution time of a data-parallel program with N physical processors, and let T_1 be the execution time of the same program with only *one* physical processor. The speed-up of a system with N processors is $\frac{T_1}{T_N}$, which



● **Finished**

○ **Unreached**

▩ **Running**
(All ancestors are in the Finished state)

▨ **Blocked**
(At least one ancestor is not in the Finished state)

$$\text{Speed-Up} = E[\text{Number of Running Processors}]$$

$$\text{Efficiency} = \frac{\text{Speed-Up}}{\text{Total Number of Proccessors}}$$

Figure 1.4: The Task Graph of Asynchronous Execution of SIMD Programs.

is equal to the expected number of processors in the running state, i.e.,

$$\text{Speed-Up} \triangleq \frac{T_1}{T_N} = E[\text{Number of Running Processors}] \tag{1.1}$$

The efficiency of the system is defined to be the probability that a processor is in the running state, i.e.,

$$\begin{aligned} \text{Efficiency} &\triangleq \text{Prob[a processor is running]} \\ &= \frac{E[\text{Number of Running Processors}]}{\text{Total Number of Processors}} \\ &= \frac{\text{Speed-Up}}{N} \end{aligned} \tag{1.2}$$

From Eq. (1.2), efficiency can be interpreted as the normalized speed-up. Let the *base* execution time of instructions be the average time to finish an instruction given all of its operands are available. The average execution time of instructions during the execution of the program will be larger than the base execution time because of the extra waiting time for unavailable operands. The progress rate (r) is defined to be the normalized rate of finishing instructions, i.e., the average number of instructions finished per processor per base execution time, i.e.,

$$r \triangleq \frac{1/\text{Expected Execution Time}}{1/\text{Base Execution Time}} \tag{1.3}$$

Furthermore, we know that

$$(\text{Expected Execution Time}) * (\text{Prob[Running]}) = (\text{Base Execution Time}) \tag{1.4}$$

From Eq. 1.2 and 1.4, the progress rate is equivalent to the efficiency of the system, i.e.,

$$\text{Progress Rate} \equiv \text{Efficiency} \tag{1.5}$$

We now can conduct some preliminary analyses and compare the performance of asynchronous vs. synchronous execution.

1.5.1 Assumptions

The SIMD machine consists of N homogeneous processors, i.e., every processor has the same processing power and executes the same instruction stream such that the behavior of every processor is *statistically* equivalent. The following are the assumptions we make throughout the dissertation in order to analyze the performance of both asynchronous and synchronous execution.

1. The execution time of instructions is exponentially distributed. Without loss of generality, let the mean execution time be 1.

2. The number of remote operands for each instruction is one, exactly. Every instruction always needs one remote operand.

3. The location of remote operands is uniformly distributed among the processors. Every processor is equally likely to hold the remote operand.

How realistic are these assumptions? They are not realistic at all. As for the first assumption, we discern that in reality, the execution time of instructions is more deterministic than memoryless. The result of more deterministic execution time is better performance for both asynchronous and synchronous execution, but synchronous execution benefits more, and hence the performance improvement of asynchronous over synchronous execution is less. Refer to Sec. 5.2 for more detail. As for the second assumption, the number of remote operands of an assembly instruction ranges from 0 to 2 in general. In reality, the average number of remote operands is less than one; often instructions only need local variables. We observe the fact that the relation between the register and memory is analogous to that between the local and remote operand. The register (local operand) reference is faster and more frequent than the memory (remote operand) reference, and the number of bits to address a register (local operand) is fewer than that to address a memory location (remote operand). Therefore, we can take the RISC[9] approach which uses explicit load/store instructions for remote access (one operand per instruction). For those instructions which load/store remote operands explicitly, there is one remote access; for those instructions which deal with only local operands, there is no remote access. From the RISC experience, confining each instruction to one or zero remote access hardly degrades the performance. The explicit load/store approach takes slightly more instructions, but this increase may be compensated for by a more uniform instruction length. For a given execution time distribution, asynchronous execution achieves better performance when there are fewer remote operands, but it makes no difference for synchronous execution. However, fewer remote references make the execution time more deterministic, which in turn benefits synchronous execution, too. Refer to Sec. 5.3 for more detail. As for the third assumption, the location of remote operands usually follows certain fixed patterns instead of uniformly distributed among processors. When an instruction on a particular processor takes a long time to finish, the tasks which are blocked directly or indirectly by this instruction form a data-dependency tree which is rooted at this instruction. The pattern of the remote access determines how seriously processors are affected by the long execution time of this

[9]RISC architectures have explicit load/store instructions to move data between the register file and the main memory (one word per instruction).

instruction. Fixed patterns usually reduce the effective branching factor of the data dependency trees, which is equivalent to reducing the expected number of remote operands. Refer to Sec. 5.4 for more detail.

In summary, these assumptions are by no means realistic, but they are simple enough to develop some insight on how asynchronous execution outperforms synchronous execution. Moreover, these assumptions are generally more pessimistic than reality. The performance of asynchronous execution based on the above assumptions is expected to be a lower bound on the actual performance, but the same argument does not apply to the performance *gain* of asynchronous over synchronous execution.

1.5.2 Performance Measures

Based on the assumptions in the previous section, we can calculate the performance of asynchronous and synchronous execution of SIMD programs. Figure 1.5 shows the speed-up and efficiency of the synchronous execution of SIMD programs from analysis [Fel90]. The efficiency of synchronous execution drops as the number of processors increases. The gap between its speed-up and the ideal linear speed-up also widens as the number of processors increases. It is not the same for asynchronous execution. Figure 1.6 shows the speed-up and efficiency of the asynchronous execution of SIMD programs from simulation. The efficiency, as well as the gap between its speed-up and the ideal speed-up, remains constant as the number of processors increases. It is obvious that asynchronous execution outperforms synchronous execution.

From Eq. (1.2), we get

$$\text{Speed-Up} = (\text{Total Number of Processors}) * \text{Efficiency} \tag{1.6}$$

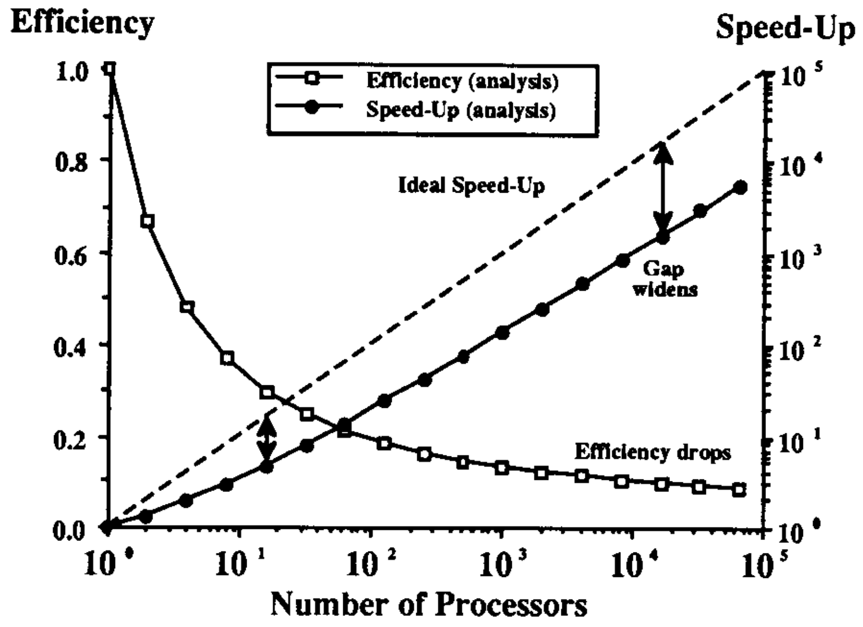


Figure 1.5: Speed-Up and Efficiency of Synchronous Execution.

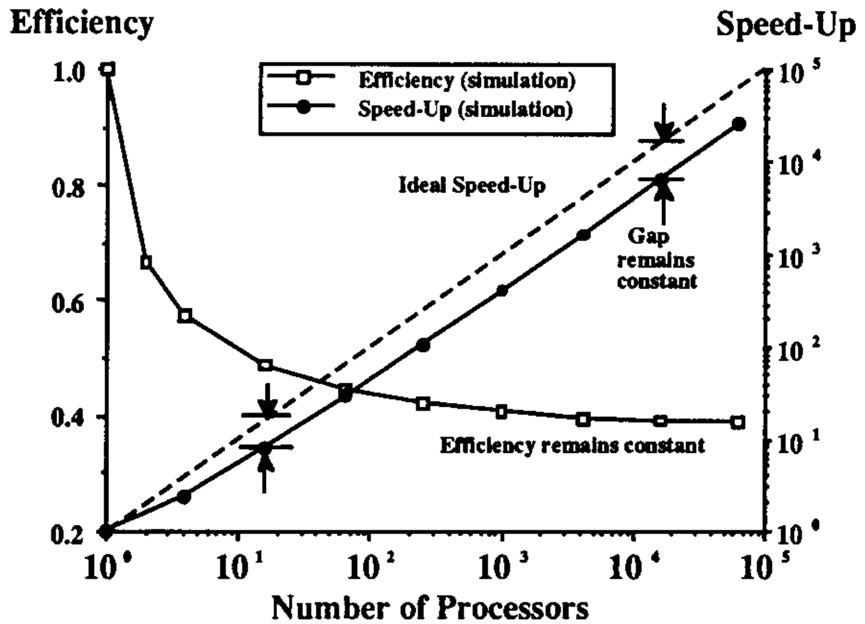


Figure 1.6: Speed-Up and Efficiency of Asynchronous Execution.

When the number of processors is fixed, speed-up is proportional to efficiency. Therefore, we will only discuss efficiency for the rest of the dissertation and we can always calculate the speed-up from Eq. (1.6).

Now we compare the efficiency of asynchronous vs. synchronous execution. Figure 1.7 puts together the efficiency curves of Fig. 1.5 and Fig. 1.6. The efficiency difference between asynchronous and synchronous execution increases as the number of processors increases. The efficiency gain of asynchronous over synchronous execution is defined to be the ratio of the efficiency of asynchronous execution and that of synchronous execution.

$$\text{Efficiency Gain} \triangleq \frac{\text{Efficiency of } \textit{Asynchronous} \text{ Execution}}{\text{Efficiency of } \textit{Synchronous} \text{ Execution}} \tag{1.7}$$

From Fig. 1.8, we find that the efficiency gain increases as the number of processors increases. Furthermore, Fig. 1.8 shows that the efficiency gain is in proportion to the logarithm of the number of processors.

1.6 Summary

This dissertation provides detailed information about the Virtual-Time Data-Parallel Machine. In Chapter 1 we have given a short introduction to the Virtual-Time Data-Parallel Machine and related work. We have illustrated how asynchronous execution is more efficient than synchronous execution. We have also shown, for a simple model, that asynchronous execution outperforms synchronous execution roughly by a factor of $(\log N)$, where N is the number of processors.

In Chapter 2 we give an in-depth discussion of the features of asynchronous execution of SIMD programs. We introduce GVT (global virtual-time), **rvt**

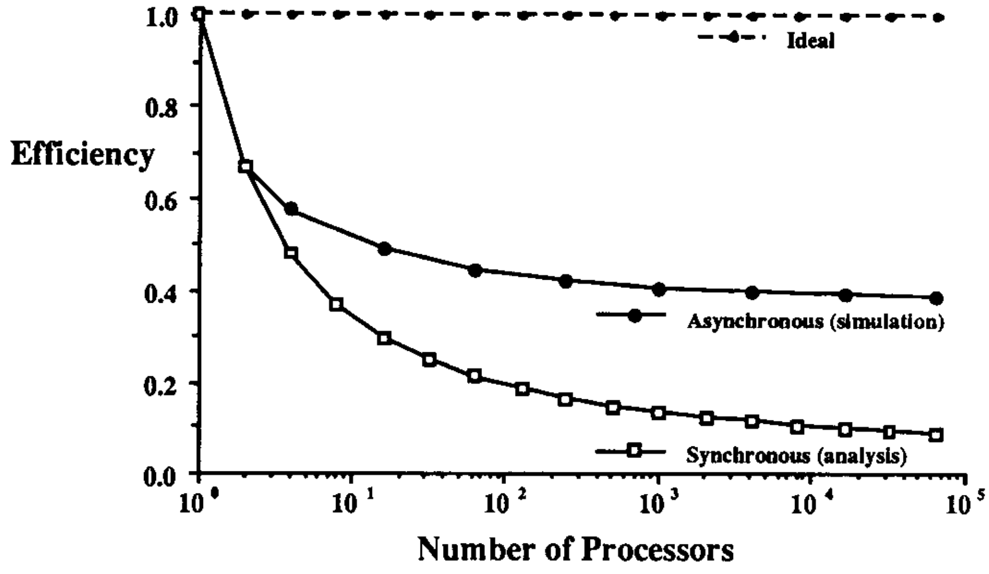


Figure 1.7: Efficiencies of the SIMD Machines – Asynchronous vs. Synchronous.

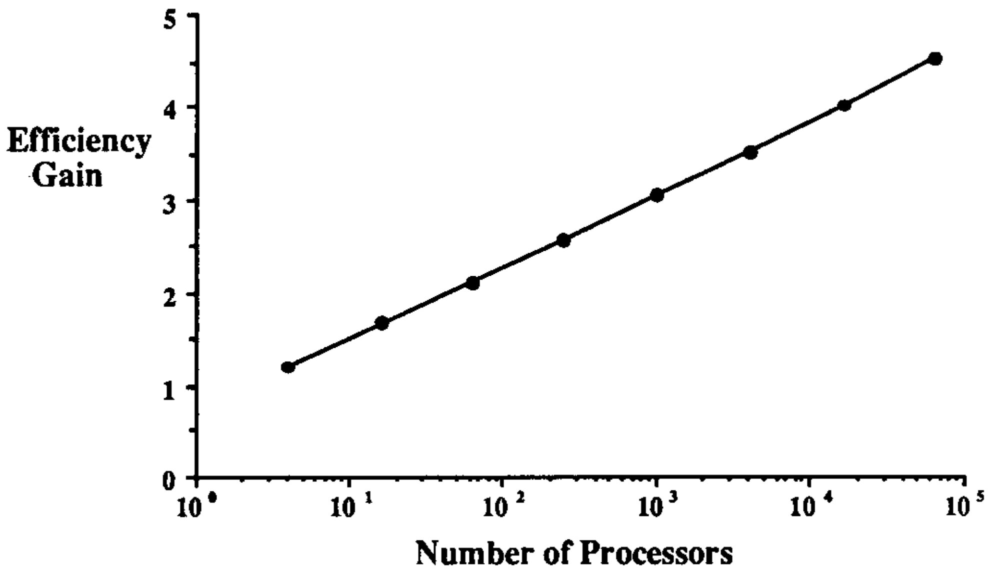


Figure 1.8: The Efficiency Gain – Asynchronous over Synchronous.

(relative virtual-time), and the processor histogram (the distribution of the number of processors at virtual-time relative to GVT). We illustrate how GVT governs the progress rate (r) of instruction execution. We also derive a simple approximation for the progress rate as a function of the expected number of processors at GVT.

In Chapter 3 we explore how to execute SIMD programs asynchronously without violating SIMD semantics. The new problem introduced by asynchronous execution is that a processor is not allowed to overwrite its variables because the previous values may be needed later by the other processors. "No overwrite" can be solved by keeping a memory history that stores the previous values of the variables in every processor. A small memory history is sufficient because the old values in the memory history are purged once it is no longer possible to reference them. The algorithm for maintaining memory history is incremental backup, which is a space-efficient algorithm. For the memory history, we design cost-effective hardware support, which is a FIFO (First-In-First-Out) priority cache. This FIFO priority cache implements the incremental backup algorithm for the memory history so that previous values can be retrieved on demand.

In Chapter 4 we develop several models to analyze the performance of the Virtual-Time Data-Parallel Machine. The performance measure is the progress rate (r) of instruction execution. The "non-persistent" model gives an upper bound on r, the "cold-start" model gives a lower bound on r, the "roll-back" model gives an approximation to r, and the "infinite-processor" model gives the limiting value of r for a large number of processors.

In Chapter 5 we address some extensions of the Virtual-Time Data-Parallel

Machine. This allows us to generalize the basic assumptions which we make throughout the dissertation. Performance evaluations based on the generalized assumptions are obtained from simulation. Other directions to further improve the performance of the machine are also discussed. With the extension, the Virtual-Time Data-Parallel Machine effectively converts the SIMD computation from control-flow to data-flow. Data-flow execution is favorable because it is less sensitive to long and unpredictable network delays.

In Chapter 6 we give a concise conclusion and examine some directions for future work in extending our understanding and verifying the usefulness of the Virtual-Time Data-Parallel Machine. The concept of asynchronous execution of SIMD programs is still in its infancy. This dissertation only examines the tip of the iceberg.

CHAPTER 2

Characteristics of Asynchronous SIMD

2.1 Introduction

The most important characteristic of the "Virtual-Time Data-Parallel Machine" is the asynchronous execution of SIMD programs. The Virtual-Time Data-Parallel Machine looks to the user exactly like a SIMD machine, and thus has its ease of programming. However, it executes the instructions asynchronously, and thus has the benefits of the faster execution of a MIMD machine, all at a cost comparable to a SIMD machine. Therefore, the "Asynchronous SIMD Machine" may be a more appropriate name. The purpose of this chapter is to let the reader become familiar with the asynchronous SIMD approach, and understand the interesting run-time behavior of the Asynchronous SIMD Machine. First, we introduce the terminology, and then we derive some fundamental equations that are widely used in the rest of the dissertation.

2.2 Distribution of Processors in Virtual-Time

In this section, we define global virtual-time (GVT), relative virtual-time (`rvt`), the (time-dependent) histogram of virtual-time, and the (time-independent) dispersion of relative virtual-time (the function which describes how processors spread out in relative virtual-time).

Figure 2.1 is a snapshot of a SIMD task graph in execution. There are four processors in the system, and each processor has the same stream of instructions to execute. Recall from Sec. 1.5 that the virtual-time of a processor is the sequence number of the instruction in execution. Processor 0 is at vt 2, processor 1 and 3 are both at vt 3, and processor 2 is at vt 4. Note that the current instruction may be running or blocked. The global virtual-time (GVT) of the system is defined to be the minimum virtual-time of all processors, i.e.,

$$\text{Global Virtual-Time (GVT)} \triangleq \min_{\forall proc.}\{\text{virtual-time}\} \tag{2.1}$$

In this example, the GVT is 2. The relative virtual-time (rvt) of an instruction is defined to be the difference between its virtual-time and the GVT, i.e.,

$$\text{Relative Virtual-Time } (\texttt{rvt}) \triangleq \text{virtual-time} - \text{GVT} \tag{2.2}$$

At this snapshot, the rvt of instruction 2 is 0, and the rvt of instruction 3 is 1, etc. The rvt of a processor refers to the rvt of the current instruction in execution. The histogram of processors at a snapshot is defined to be the function whose abscissa is the virtual-time and ordinate is the number of processors at that virtual-time, i.e.,

$$\text{histogram}(i) \triangleq \text{Number of Processors at } \texttt{vt} = \text{i} \tag{2.3}$$

In other words, the histogram is the frequency count of the virtual-times of the processors. In Fig. 2.1, there is one processor at vt 2, two processors at vt 3, and one processor at vt 4. Therefore, we have

$$\begin{aligned}\text{histogram}[2] &= 1\\ \text{histogram}[3] &= 2\\ \text{histogram}[4] &= 1\end{aligned}$$

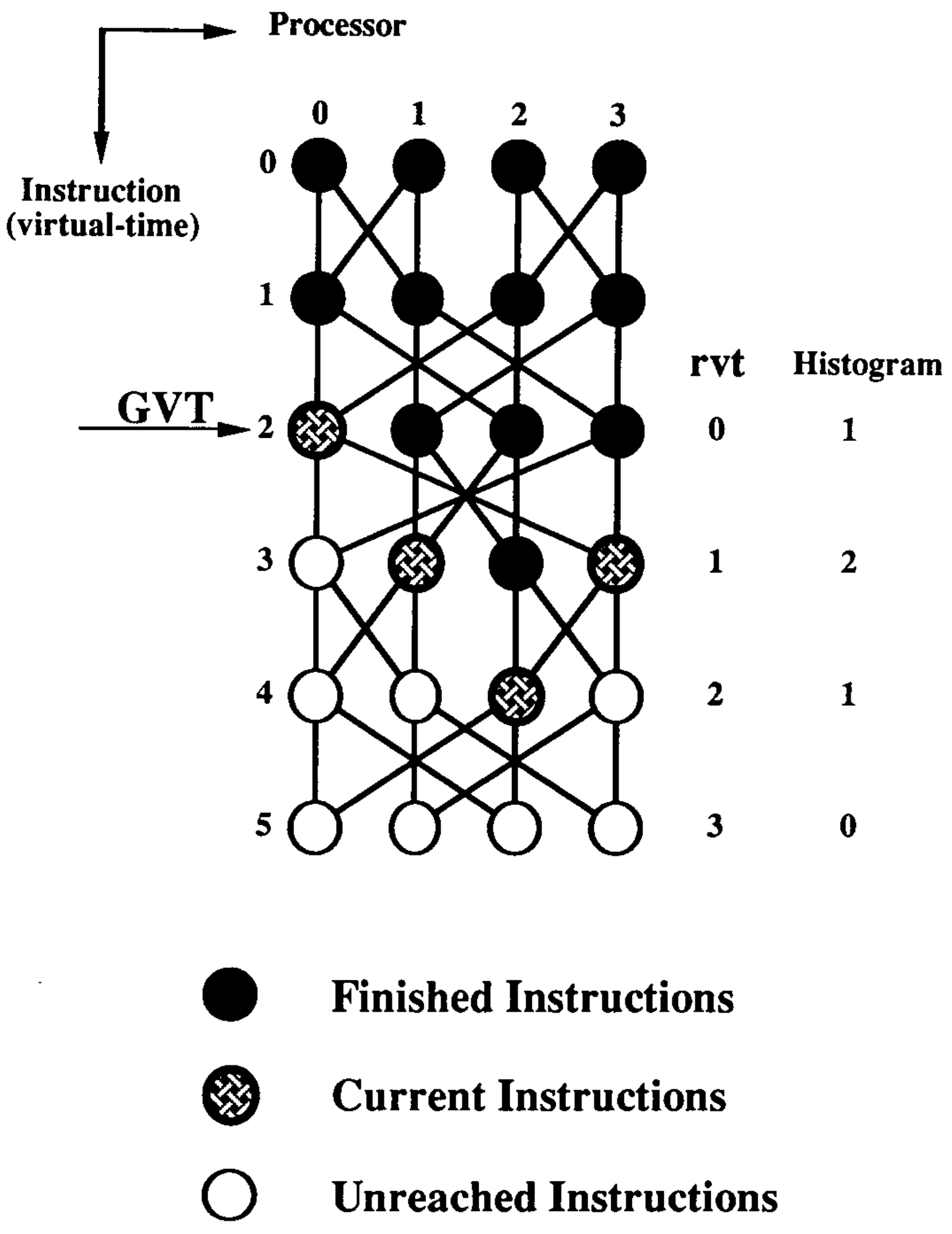


Figure 2.1: A Snapshot of a Task Graph in Execution.

The shape of the histogram represents the distribution of processors in virtual-time during the asynchronous execution of SIMD programs. We can normalize the abscissa of the histogram by shifting the origin to coincide with GVT. Moreover, we scale down the ordinate of the histogram by N, where N is the number of processors in the system. Thus, we define the dispersion function whose abscissa is the `rvt`, and the ordinate is the probability that a processor is at that `rvt`, i.e.,

$$\text{dispersion}(i) \triangleq \text{Probability that a Processor's } \texttt{rvt} = \text{i} \tag{2.4}$$

In Fig. 2.1, there is one processor at `rvt` 0, two at `rvt` 1, and one at `rvt` 2. Therefore, we have

$$\begin{aligned} \text{dispersion}[0] &= \frac{1}{4} \\ \text{dispersion}[1] &= \frac{2}{4} \\ \text{dispersion}[2] &= \frac{1}{4} \end{aligned}$$

Note that GVT is a function of "real-time". As the execution of the program continues, the real-time increases, more instructions are finished, and the GVT advances. Because the `rvt` of a processor is a function of GVT, it is in turn a function of real-time as well.

Let the execution continue. Figure 2.2 shows another snapshot of the SIMD task graph at a later time. Every processor has been executing its instructions at its own pace (under the data dependency constraints, of course). A processor at a small virtual-time (relative to the other processors) earlier may be at a large virtual-time now. For example, processor 0 was at a smaller virtual-time than processor 3 before, but processor 0 is at a larger virtual-time than processor 3 now. The GVT has moved from 2 to 4, and the histogram moves along with

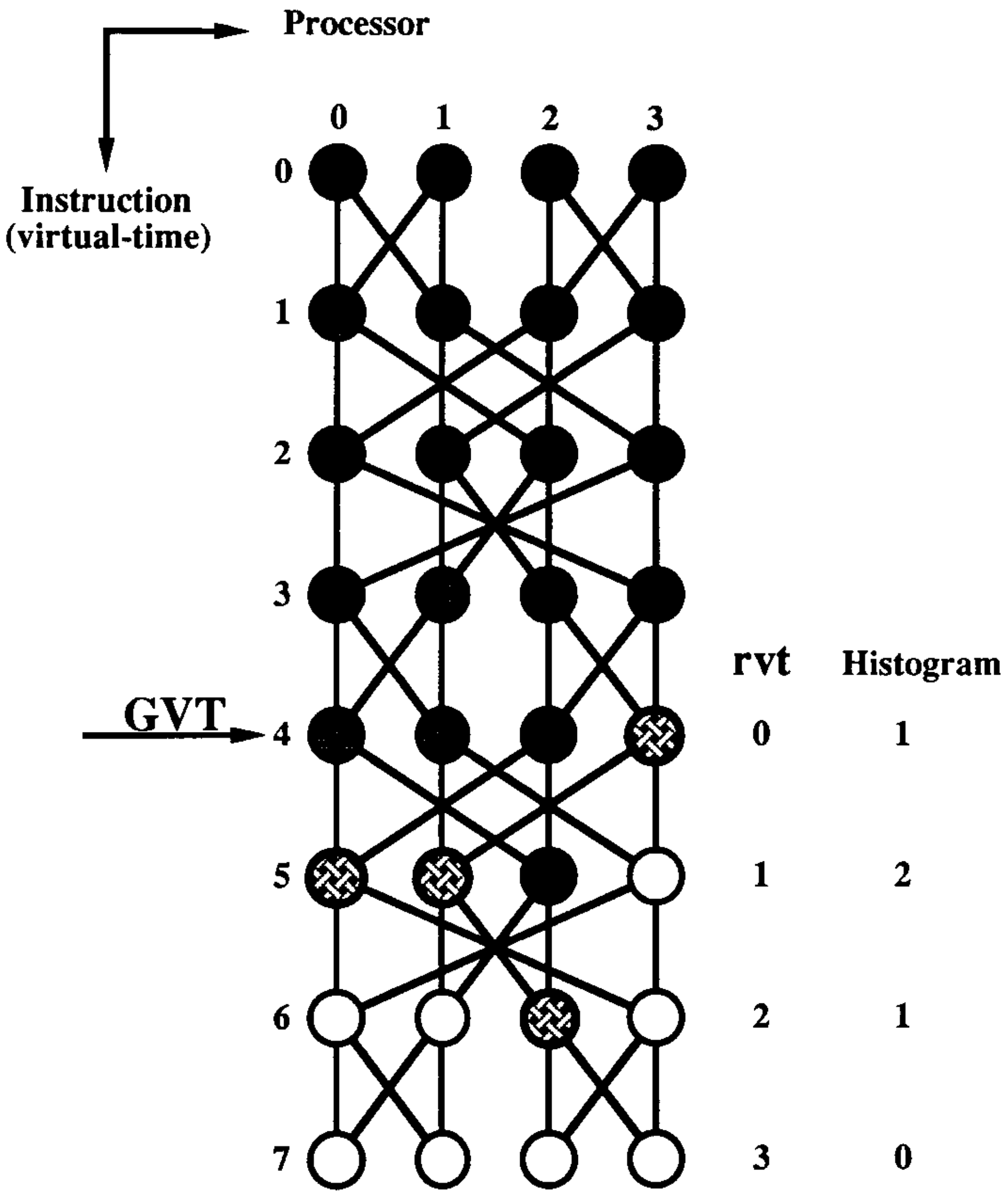


Figure 2.2: Another Snapshot of the Task Graph.

the GVT. A lot of things have changed during the execution of the program. However, under the assumptions in Sec. 1.5.1, the dispersion function (the shape of the histogram) does not change statistically when the number of processors is large. The reason why the dispersion function does not change is as follows: Let the random variable x_i be the `rvt` of processor i. The collection of these random variables can be thought of as a stochastic process. The dispersion function corresponds to some ensemble average of the stochastic process. Since we are only interested in the steady state behavior, the ensemble average of a stationary stochastic process does not change over time. In other words, the dispersion function does not change over real-time. In summary, the histogram moves along the GVT while holding the same shape. It is just like a wave moving across the ocean, with the wave being the shape of the histogram, and the tide being the GVT.

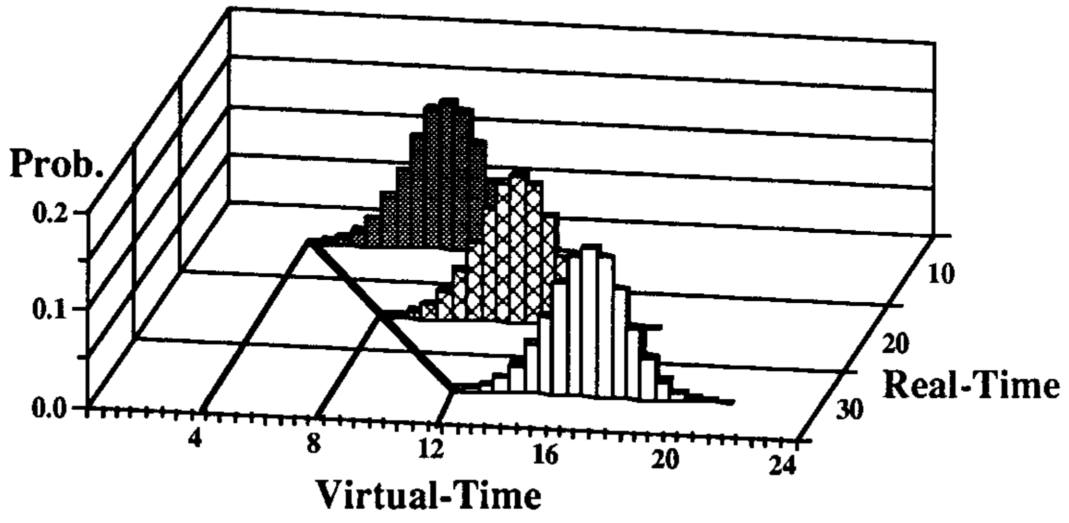
Figure 2.3 shows the distribution of processors in both real-time and virtual-time from simulation. The meaning of "Prob[Virtual-Time | Real-Time]" is explained as follows: This conditional probability means taking a snapshot at some real-time, and then finding the probability that the virtual-time of a (tagged) processor has a particular value.

$$\text{Prob[a processor is at } \texttt{vt}\ i] = \frac{\text{Number of Processors at } \texttt{vt}\ i}{\text{Total Number of Processors}} \tag{2.5}$$

The fact that in steady state, the dispersion function does not change statistically is very important in calculating the progress rate. For example, if the GVT advances 8 instructions in 20 units of time, then on the average, every processor has finished 8 instructions in 20 units of time. We can calculate the progress rate simply by tracking the rate that GVT advances with respect to real-time. In this example, the progress rate (r) is $\frac{8}{20} = 0.4$, which is also the

Prob [Virtual-Time | Real-Time]

(from simulation)



$$\text{Progress Rate } (r) = \frac{\Delta \text{ Global Virtual-Time}}{\Delta \text{ Real-Time}}$$

$$\left(\text{e.g. } r = \frac{12-4}{30-10} = 0.4\right)$$

Figure 2.3: The Distribution of Processors in Both Real-Time and Virtual-Time.

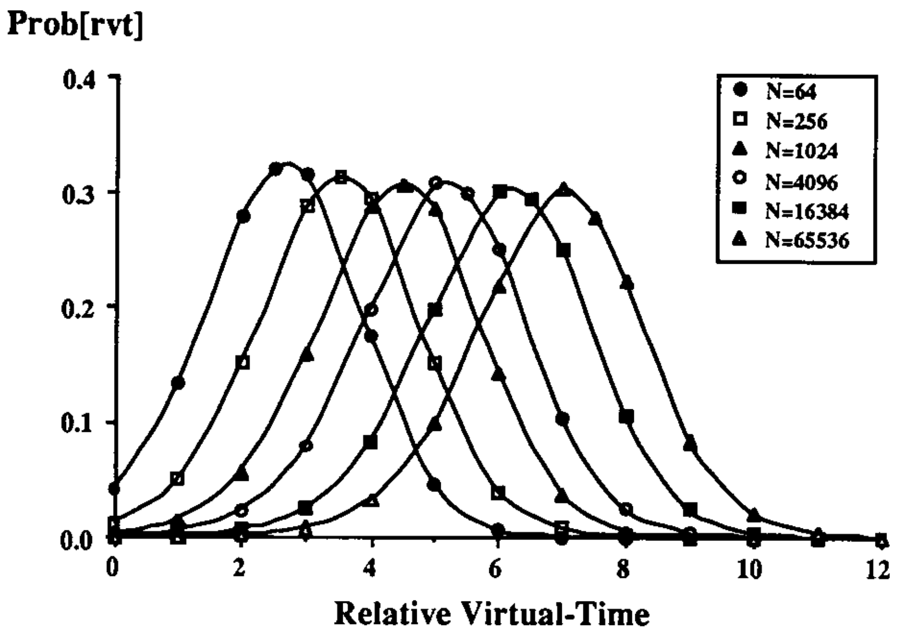


Figure 2.4: The Distribution of Processors in Relative Virtual-Time in Steady State (Simulation).

efficiency of the system.

The dispersion function is a function of the number of processors in the system. Figure 2.4 shows the dispersion functions for various number of processors from simulation. The basic bell shape of these dispersion functions does not change over the number of processors. In other words, even though N may change a lot, the relative position of processors with respect to each other is kept the same for most of the processors. The execution time of an instruction on a processor is mainly determined by the virtual-time difference of its two ancestors (one is this processor itself and the other is the processor which holds the remote operand of the instruction) because it determines whether and when the remote operand will be available. If the environment of every instruction does not change over the number of processors, then the performance of the system must also be insensitive to the number of processors. As a consequence, the progress rate r (i.e., the efficiency) of the system is rather independent of N, as we now show.

We also observe that the center of the dispersion function shifts right (i.e., in increasing `rvt` direction) in proportion to the logarithm of the number of processors. Furthermore, most of the processors are concentrated around the center of the dispersion function. As `rvt` increases, the dispersion function drops to zero very fast once `rvt` passes the center. Figure 2.4 shows that the probability of a processor being beyond `rvt` 12 is almost zero for up to 65536 processors.

2.3 The Number of Processors at GVT

The easiest way to calculate the progress rate is to track the advance of GVT. Processors at GVT are not blocked because their operands are definitely available. Figure 2.5 shows an example of the number of processors at GVT as a function of real-time. The number of processors at GVT decreases when a processor at GVT moves forward after the processor finishes the instruction at GVT. The number of processors at GVT keeps on decreasing. When the last processor at GVT finishes its instruction, it moves forward, and then, GVT moves forward as well. All the processors originally at GVT + 1 before GVT advances are at GVT right after GVT advances, which causes a jump in the number of processors at GVT as shown in Fig. 2.5. The progress rate (r), i.e.,

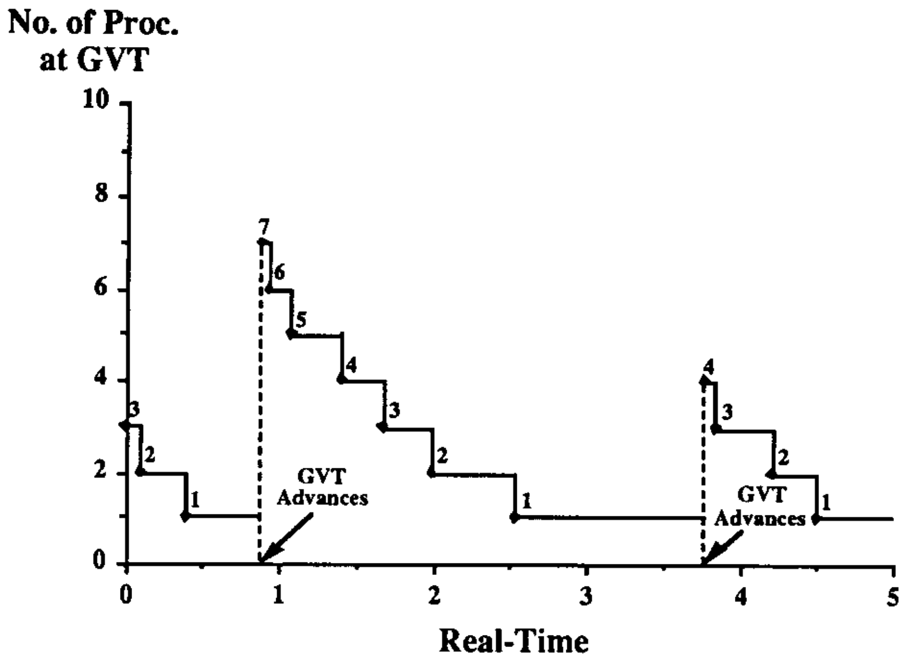


Figure 2.5: The Number of Processors at GVT.

the rate at which GVT advances, is the ratio of the base execution time[1] to the average time interval between two consecutive GVT advances, i.e.,

$$r = \frac{\text{Base Execution Time}}{E[\text{Time Interval between Two Consecutive GVT Advances}]} \tag{2.6}$$

For simplicity, let the base execution time be the unit of time. Let us now derive the expected time interval between two consecutive GVT advances; then the progress rate can be obtained from Eq. (2.6).

Let the random variable g be the number of processors at GVT right after GVT advances, and $g_i \triangleq \text{Prob}[g = i]$. Figure 2.6 shows the state-transition-rate diagram of the number of processors at GVT. In steady state, the rate at which

[1]The base execution time is the average execution time of instructions given all the operands are available, as defined in Sec. 1.5

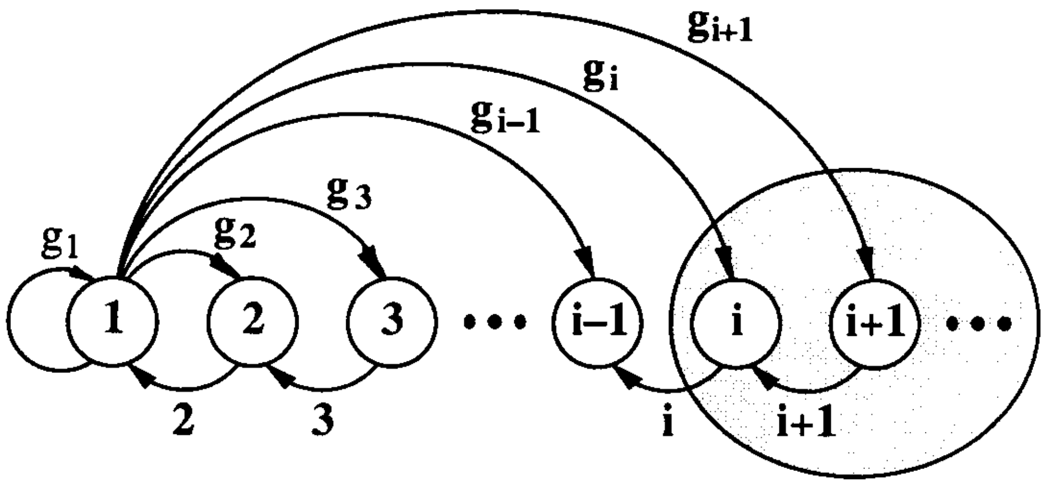


Figure 2.6: The State-Transition-Rate Diagram of the Number of Processors at GVT.

processors move out of GVT is the same as the number of processors at GVT. When GVT advances, g_i is the probability that there are i processors at the new GVT. This is the same state-transition-rate diagram as in the parallel-redundant fault-tolerant model described in [Tri82]. In this parallel-redundant model, there are several independent copies of the same component installed in the system. All the components are actively running but subject to failure at a constant failure rate. As time goes by, some components may fail early but the system will not fail as long as there is at least one component still running. When all the components fail[2], the system fails and the service crew immediately repairs i faulty components with probability g_i. The mean-time-to-failure of this system is equivalent to the mean-time-to-move of GVT.

[2]The service crew does not do regular maintenance to replace faulty components when the system is still running.

Let q be the state variable (i.e., the number of processors at GVT) of the Markov chain, and $q_i \triangleq \text{Prob}[q = i]$. To solve this chain, consider all the states greater than or equal to i as a group. The flow balance equation at steady state says that the rate that flows out of the group is equal to the rate that flows into the group. Therefore, we have

$$q_i * i = q_1 * \sum_{j=i}^{\infty} g_j \qquad i > 1 \tag{2.7}$$

We also have the conservation equation

$$\sum_{i=1}^{\infty} q_i = 1 \tag{2.8}$$

Solving Eq. (2.7) and (2.8), we get

$$q_1 = \frac{1}{\sum_{j=1}^{\infty} g_j * H(j)} \tag{2.9}$$

$$q_i = \frac{\sum_{j=1}^{\infty} g_j}{i * \sum_{j=1}^{\infty} g_j * H(j)} \qquad i > 1 \tag{2.10}$$

where $H(n) = \sum_{k=1}^{n} \frac{1}{k}$ is the well-known Harmonic function. Let G and Q be the expected value of g and q, respectively, i.e.,

$$G \triangleq E[g] = \sum_{i=1}^{\infty} i * g_i \tag{2.11}$$

and

$$Q \triangleq E[q] = \sum_{i=1}^{\infty} i * q_i \tag{2.12}$$

From Eq. (2.9), (2.10), (2.11), and (2.12), we have

$$Q = \frac{G}{\sum_{j=1}^{\infty} g_j * H(j)} \tag{2.13}$$

In order to simplify Eq. (2.9), (2.10) and (2.13), we must find an approximation for the expression $\sum_{j=1}^{\infty} g_j * H(j)$.

Figure 2.7 plots the Harmonic function $H(n)$. We note that the Harmonic

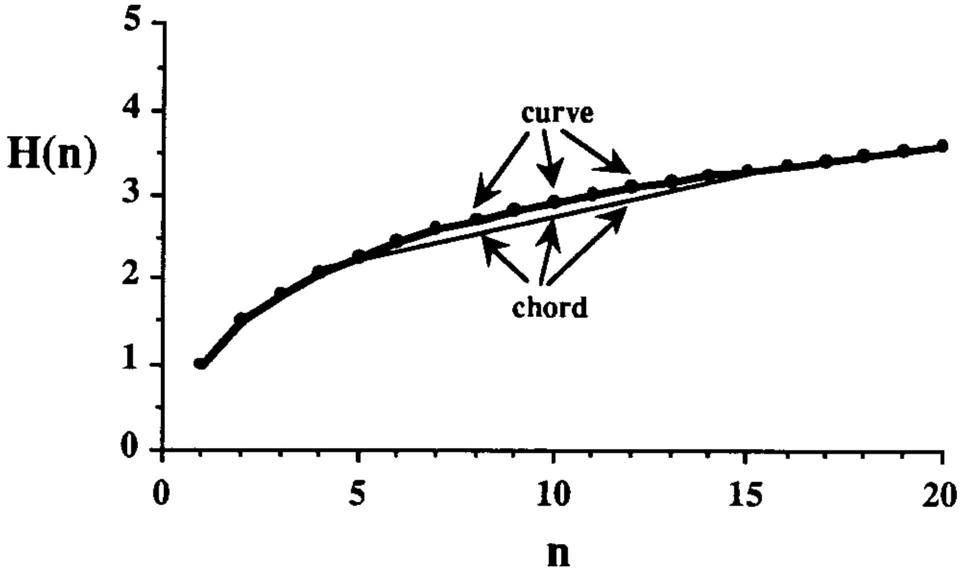


Figure 2.7: The Harmonic Function $H(n)$.

function is a very smooth curve. The difference between the curve and the chord is small, and we can use the curve to approximate the chord. Therefore, we have

$$\sum_{i=1}^{\infty} g_i * H(i) \approx H\left(\sum_{i=1}^{\infty} g_i * i\right) = H(G) \tag{2.14}$$

Substituting Eq. (2.14) into Eq. (2.9), (2.10) and (2.13), we have

$$q_1 \approx \frac{1}{H(G)} \tag{2.15}$$

$$q_i \approx \frac{\sum_{j=i}^{\infty} g_j}{i * H(G)} \qquad i > 1 \tag{2.16}$$

and

$$Q \approx \frac{G}{H(G)} \tag{2.17}$$

We know that the rate of GVT advance is simply q_1. Therefore, we have

$$r = q_1 \approx \frac{1}{H(G)} \tag{2.18}$$

Furthermore, the Harmonic function is a convex (i.e., concave downward) curve. Therefore, we have

$$\sum_{i=1}^{\infty} g_i * H(i) \leq H\left(\sum_{i=1}^{\infty} g_i * i\right) = H(G) \tag{2.19}$$

Following the same procedure of deriving Eq. (2.18), we have

$$r = q_1 \geq \frac{1}{H(G)} \tag{2.20}$$

As long as we have G, i.e., the expected number of processors at GVT right after GVT advances, we can calculate the progress rate from Eq. (2.18). If we only know Q, the expected number of processors at GVT at a *random* time, we have no closed-form solution for r. Fortunately, Table 2.3 gives us some hints for a good approximation. Note that the product of Q and r is close to 1 for small G (say $G \leq 10$), i.e.,

$$Q * r \approx 1 \qquad \text{for small } G \tag{2.21}$$

From Eq. (2.21), we have

$$r \approx \frac{1}{Q} \qquad \text{for small } G \tag{2.22}$$

The fact that G is small is verified by simulation. Simulation results under the assumptions in Sec. 1.5.1 for a large number of processors show that G is approximately 7, which is in the small number range and so validates the

G	$r \approx \frac{1}{H(G)}$	$Q \approx \frac{G}{H(G)}$	$Q * r \approx \frac{G}{H(G)^2}$
1	1.000	1.000	1.000
2	0.667	1.333	0.889
3	0.545	1.636	0.893
4	0.480	1.920	0.922
5	0.438	2.190	0.959
6	0.408	2.449	1.000
7	**0.386**	**2.700**	**1.041**
8	0.368	2.943	1.083
9	0.353	3.181	1.125
10	0.341	3.414	1.166

Table 2.1: Values of $Q * r$ for Small G.

assumption of the approximation. The row $G = 7$ of Table 2.3 is highlighted, and the approximation $Q * r \approx 1$ is very accurate at $G = 7$. The reason why we expect a small G is as follows: From Eq. (2.18), we know that r decreases as G increases. The goal of the machine is to achieve constant efficiency when the number of processors is large (i.e., linear speed-up). Since r is equal to the efficiency, which is approaching a constant limit for a large number of processors (see Fig. 1.7), G must be very insensitive to the number of processors. In addition, for a good architecture the efficiency *must not* be small, and in other words, G cannot be large. We have not yet proved that our architecture achieves the "reasonably good" constant efficiency. Nevertheless, if it *does* (which is demonstrated in Chapter 4), then G must be small and Eq. (2.22) is valid.

2.4 Conclusions

Because of the asynchronous execution, processors are distributed at different virtual-times. The dynamic behavior of the asynchronous SIMD machine can be found by examining the dispersion function and the number of processors at GVT. The dispersion function describes how processors spread out in rvt, which is a critical parameter for the hardware support (discussed in Chapter 3). The number of processors at GVT tracks the advance of GVT, which in turn determines the progress rate (r) of instruction execution. Equations (2.18), (2.20), and (2.22), which we derived in this section, are widely used in Chapter 4 to calculate the progress rate (r) of the system. With the foundation of this section, we are now ready for more advanced topics on the asynchronous execution of SIMD programs.

CHAPTER 3

Hardware Support for Asynchronous Execution

3.1 Introduction

A new problem introduced by the asynchronous model is that previous values of a processor's variables may be needed by another processor in the future. During asynchronous execution, processors are allowed to advance at their own pace (though subject to data-dependency constraints), and therefore, may be spread out at different virtual-times. When a slower processor (i.e., a processor with a smaller virtual-time) requests a remote operand from a faster processor (i.e., a processor with a larger virtual-time), the requested value has actually been generated in a previous instruction on the faster processor. The *time* of the request is *current* to the slower processor but *previous* to the faster processor. From the faster processor's point of view, other processors may need to access previous values of its variables. As a result, the faster processor has to save those previous values.

Suppose a processor (the faster processor) currently at **vt** 10 wants to update its local variable X, and *later on*, another processor (the slower processor) at **vt** 7 needs to read the value of variable X on the faster processor. The faster processor cannot simply overwrite the variable X with the new value *now* (i.e., at **vt** 10) because this processor has no idea whether or not previous values of

X will be requested by other slower processors *later on*. *"Previous values are possibly needed in the future"* is the new problem of the asynchronous model.

In the asynchronous model, a memory request is a time-stamped request. The time-stamp of a memory request is the virtual-time of the instruction which issues the memory request. A remote memory request (i.e., request for the value of another processor's variable) is represented by the tuple "<sender, receiver, address, virtual-time>" where the sender, receiver, address, and virtual-time are the processor-id of the sender, the processor-id of the receiver, the address of the requested variable, and the virtual-time of the sender, respectively. At the receiving processor, the "receiver" field is stripped off because it is no longer useful. The "sender" field is not needed to pinpoint the requested value and serves only as the return address for the reply message of the memory request. Therefore, as far as the memory access is concerned, a memory request is represented by the "<address, virtual-time>"–pair. A local memory request (i.e., request for the value of a processor's own variable) uses the current program counter of this processor as the time-stamp of the memory request. The memory system of a processor does not distinguish local and remote memory requests.

In the synchronous model, time-stamps on memory requests are not useful because all processors are executing the same instruction and therefore, all time-stamps are the same, which is equal to the program counter of the front-end processor. For a time-stamped memory request, the concatenation of the address and virtual-time can be treated as a long-form address of the memory request. From this point of view, the interface between the processor and the memory is the same for asynchronous and synchronous execution. The only difference is that the address appears to be longer for asynchronous execution.

In general, a processor may receive requests from other processors at any virtual-time greater than or equal to the global virtual-time (GVT), where GVT is equal to the minimum program counter among all processors. According to the definition of GVT, no processor will be executing instructions earlier than GVT and no request will have a time-stamp less than GVT. If the time-stamp of an incoming request is larger than the program counter of the processor, the request is asking for a future value. The processor does not have the future value now, but the processor will generate it later on. The request cannot be granted now and the processor which issues the request will be blocked until the requested value is available. If the time-stamp is equal to the program counter of the processor, the request is asking for a current value, which is available now. If the time-stamp is less than the program counter, the request is asking for a previous value. Therefore, every processor has to maintain a *memory history*. The memory history stores previous values of variables during the virtual-time period from the current program counter back to the GVT. Anything before the GVT will no longer be needed, and therefore can be purged. The memory history must be able to provide previous values of variables back to GVT on demand.

For practical reasons, there is a physical limit on the size (i.e., the length) of the memory history. By setting the size of the memory history to K, we mean that if a processor goes so fast that its program counter is K instructions ahead of the GVT (i.e., it is K instructions faster than the slowest processor), the fast processor has to be temporarily suspended because it has used up all the space in its memory history. As long as K is reasonably large, the probability that a processor gets suspended because of insufficient memory history is tiny, and the performance of the machine hardly degrades due to the limited size of the memory history.

Memory history is a new feature in the memory system. It is so important and it is so frequently used that it deserves special hardware support. The hardware support should achieve the following goals:

Fast: Retrieving information from the memory history is as fast as an ordinary memory reference.

Inexpensive: The cost of the hardware support in terms of the number of transistors is small compared to the cost of the whole system.

Transparent: The memory history is transparent to main memory in the sense that main memory still uses traditional random access memory without any modification.

The reasons why these goals are necessary are as follows: Whenever we add hardware support to a system to enhance its functionality, the extra hardware may slow down the speed of some basic operations, increase the cost of the whole system, and require the redesign of the rest of the system in order to incorporate the new hardware. We want the hardware support simply to increase the functionality without slowing down any operation. We also want the hardware support to be so inexpensive that the extra functionality costs little compared to the whole system. Last, but not least, we want to maintain the new system's compatibility with the previous design. The purpose of hardware support in general is to add an additional feature to the original design. Even though adding more hardware may not cost much more to manufacture, redesigning the rest of the system to incorporate the additional hardware may be very expensive. We want to minimize the redesign due to the addition of the hardware support. This may not be an important issue for a brand new design but may be the most important issue in keeping a current design alive.

The hardware support we are looking for has to satisfy all three goals above – fast, inexpensive, and transparent.

3.2 Memory History

Memory history can solve the *"no overwrite"* problem of asynchronous execution of SIMD programs. In this section, we discuss how to implement the memory history cost-effectively.

A simple way to implement the memory history is full backup. Full backup stores one copy of each variable of a processor at every virtual-time from the GVT up to the program counter of the processor. On retrieving information, the memory history takes the virtual-time of the request as the index to the memory module at this virtual-time. With full backup, the memory history does not slow down memory reference, but the drawback is the tremendous cost. Full backup is prohibitively expensive to implement because it takes K times the main memory to implement a memory history of size K.

An intelligent alternate solution to implement the memory history is incremental backup. Incremental backup stores only the updates, i.e., the new values of the updated variables, instead of complete copies of all variables. Since one instruction modifies at most one variable, incremental backup only needs to store K updates to implement a memory history of size K. From the K updates and the main memory[1], we can calculate previous values of any variable

[1]For incremental backup, data in the main memory are the outdated values of the variables.

up to K steps backward. It takes more work for incremental backup to retrieve previous values, but it is extremely space-efficient. Therefore, we choose to implement the memory history with incremental backup.

3.2.1 Algorithm for Incremental Backup

Figure 3.1 explains the algorithm for incremental backup. The update queue of a processor stores all the updates from the GVT up to the program counter of the processor. These updates are collectively called the *outstanding updates*. Each update consists of three fields – address, virtual-time, and data. Every memory request contains two fields – address and virtual-time.

A formal description of the algorithm is as follows: The virtual-time of a memory request is compared with the program counter of the processor. If the program counter is smaller, then the requested value is not available and the sender of the request gets blocked. Otherwise the memory request is asking for a previous or current value which is in the memory history. This value can be obtained from the outstanding updates and the main memory. We conduct an associative search for *hits* in the update queue, where a hit is any outstanding update of the same variable at a virtual-time earlier than the virtual-time of the request. If no hit can be found, then the variable has not been modified since GVT and the value in main memory is the requested value. Otherwise, there is at least one hit; the data in the latest hit is the requested value because the latest update overwrites the earlier ones.

Let us illustrate the algorithm through examples. In the first example, the

Nevertheless, some of the outdated data are still *up-to-date* if they have not been modified since then. The latest values are not stored in the main memory, and therefore, have to be calculated on demand.

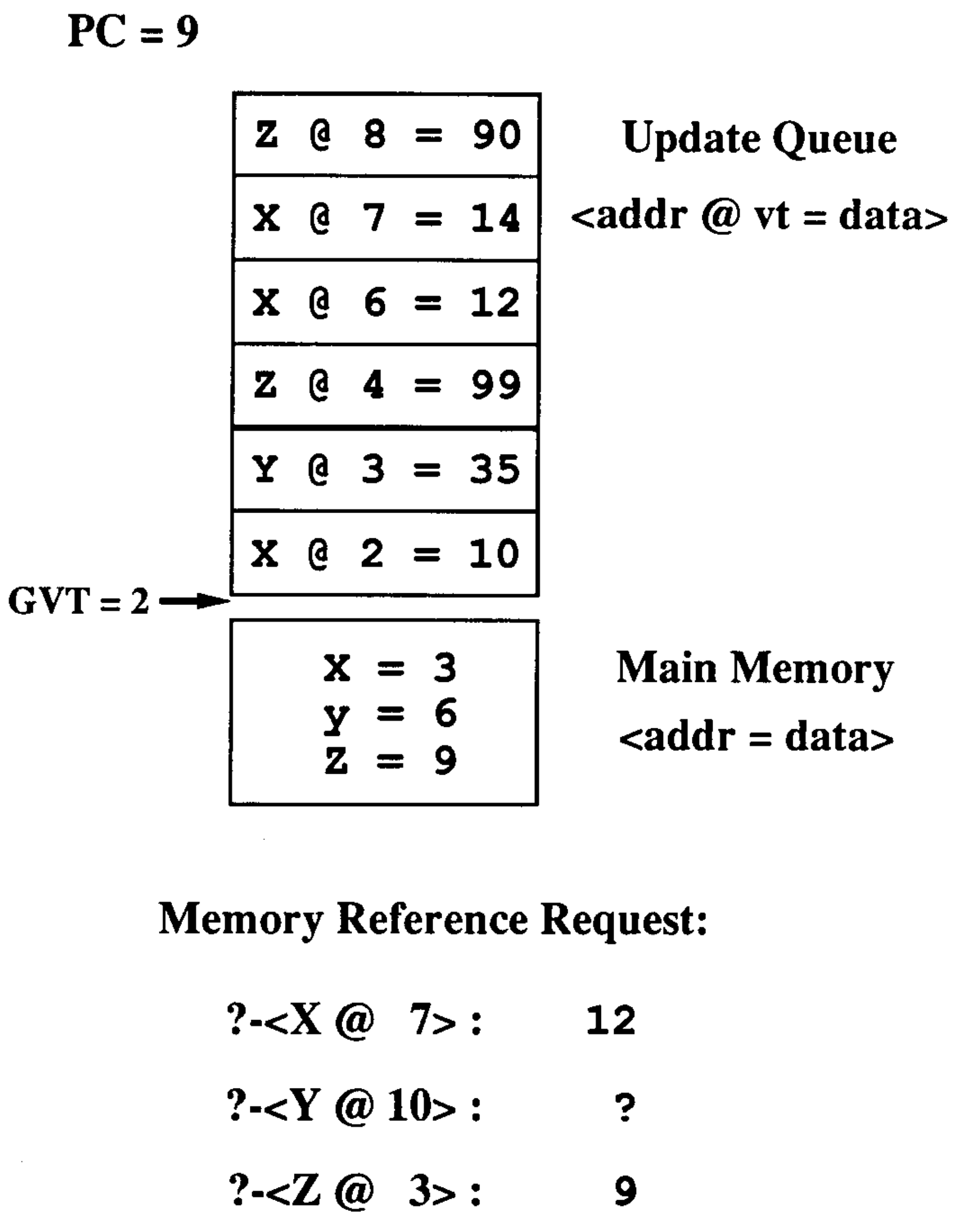


Figure 3.1: Examples for the Incremental Backup Algorithm.

memory request asks for variable X for the instruction at vt 7. According to the SIMD semantics, what it really asks is the value of X at the end of instruction 6. In this case, any update to X before vt 7 is allowed to change the requested value. Therefore, the first step is to conduct an associative search for the updates which match the address and have *smaller* virtual-times (i.e., hits). There are two hits in this example. The requested value is the data in the latest hit because the latest update overwrites all previous updates. So, the answer is 12 for the first example. In the second example, variable Y at vt 10 is requested. Because the program counter (pc) is less than 10, the processor has no idea whether there will be any new update on Y before vt 10 or not. Therefore, the answer to the request is "unknown" for the second example. An unknown reply causes the processor which issues the request to be blocked. In the last example, variable Z at vt 3 is requested. There is no hit in the update queue, which means the value of variable Z has not been changed from the GVT to vt 3. Therefore, the value in the main memory is the requested value. So, the answer is 9 for the last example. These three examples cover all three cases of the operation of the memory history.

3.2.2 Hardware for Incremental Backup

Figure 3.2 shows a hardware implementation of the incremental backup. In the middle of Fig. 3.2, the four boxes represent the FIFO queue which stores the outstanding updates. New updates come in from the top of the FIFO queue. Every new update will have a larger virtual-time than the preceding updates so that the virtual-time in the FIFO queue is monotonically increasing. Old updates are purged from the bottom of the FIFO queue when the GVT

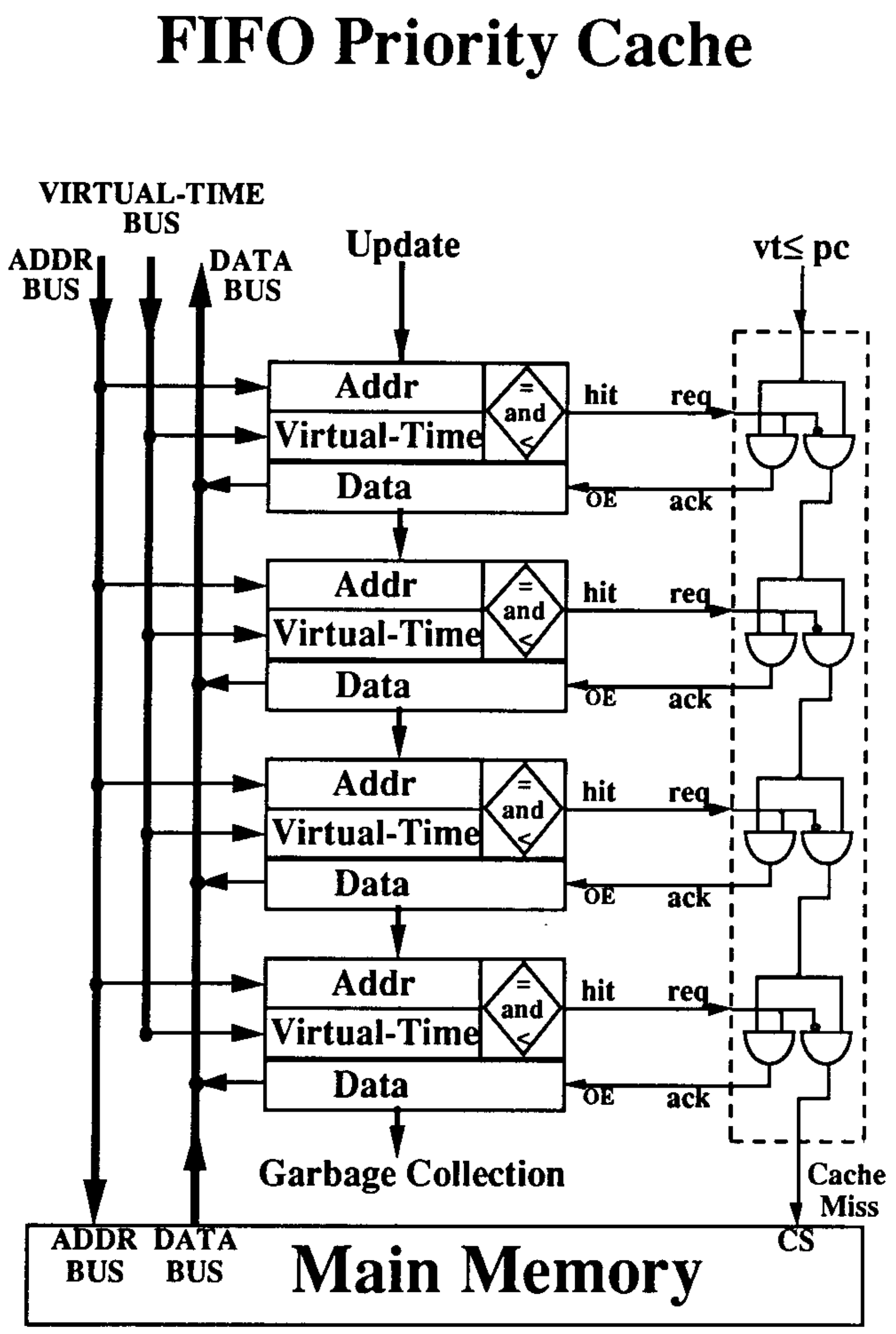


Figure 3.2: The Hardware Support for Incremental Backup.

exceeds the virtual-time of the old updates. The data in the purged updates are written to the main memory and these updates are then discarded. For every memory request, if its virtual-time is larger than the program counter of the processor, the requested value is not available. Otherwise, the FIFO queue serves as a cache memory, which associatively searches for hits, where a "hit" is an outstanding update with the same address and smaller virtual-time. Therefore, the FIFO queue is in fact a FIFO cache. If there is more than one hit, then the daisy-chain (i.e., the AND-gates on the right side of Fig. 3.2) enforces the priority which selects[2] the latest hit. If there is no hit, it is called a cache miss. A cache miss causes the memory request to be passed to the main memory[3] because the requested value is stored in the main memory.

The characteristics of the hardware support are:

FIFO: FIFO queue to store the outstanding updates,

Cache: cache memory to search hits associatively, and

Priority: daisy chain to select the latest hit.

Therefore, we name the hardware support "FIFO priority cache".

Past research on the efficient implementation of state-saving is mainly concerned with the support of roll-back for optimistic distributed simulation. Richard Fujimoto, et al have been actively working on special purpose hardware for Time Warp [Fuj88], the design of Virtual Time Machine [Fuj89b], and the space-time memory [Gho91]. Both the hardware and operating system support for roll-back are equally applicable to the Virtual-Time Data-Parallel Machine.

[2]Selecting a hit means setting the output-enable (OE) bit of the hit to 1.

[3]The time-stamp of the memory request is stripped off when the memory request is passed to the main memory.

3.3 The Size of the Memory History

The size of the memory history (K) is an important design parameter. When K is too small, processors may get blocked frequently because of insufficient memory history space. When K is too large, the efficiency improvement has already reached the point of diminishing returns but the hardware overhead is still increasing with K. Therefore, there exists an optimal value for the size of the memory history. When the optimal size of the memory history is unknown or hard to find, it is better to choose a larger memory history. The reasons are as follows: First, the overhead of the memory history is very small in terms of the number of transistors compared with the cost of the whole system. Even if we were to choose twice the optimal size, the cost is still relatively small. In general, there is no harm to have many inexpensive spare parts *just in case*. Moreover, the *effective* size of the memory history is smaller than the actual size because of the latency in calculating the GVT as explained below. The GVT is never up-to-date because of the inevitable propagation delays in collecting global information. However, an outdated GVT is still *correct* in the sense that it does not cause any incorrect results to be generated. If there is a genie who knows the *current* GVT, then the estimated GVT on every processor[4] would always be less than that. The GVT lag of a processor is defined to be the difference of the current GVT and the estimated GVT on this processor. The GVT lag reduces the effective size of the memory history. It is better to leave some safety room (i.e., a larger memory history) to compensate for the GVT lag. In addition, the FIFO cache is an effective cache design itself, especially when the cache size is small. Even though the values before GVT should be purged

[4]Even though there is a unique *current* GVT, the *estimated* GVT on every processor can be different.

to the main memory, it may still be better to store them in the memory history because these recently[5] computed values are very likely to be needed in the near future based on the likelihood of temporal locality [Smi82] in the memory request sequence. If these values are kept in the memory history, reference to them will generate *hits* instead of *misses*. The value of a hit is immediately available in the cache while the value of a miss will not be available until going through the prolonged main memory access. In summary, the *side-effect* of a large memory history may well in itself justify the extra cost. Therefore, the purpose of analyzing the size of the memory history is not to calculate the optimal size. Instead, an upper bound on the memory history size is derived. The upper bound tells us how large a memory history is *sufficient* to eliminate the out-of-memory-history blocking.

Asynchronous execution allows processors to execute instructions at their own pace. That is the reason why the distribution of the virtual-times of the processors may spread out widely. However, because of the data-dependency constraints, processors are brought closer to one another. Another reason to bring processors close together in virtual-time is the limited size of the memory history. When a processor runs so fast that it uses up all the space in its memory history, it is suspended temporarily to allow the other processors to catch up. This in turn brings processors close together. If the the data-dependency is strong, very few processors will be able to run much faster than the others. Almost all the faster processors will be blocked because of the data-dependency constraints before they can use up the space in their memory history. On the other hand, if there is weak data-dependency, processors have a lot of freedom

[5]Even though the newly purged updates are older than those in the FIFO queue, they are still *recent* from the point of view of temporal locality.

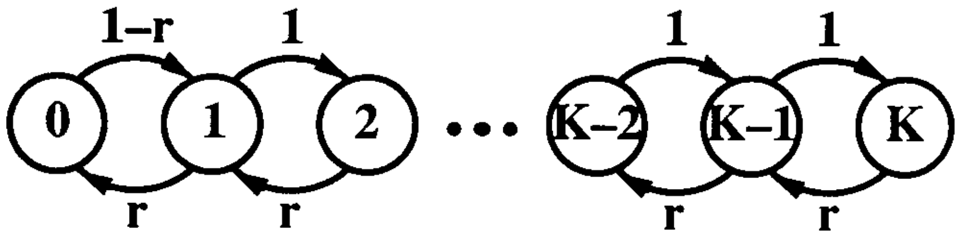


Figure 3.3: The State-Transition-Rate Diagram of Relative Virtual-Time for the Decoupled Model.

to go ahead and the faster processors are more likely to be blocked because of the limited size of the memory history. Therefore, we introduce the "decoupled" model, in which there is no data-dependency constraint at all. Every processor is allowed to execute the next instruction as long as it is within K instructions of the slowest processors, where K is the size of the memory history. This model gives an upper-bound on the loss of performance due to the limited size of memory history.

Figure 3.3 shows the state-transition-rate diagram of the relative virtual-time of a (tagged) processor in steady state for the decoupled model. Recall from Sec. 2.2 that the relative virtual-time of a processor is defined as the difference between its virtual-time and the GVT. Processors at state K will be blocked because of insufficient memory history. Processors at state 1 to $K-1$ are free to go ahead so that the state transition rate from these states to their subsequent states is 1[6]. GVT advances at the progress rate r in steady state, and the advance of the GVT causes the relative virtual-time of every processor

[6]Recall the assumption in Sec. 1.5.1 that the execution time of instructions is exponentially distributed with mean 1.

to decrease by one. Therefore, the state-transition-rate from one state back to its preceding state is r for states 1 to K. State 0 (i.e., the processor is at GVT) is special. When a processor at GVT finishes its instruction, its relative virtual-time may *not* increase to 1 because it may be the only processor left at GVT. When the last processor at GVT finishes its instruction, its virtual-time advances and so does GVT. As a consequence, the relative virtual-time of this processor is still 0. Therefore, the state-transition-rate from state 0 to state 1 is only $1 - r$.

Let the random variable p be the relative virtual-time of a (tagged) processor in steady state, and $p_i \triangleq \text{Prob}[p = i]$. Setting up the flow balance equations from the state diagram, we have

$$p_i * r = \begin{cases} p_0 * (1 - r) & i = 1 \\ \\ p_{i-1} & 1 < i \leq K \end{cases} \tag{3.1}$$

From Eq. (2.22) in Sec. 2.3, we have

$$r \approx \frac{1}{Q} = \frac{1}{p_0 * N} \qquad \text{for small } Q \tag{3.2}$$

where Q is the expected number of processors at GVT, and N is the total number of processors in the system. If K is sufficiently large such that the efficiency of the system is fairly good, then Q must be a small number, say less than 10 (refer to Sec. 2.3). Rearranging Eq. (3.2), we have

$$p_0 \approx \frac{1}{r * N} \tag{3.3}$$

In addition, we also have the conservation equation

$$\sum_{i=0}^{K} p_i = 1 \tag{3.4}$$

Solving Eq. (3.1), (3.3) and (3.4), we have

$$p_i \approx \begin{cases} r^K & i = 0 \\ \\ (1-r) * r^{K-i} & 1 \leq i \leq K \end{cases} \tag{3.5}$$

and

$$r \approx \sqrt[K+1]{\frac{1}{N}} \tag{3.6}$$

The performance loss due to the limited size of the memory history manifests itself as the blocking probability at relative virtual-time K (i.e., in state K). From Eq. (3.5), we have

$$p_K = 1 - r \tag{3.7}$$

Substituting Eq. (3.6) into Eq. (3.7), we have

$$p_K \approx 1 - \sqrt[K+1]{\frac{1}{N}} \tag{3.8}$$

which is the maximum performance loss due to the limited size of memory history; see Fig. 3.4. When the number of processors (N) increases, we must increase the size of the memory history (K) so that the performance loss (p_K) due to the limited size of the memory history is kept constant. From Eq. (3.8), we find that K must increase as follows in order to maintain this constant loss of p_K:

$$K = \frac{\log N}{-\log(1-p_K)} - 1 \tag{3.9}$$

The bold line in Fig. 3.4 corresponds to 30% performance loss. From Eq. (3.9), we note that in order to keep a constant percentage of performance loss, the size of the memory history should be on the order of ($\log N$).

The decoupled-model gives an upper bound on the performance loss due to insufficient memory history. Therefore, Eq. (3.9) gives an upper bound on the

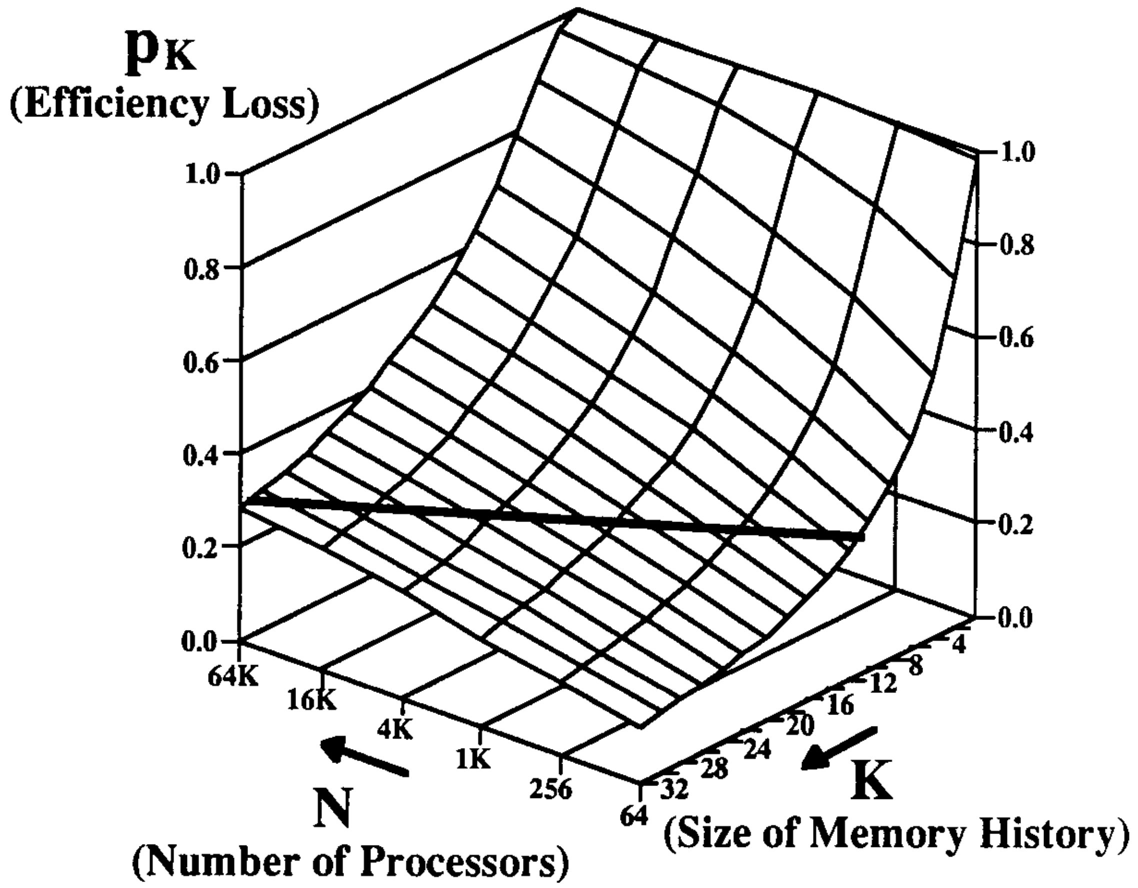


Figure 3.4: Maximum Efficiency Loss Due to the Limited Size of Memory History.

choice of K (for given N and p_K). With data-dependency, a smaller value of K will be sufficient to maintain the same out-of-memory-history blocking.

3.4 Conclusions

This section summarizes some important issues about the memory history. The architecture of the hardware support is a FIFO priority cache, which implements the incremental backup algorithm. Outstanding updates are stored in a FIFO queue and searched associatively for the latest hit. The garbage collection of the updates is to purge to the main memory those updates whose virtual-times are smaller than GVT. When a processor goes so fast that it runs out of memory history space, it is blocked. The overhead of the memory history is about the same as a K-word cache, where K is the size of the memory history. K should be chosen in proportion to $(\log N)$, where N is the total number of processors in the system.

The "FIFO Priority Cache" satisfies all the goals of the hardware support – fast, inexpensive, and transparent. The cache memory will not slow down the memory reference. On the contrary, the cache memory accelerates memory references. The small cache does not cost too many transistors to implement and is a relatively small increase to the total cost of the system. Furthermore, the cache memory is transparent to the main memory and is compatible with the rest of the system. There is no need to redesign any part of the system to incorporate this hardware support.

CHAPTER 4

Performance Analysis of the Progress Rate

4.1 Introduction

The progress rate (r) is the most important performance measure of the Virtual-Time Data-Parallel Machine. From Eq. (1.5), we know that the progress rate is equivalent to the efficiency of the system, from Eq. (1.2), we can calculate the speed-up of the system from the progress rate, and from Eqs. (2.18) and (2.22), we can derive the expected number of processors at GVT from the progress rate. In short, the progress rate is the key to exploring other aspects of the Virtual-Time Data-Parallel Machine.

In this chapter, we evaluate the progress rate r for sufficiently large memory history (e.g., $K = \infty$). Because the exact model is hard to solve (the state-space is too large), we develop several slightly modified models to analyze r. First, we consider the *"non-persistent"* model and *"cold-start"* model, which give an upper bound and lower bound on r, respectively. We then study the *"roll-back"* model, which gives an approximation to r as a function of the number of processors in the system. Finally, we evaluate the *"infinite-processor"* model, which gives the limiting value of r for an infinite number of processors.

Related research mainly addresses the performance analysis of the Time-Warp synchronization mechanism for optimistic parallel discrete event simulation [Fel90] [Fel91] [Gup91] [Nic91] [Sam85], and partial synchronization in parallel processing systems [Cha91]. Our analysis is customized for the Virtual-Time Data-Parallel Machine under the assumptions in Sec. 1.5.1. Nevertheless, the techniques used in our approach can be extended to more general types of synchronization in parallel computations.

4.2 The Non-Persistent Model

The non-persistent model gives an upper bound on the progress rate because the way it handles unavailable remote operands is different from the real system. If the remote operand of an instruction is not currently available, the *correct* execution of the instruction is to wait for *this* operand *persistently* until it becomes available; therefore, we call the correct execution the *"persistent"* model. To do this, in addition to the program counter of a processor, the state of the processor must also contain the remote operand of the current instruction, which makes the persistent model hard to analyze. On the other hand, it is assumed in the *non-persistent* model that the processor will *not* wait for this remote operand persistently. If the reply of the memory request for a remote operand says that this remote operand is not available, then in the next cycle,the instruction will randomly choose *another* remote operand, and send out a new memory request immediately. This is certainly an *incorrect* implementation of the machine, but it is a *simple* model to analyze.

From Sec. 2.2, we note that the relative virtual-time (`rvt`) of a processor has a steady-state distribution. Therefore, we choose the `rvt` as the state variable of a processor.

Let the random variable p be the `rvt` of a (tagged) processor and $p_i \triangleq \text{Prob}[p = i$

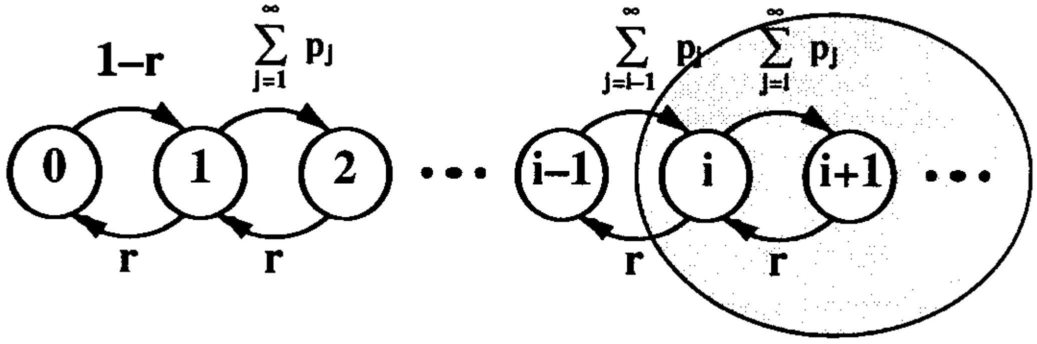

1−r
$\sum_{j=1}^{\infty} p_j$
$\sum_{j=i-1}^{\infty} p_j$
$\sum_{j=i}^{\infty} p_j$
0
1
2
i−1
i
i+1
r
r
r
r

Figure 4.1: The State-Transition-Rate Diagram of Relative Virtual-Time for the Non-Persistent Model.

Figure 4.1 shows the state-transition-rate diagram of the rvt of the processor in steady state for the non-persistent model. We assume the size of the memory history is infinite. The state transition rates between adjacent states are similar to those in the decoupled model in Sec. 3.3. Global virtual-time (GVT) advances at a rate r, and therefore, the state transition rate from one state to its preceding state is r for all states except state 0. The advance of a processor at GVT (i.e., state 0) will cause GVT to advance along with it simultaneously if and only if it is the only processor in state 0. In this case, the rvt of that processor remains unchanged because both the virtual-time of the processor and the GVT advance together. In other words, these two transitions cancel out each other as if, as far as the rvt is concerned, no transition has happened at all. This phenomenon reduces the state transition rate from state 0 to state 1 to $1 - r$, which is r less than the rate at which a processor at GVT finishes its instruction[1].

[1]The rate at which a processor at GVT finishes its instruction is 1 because its operands are definitely available.

When the tagged processor is at state i where $i \geq 1$, its remote operand is available only if the processor which holds the remote operand is at a state j where $j \geq i$ (i.e., that processor is currently ahead of the tagged processor). The probability of the above condition is $\sum_{j=i}^{\infty} p_j$, and this is the state transition rate from state i to state $i+1$. Setting up the flow balance equations, we have

$$p_i * r = \begin{cases} p_0 * (1-r) & i = 1 \\ p_{i-1} * \sum_{j=i-1}^{\infty} p_j & i > 1 \end{cases} \tag{4.1}$$

From this equation, together with the conservation equation

$$\sum_{i=0}^{\infty} p_i = 1 \tag{4.2}$$

and Eq. (3.3) in Sec. 3.3, which we repeat here,

$$p_0 \approx \frac{1}{r * N} \tag{4.3}$$

we can approximately solve the p_i's numerically. Figure 4.2 shows the distribution of processors in rvt for various value of N in steady state.

This result for the non-persistent model is optimistic compared to that of the persistent model (i.e., the correct model). This is because, in the non-persistent model, a processor does not suffer from "the persistence of bad news". The persistence of bad news refers to the fact that when a bad situation happens, it takes a longer time to clear the bad situation than the time for which a good situation can last. The good news in executing an instruction is that the remote operand of the instruction is immediately available. The processor can execute the current instruction and proceed to the next instruction without delay. Whether the remote operand for the next instruction will be available or

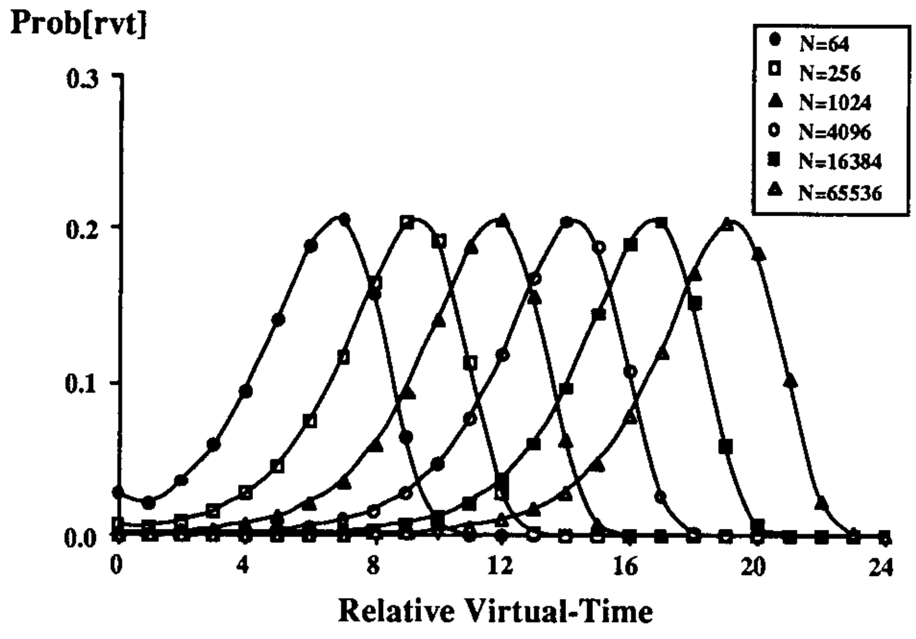


Figure 4.2: The Distribution of Processors (Non-Persistent Model).

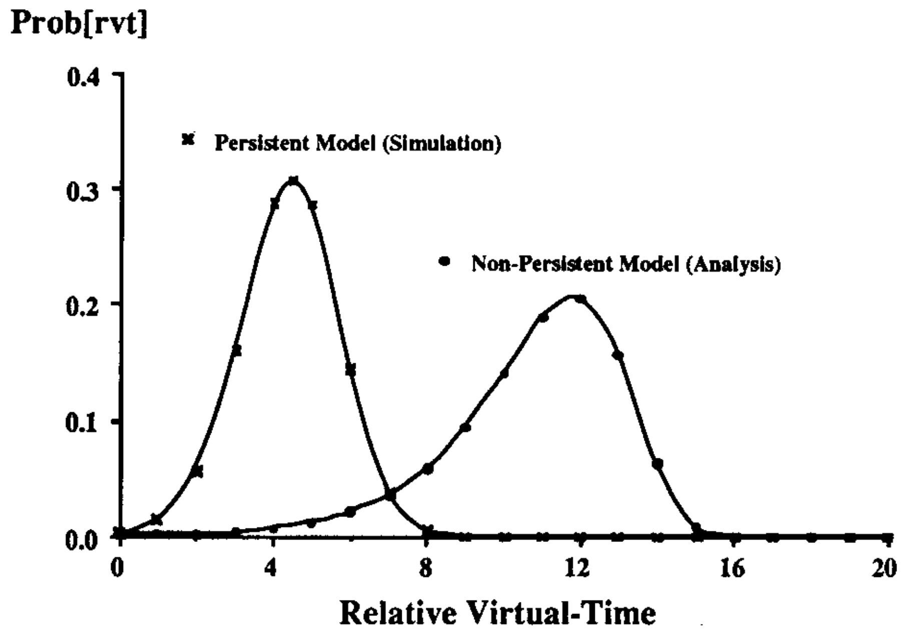


Figure 4.3: The Distribution of Processors (Simulation vs. Non-Persistent Model for $N = 1024$).

not is not biased by the fact that the remote operand of the current instruction is immediately available. That means the good news lasts only for the execution time of *one* instruction. The bad news when executing an instruction is that its remote operand is not available. In correct execution (i.e., the persistent model), the processor is blocked until this remote operand becomes available. Suppose the blocked processor is at `rvt` i and the blocking processor (i.e., the processor which holds the remote operand) is at a `rvt` $j < i$. The difference between i and j is always greater than or equal to one. That means bad news usually lasts longer than the execution time of one instruction. In summary, bad news is expected to last longer than good news.

In the non-persistent model, a processor does not suffer from the persistence of bad news and therefore a better performance is achieved. Figure 4.3 compares the dispersion function for the persistent model from simulation vs. our analysis for the non-persistent model. Both of these curves from have the same basic bell shape, with the one from the non-persistent model wider and flatter. In other words, in the non-persistent model, data dependency constraints are weaker and processors have more freedom to advance at their own pace. That also explains why the performance of the non-persistent model serves as an upper bound on the performance of the persistent model.

4.3 The Cold-Start Model

The cold-start model gives a lower bound on the progress rate. A lower bound on the performance of the system is usually more important than an upper bound because the lower bound *guarantees* the minimum quality of the system. In contrast to the other models which deal with the steady-state behavior of the system, the "cold-start" model emphasizes the behavior of the system at the beginning of the program execution. What interests us most is the number of processors at GVT right after GVT advances. Let G_i be the expected number of processors at GVT right after GVT moves to i, and let G be the limiting value of the sequence, i.e., $G = \lim_{i \to \infty} G_i$. G corresponds to the steady state value of the number of processors at GVT right after GVT advances. Figure 4.4 shows the the first few values of G_i obtained from simulation where $N = 256$. Initially, every processor is at vt 0 and the GVT is 0; thus, $G_0 = N$. When the

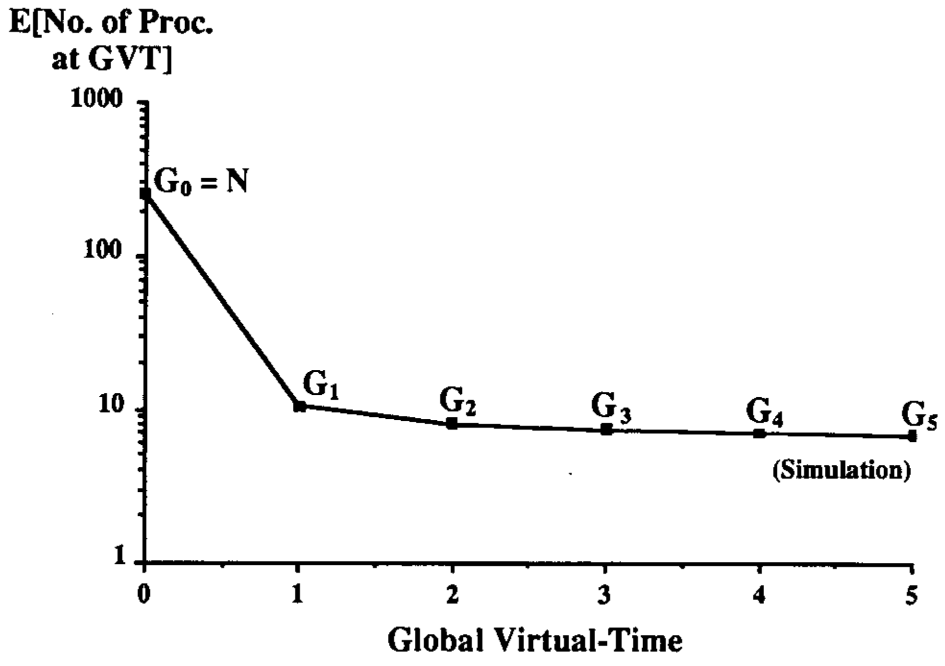


Figure 4.4: The Expected Number of Processors at GVT in Cold-Start.

program starts running, processors begin to spread out in rvt. As GVT advances, the expected number of processors at GVT is monotonically decreasing and gradually reaches the steady state value G. Therefore, we have

$$N = G_0 \geq G_1 \geq G_2 \geq \cdots \geq G_\infty = G \tag{4.4}$$

Together with Eq. (2.20) in the parallel-redundant model, which we repeat here,

$$r \geq \frac{1}{H(G)} \tag{4.5}$$

we have

$$\begin{aligned} r &\geq \frac{1}{H(G)} = \frac{1}{H(G_\infty)} \geq \cdots \geq \frac{1}{H(G_i)} \geq \cdots \\ &\geq \frac{1}{H(G_2)} \geq \boxed{\frac{1}{H(G_1)}} \geq \frac{1}{H(G_0)} = \frac{1}{H(N)} \end{aligned} \tag{4.6}$$

Any of the terms $\frac{1}{H(G_i)}$ for all i can be used as a lower bound on r. For larger i, the lower bound is tighter but harder to calculate. From Fig. 4.4, we see that there is a big drop from G_0 to G_1 but not as significant a difference between G_1 and G_2 and so on. Based on the above considerations, we choose to compute $\frac{1}{H(G_1)}$ as a lower bound on r.

Now let us derive the value of G_1. Let the pid (processor-id) of a processor be the *reverse* order of finishing instruction 0 (i.e., the first instruction). The first processor to finish instruction 0 will have a pid equal to N, the second $(N-1)$, etc. Let a tagged processor's pid be i and the pid of the processor holding its remote operand be j. If $j \geq i$, then the remote operand for instruction 1 will be available by the time processor i (i.e., the processor with pid i) finishes instruction 0. On the other hand, if $j < i$, then processor i must wait for processor j. For example, in Fig. 4.5, processor 6 is executing instruction 1 because processor 4 has finished instruction 0, but processor 4 cannot start executing instruction 1 until processor 2 finishes executing instruction 0. In

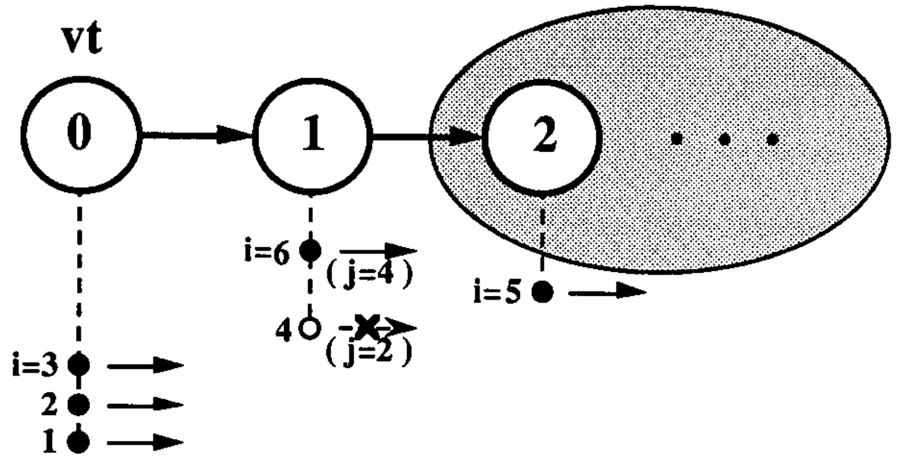


Figure 4.5: The State Transition in Virtual-Time for the Cold-Start Model.

summary, processor i will not start executing instruction 1 until processor x finishes executing instruction 0, where x is the minimum value of i and j. At that time, there are still $x-1$ processors executing instruction 0. Based on the memoryless assumption in Sec. 1.5.1, the execution time for processor i to finish instruction 1 is the same as the residual execution time for any processor from processor 1 to $x-1$ to finish instruction 0. When all the processors from processor 1 to $x-1$ finish instruction 0, the GVT moves to 1. If processor i has not finished instruction 1 by then, it will be at GVT again. The probability of the above situation is $\frac{1}{x}$, which is the probability that the tagged processor (i.e., processor i) takes the longest time to finish its instruction among the x processors (processor i plus processors 1 to $x-1$). Let π_{i_j} be the probability that a processor stays at vt 1 when GVT moves to 1 given its pid is i and the pid of the processor holding its remote operand is j. From the above argument, we have

$$\pi_{i,j} = \frac{1}{\min(i,j)} \tag{4.7}$$

The term $\frac{G_1}{N}$ is the probability that any single processor stays at vt 1 when GVT moves to 1. Unconditioning Eq. (4.7) over i and j, we have

$$\frac{G_1}{N} = \sum_{i=1}^{N}\sum_{j=1}^{N} \frac{\pi_{i,j}}{N^2} \tag{4.8}$$

Substituting Eq. (4.7) into Eq. (4.8), we have

$$\begin{aligned}
G_1 &= \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{N}\frac{1}{\min(i,j)} \\
&= \frac{1}{N}\left(\sum_{i=1}^{N}\sum_{j=i+1}^{N}\frac{1}{i} + \sum_{i=1}^{N}\sum_{j=i}^{N}\frac{1}{j}\right) \\
&= \frac{1}{N}\left(\sum_{i=1}^{N}\frac{N-i}{i} + \sum_{j=1}^{N}\frac{N-j+1}{j}\right) \\
&= \frac{1}{N}(N * H(N) - N + (N+1)H(N) - N) \\
&= 2 * (H(N) - 1) + \frac{H(N)}{N} \\
&\approx 2 * (H(N) - 1) \qquad \text{for large } N
\end{aligned} \tag{4.9}$$

Substituting Eq. (4.9) into Eq. (4.6), we have the lower bound on the progress rate r.

$$r \geq \frac{1}{H(2 * (H(N) - 1))} \propto \frac{1}{\log\log N} \tag{4.10}$$

This lower bound is inversely proportional to $(\log\log N)$. Though the lower bound drops to 0 for extremely large N, this bound shows that asynchronous execution is capable of improving the progress rate from $\frac{1}{\log N}$ (the progress rate of synchronous execution [Fel90]) to at least $\frac{1}{\log\log N}$.

The cold-start model can be further generalized to the case in which every instruction needs more than one remote operand. Let A be the number of ancestors of each instruction (i.e., every instruction needs $A-1$ remote operands). Then Eq. (4.7) becomes

$$\pi_{i_1,\ldots,i_A} = \frac{1}{\min(i_1,\ldots,i_A)} \tag{4.11}$$

and Eq. (4.9) becomes

$$\begin{aligned} G_1 &= \frac{1}{N^{A-1}} \sum_{i_1=1}^{N} \cdots \sum_{i_A=1}^{N} \frac{1}{\min(i_1,\ldots,i_A)} \\ &\approx A*(H(N)-H(A-1)) \qquad \text{for large } N \end{aligned} \tag{4.12}$$

Finally, the lower bound on r, i.e., Eq. (4.10), becomes

$$r \geq \frac{1}{H(A*(H(N)-H(A-1)))} \tag{4.13}$$

4.4 The Roll-Back Model

The roll-back model gives an approximation to r because we cannot derive the progress rate from the persistent model directly. Like the non-persistent model, the roll-back model also handles the unavailable remote operands differently from the persistent model. Whenever the remote operand of the instruction of a processor is not available, the processor will not wait for it. Instead, the processor makes a *guess* of the unavailable remote operand and then goes on to execute the next instruction so that the processor never gets blocked. In order to maintain correct execution, roll-back is mandatory when the guess is wrong. When the remote operand becomes available, it is checked against the guess. If the guess is wrong, the processor has to roll-back to the instruction where the guess is made. We assume that the guess is *always* wrong but that there is no roll-back overhead. Then, the net progress of this processor is the same as if it had been blocked there in the first place when the remote operand is

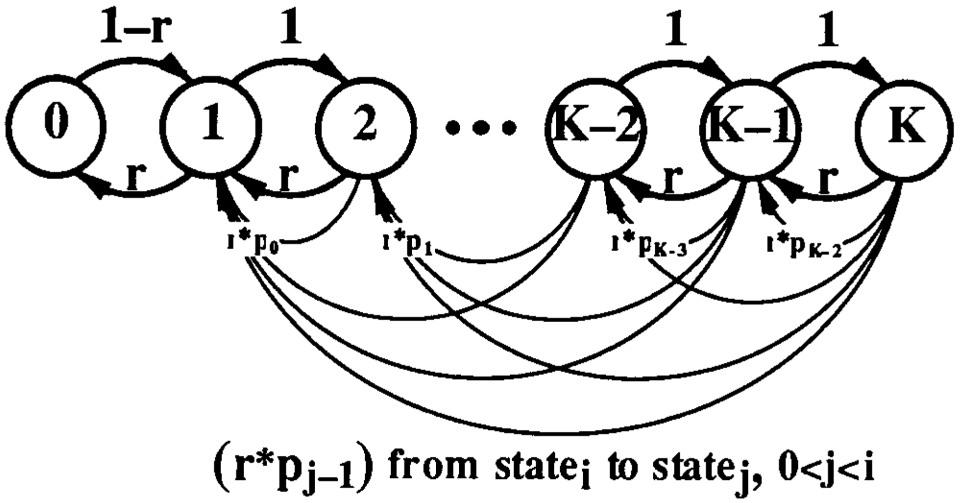


Figure 4.6: The State-Transition-Rate Diagram of Relative Virtual-Time for the Roll-Back Model.

unavailable. We are *not* proposing to implement the machine according to the roll-back model. We simply point it out that with respect to performance, the roll-back model gives the same result as the persistent model.

Figure 4.6 shows the state-transition-rate diagram of the relative virtual-time of a (tagged) processor in steady state for the roll-back model. In this model, we assume the memory history has a limited size K. The state-transition-rate diagram for the roll-back model is similar to that of the decoupled model. The difference is the existence of extra roll-back transitions in the roll-back model. Suppose that a processor (processor A) is at `rvt` i and the processor holding its remote operand (processor B) is at `rvt` j, where $j < i$. When processor B finishes its instruction at `rvt` j (and hence, moves to `rvt` $j + 1$), processor A must be rolled-back to `rvt` $j + 1$. However, if processor B is later rolled-back beyond `rvt` $j + 1$, then the intermediate roll-back of processor A

now is unnecessary because it will be rolled-back again later on. On average, the probability that the execution of an instruction is *not* rolled-back is simply the progress rate r. Therefore, a simple approximation for the roll-back rate from state i to state $j+1$ is $r * p_j$ for $j < i$. For this chain, we consider all the states greater than or equal to i as a group. Setting up the flow balance equations (the rate that flows out of the group is equal to the rate that flows into the group in steady state), we have

$$p_1 * r = p_0 * (1 - r) \tag{4.14}$$

and

$$p_i * r + \left(\sum_{j=i}^{K} p_j * r \right) * \sum_{j=0}^{i-2} p_j = p_{i-1} \qquad 1 < i \leq K \tag{4.15}$$

Together with the conservation equation

$$\sum_{i=0}^{K} p_i = 1 \tag{4.16}$$

and Eq. (3.3) in Sec. 3.3, which we repeat here,

$$p_0 \approx \frac{1}{r * N} \tag{4.17}$$

we can solve r and the p_i's numerically. Figure 4.7 and Figure 4.8 show the distributions of processors in **rvt** for $K = 24$ and $K = 16$, respectively. We note that the curves in these two diagrams are almost identical in the region in which the **rvt** is less than 12, where the magic number 12 is a sufficiently large size for the memory history as shown in Fig. 2.4. A larger memory history does not help to improve the progress rate because processors at large **rvt** will usually be rolled-back anyway. If the roll-back overhead is very large, it is better to suspend those processors which are too much ahead of the others.

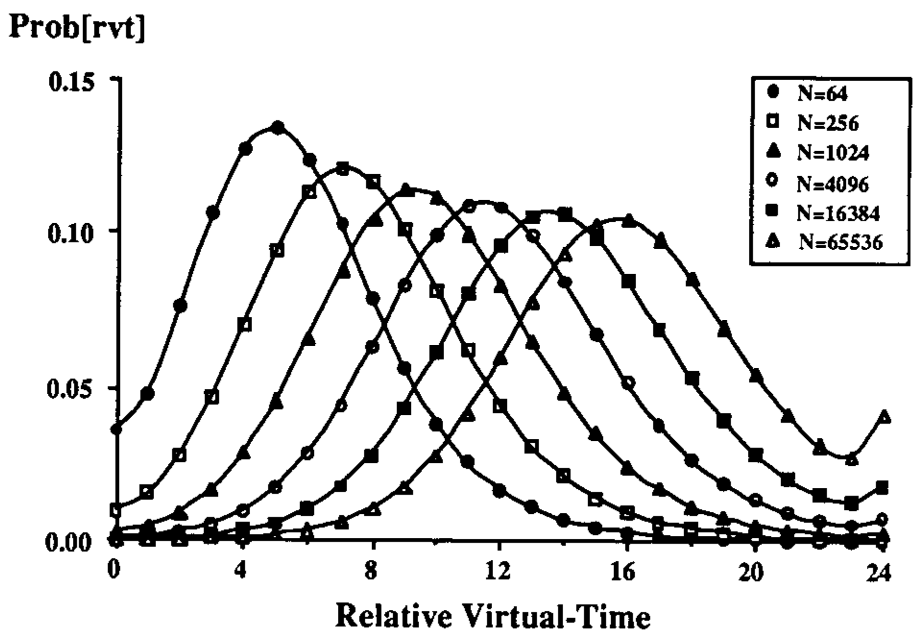


Figure 4.7: The Distribution of Processors(Roll-Back Model with $K = 24$).

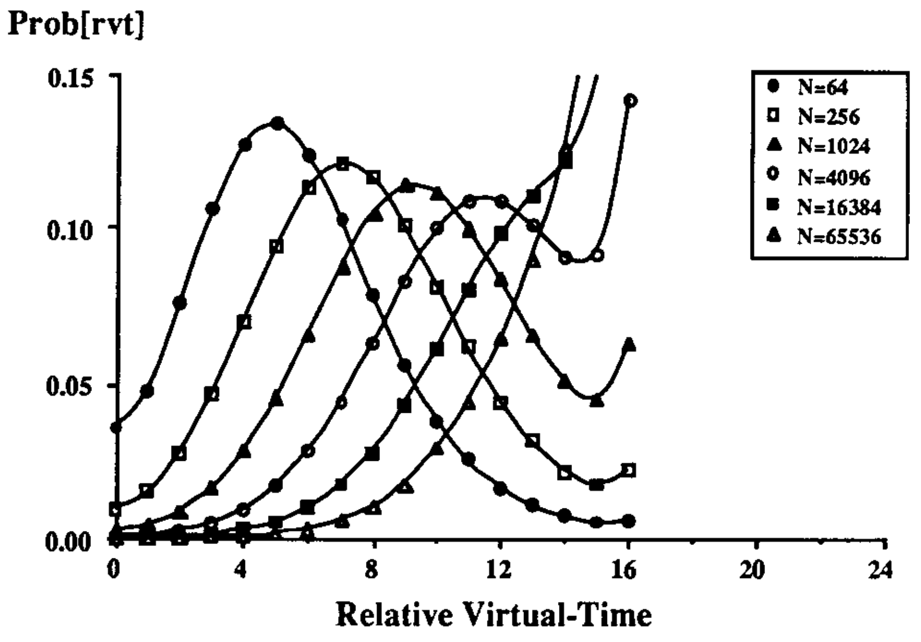


Figure 4.8: The Distribution of Processors (Roll-Back Model with $K = 16$).

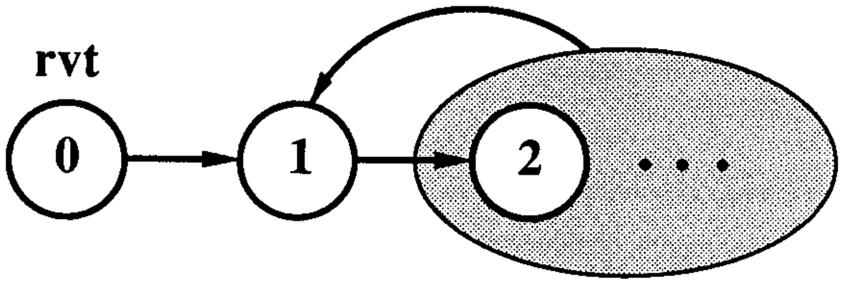


Figure 4.9: The State-Transition in Relative Virtual-Time for the Infinite-Processor Model.

4.5 The Infinite-Processor Model

The infinite-processor model gives the limiting value of r for a large number of processors in the system. This model combines the techniques developed in the cold-start and roll-back models to derive the steady-state behavior of the system. From the assumptions in Sec. 1.5.1, every instruction always needs one remote operand and the location of the remote operand is uniformly distributed among the processors. Conversely, the result of an instruction is expected to be requested by one of the other processors. When a processor at `rvt` 0 finishes its instruction, it advances to `rvt` 1 (suppose GVT does not advance), and at the same time, it may cause other processors to roll-back to `rvt` 1. The expected number of rolled-back processors is 1 if the number of remote operands of every instruction is 1.

Figure 4.9 shows the interesting state transitions in `rvt` between two consecutive advances of GVT. In steady state, the expected number of processors at `rvt` 1 right before GVT advances[2] is equal to the expected number of processors

[2]Assume that the advancing of the last processor at GVT and its corresponding roll-backs

at `rvt` 0 right after GVT advances.

Right before GVT advances, processors at `rvt` 1 are from one of the following:

1. originally[3] at `rvt` 1, and still at `rvt` 1,
2. originally at `rvt` 0, advanced to `rvt` 1, and still at `rvt` 1, or
3. originally at `rvt` > 1, rolled-back to `rvt` 1, and still at `rvt` 1.

Let N_0 and N_1 be the expected number of processors at `rvt` 0 and 1, respectively, right after GVT advances. The probability of the first case is $\frac{1}{N_0+1}$, which is the probability that this processor takes a longer time to finish its instruction than do the N_0 processors at `rvt` 0. The second and third cases are highly related. When a processor moves from `rvt` 0 to `rvt` 1, it causes, on average, one processor to roll-back from `rvt` > 1 to `rvt` 1. Because of this, we need only consider the second case; the third case has the same behavior. Let i be the *reverse* order of finishing the instruction at `rvt` 0 for the N_0 processors at `rvt` 0. For processor i, the probability that it is still at `rvt` 1 (after it advances to `rvt` 1) when all the $i-1$ processors at `rvt` 0 finish the instruction at `rvt` 0 is $\frac{1}{i}$. Therefore, we have

$$N_0 = \frac{N_1}{N_0+1} + 2 * \sum_{i=1}^{N_0} \frac{1}{i} \tag{4.18}$$

In addition, we have the flow balance equation between `rvt` 0 and `rvt` 1.

$$(1-r) * N_0 = r * N_1 \tag{4.19}$$

happens before GVT advances.

[3]The time of reference is right after GVT advances.

From Eq. (2.18) in Sec. 2.3, we have

$$r \approx \frac{1}{H(N_0)} \tag{4.20}$$

Substituting Eq. (4.20) and (4.19) into Eq. (4.18), we have

$$N_0 \approx \frac{(H(N_0) - 1) * N_0}{N_0 + 1} + 2 * H(N_0) \tag{4.21}$$

From Eq. (2.17), (2.18), and (2.22), we have

$$H(N_0) \approx \sqrt{N_0} \qquad \text{for small } N_0 \tag{4.22}$$

Substituting Eq. (4.22) into Eq. (4.21), we can solve N_0 and r.

$$N_0 \approx 7 \tag{4.23}$$

and

$$r \approx 0.386 \tag{4.24}$$

The infinite-processor model can be further generalized to the case where every instruction needs more than one remote operand. Let A be the number of ancestors of each instruction, as defined in Sec. 4.3. Then Eq. (4.18) becomes

$$N_0 = \frac{N_1}{N_0 + 1} + A * \sum_{i=1}^{N_0} \frac{1}{i} \tag{4.25}$$

From Eq. (4.25), (4.19), and (4.20), we can solve r numerically. Figure 4.10 shows the efficiency of the system with an infinite number of processors.

4.6 Conclusions

Figure 4.11 shows the summary of the performance analysis of the progress rate r. The upper bound on r from the non-persistent model is not tight.

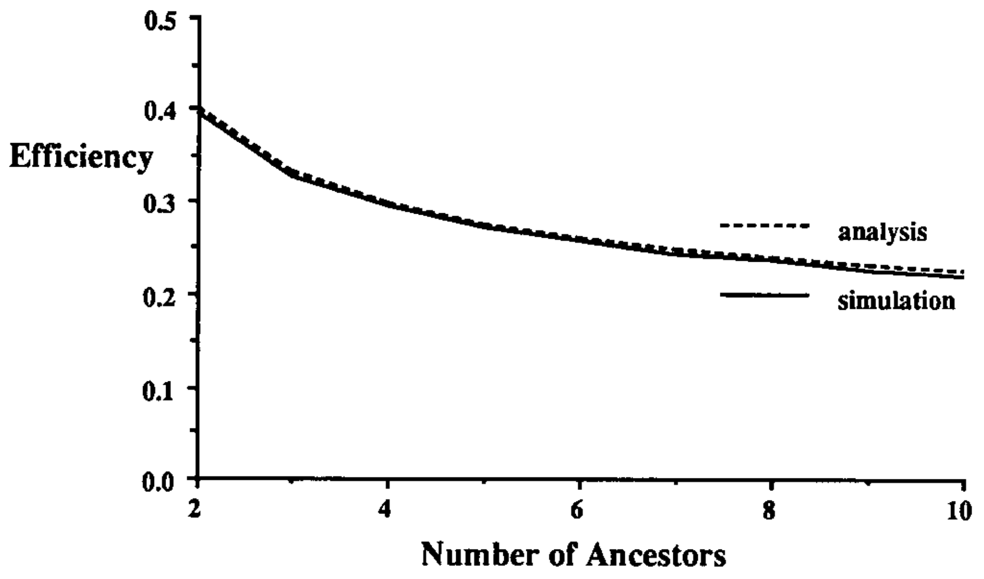


Figure 4.10: Efficiency of the System for the Infinite-Processor Model.

However, the non-persistent model gives us a quick-and-dirty illustration of the behavior of the Virtual-Time Data-Parallel Machine, e.g., constant efficiency and the bell shape of the dispersion function. The lower bound on r from the cold-start model is not tight, either. However, the cold-start model is still very important simply because it gives a *lower bound* on the progress rate. The lower bound serves as a *proof* that asynchronous execution is much more efficient than synchronous execution for a large number of processors, even though the lower bound drops to 0 when the number of processors increases without bound. The roll-back model gives a moderately good approximation to r. The approximation curve follows the simulation curve to within 10% error. However, the approximation from the roll-back model does not have a

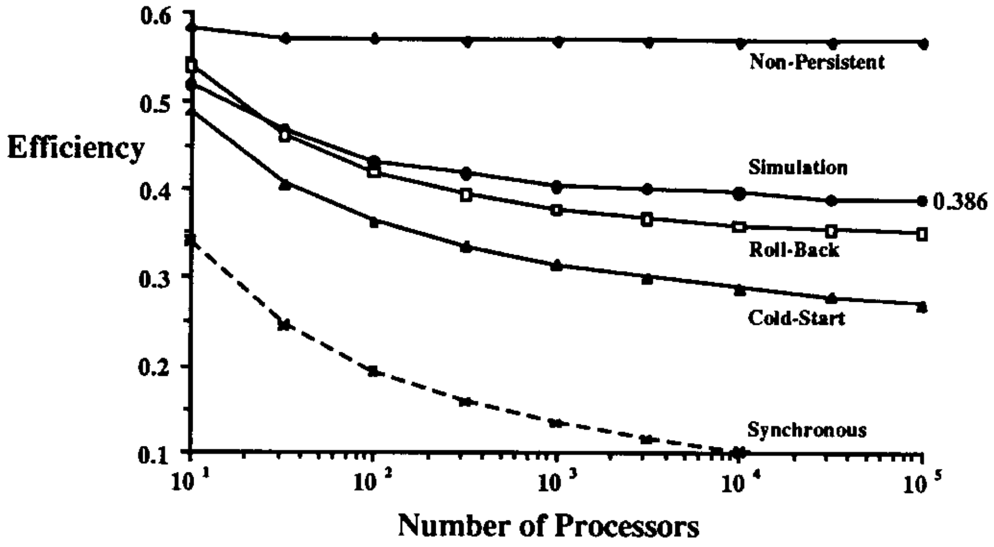


Figure 4.11: Summary of the Performance Analysis of the Progress Rate.

closed-form solution. The infinite-processor model gives an accurate estimate (within 2% error) of the efficiency of a very large system. Since we know that efficiency is a monotonically decreasing function of the number of processors in the system, an accurate estimate for a large system serves as a good lower bound for small systems as well. Though each of the four models has its own deficiencies, together, they give a comprehensive coverage of the performance analysis of the progress rate of the Virtual-Time Data-Parallel Machine.

Table 4.6 summarizes the models for the performance analysis of the Virtual-Time Data-Parallel Machine. The models in the previous two chapters are also included.

Model Name	Closed Form	Solves for
Parallel-Redundant	Yes	Mean-Time-to-Move of GVT
Decoupled	Yes	Size of Memory History
Non-Persistent	No	Upper Bound on r
Cold-Start	Yes	Lower Bound on r
Roll-Back	No	Approximation of r
Infinite-Processor	Yes	$\lim_{N\to\infty} r$

Table 4.1: Summary of the Models for Performance Analysis.

CHAPTER 5

Extensions of the Virtual-Time Data-Parallel Machine

5.1 Introduction

In the following three sections, the basic assumptions that we made in Sec. 1.5.1 are generalized to more realistic cases. Based on these generalized assumptions, we evaluate the efficiency of the Virtual-Time Data-Parallel Machine through the use of simulation. The next four sections address the directions to further improve the performance of the machine. By eliminating as much unnecessary blocking as possible, the Virtual-Time Data-Parallel Machine effectively converts the SIMD computation from control-flow to data-flow. The remaining sections address some miscellaneous issues such as the GVT algorithm and the interconnection network.

5.2 Execution Time Distribution

In Sec. 1.5.1, we made the assumption that the execution time of instructions is exponentially distributed. We know that in general, the execution time is more deterministic than this memoryless distribution. Therefore, we now choose a family of distributions with an adjustable coefficient of variation from 0 to 1 as a generalization of the execution time distribution. The family of r-stage Erlangian distributions (E_r) is chosen for this generalization, which is shown in Fig. 5.1 [Kle75]. The probability density function of the r-stage Erlangian

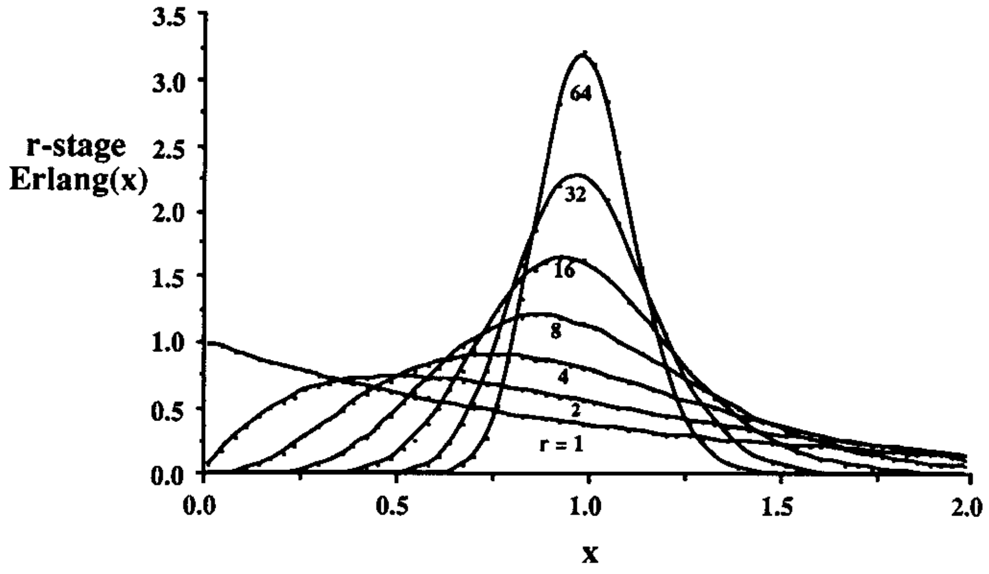


Figure 5.1: The Family of r-Stage Erlangian Distributions.

distribution is

$$E_r(x) = \frac{r(rx)^{r-1}e^{-rx}}{(r-1)!} \tag{5.1}$$

and its coefficient of variation is $\frac{1}{\sqrt{r}}$, which ranges between 0 and 1. Moreover, when r equals 1, the 1-stage Erlangian distribution is simply the exponential distribution, and when r equals infinity, the ∞-stage Erlangian distribution is deterministic.

From simulation, Fig. 5.2 shows the efficiency of asynchronous vs. synchronous execution of SIMD programs as a function of the coefficient of variation based on the above generalization, and Fig. 5.3 shows the efficiency gain of asynchronous over synchronous execution. From Fig. 5.2 and 5.3, we see that this model of asynchronous execution always[1] outperforms synchronous execution regardless of the execution time distribution; the larger the coefficient of variation, the greater the efficiency gain. Furthermore, the efficiency gain increases as the number of processors increases.

5.3 The Number of Remote Operands

In Sec. 1.5.1, we made the assumption that there is exactly one remote operand for each instruction. We know that in general, the expected number of remote operands per instruction is less than 1 because most of the time, instructions only need local variables. Note also, that in the instruction sets of most assembly languages, the number of source operands of an instruction ranges from zero to two. Therefore, the number of ancestors of each instruction ranges from one to three[2].

From simulation, Fig. 5.4 shows the machine efficiency (with 1024 processors) for various numbers of ancestors (denoted by A) per instruction, where the synchronous execution corresponds to having N ancestors, and Fig. 5.5 shows the efficiency gain of asynchronous over synchronous execution for this model. From Fig. 5.4 and 5.5, we note that asynchronous execution (i.e., with few ancestors) again outperforms synchronous execution (i.e., with many ancestors). The performance gain increases as the expected number of ancestors decreases

[1]However, for constant execution times, synchronous execution performs equally well as asynchronous execution.

[2]The *sequential* semantics of the language adds an (implicit) ancestor to each instruction, which is the preceding instruction on the same processor.

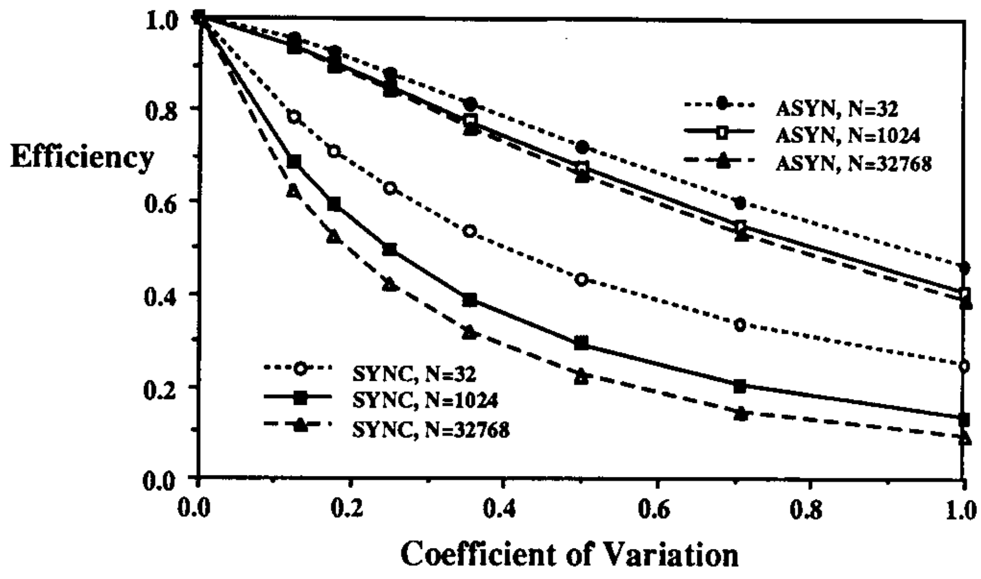


Figure 5.2: Efficiencies of the SIMD Machines for the Generalized Execution Time – Asynchronous vs. Synchronous.

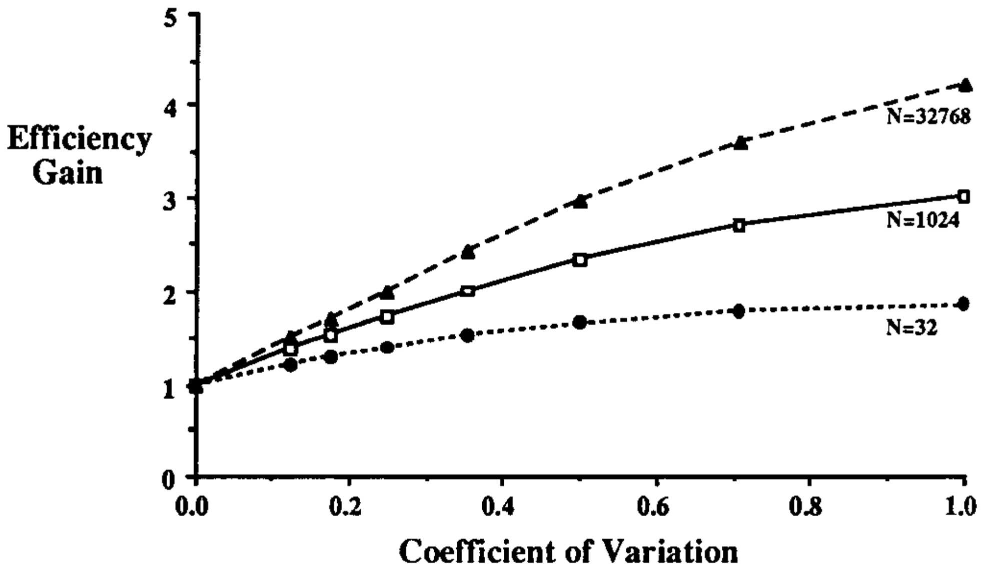


Figure 5.3: The Efficiency Gain for the Generalized Execution Time – Asynchronous over Synchronous.

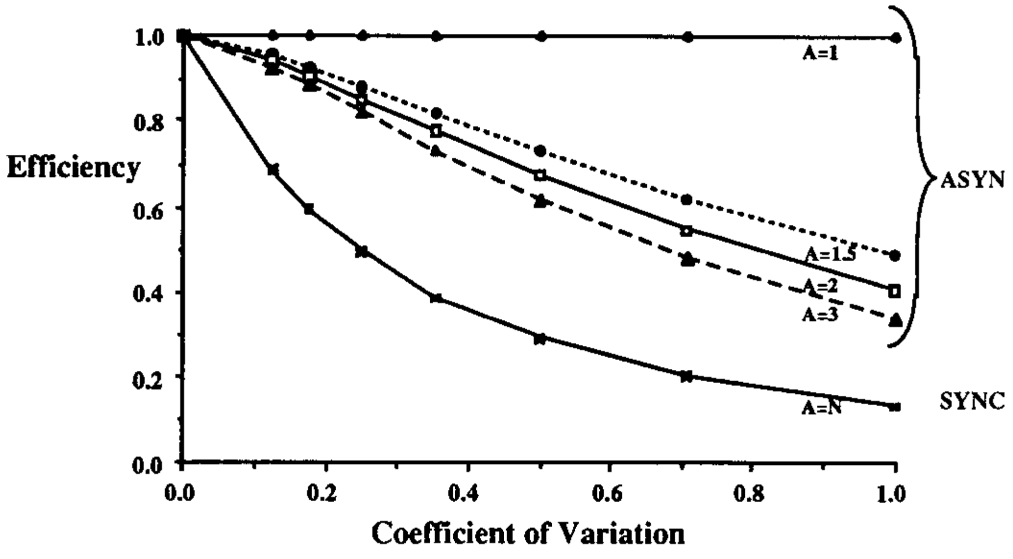


Figure 5.4: Efficiency of the Machine for Various Numbers of Ancestors (N=1024).

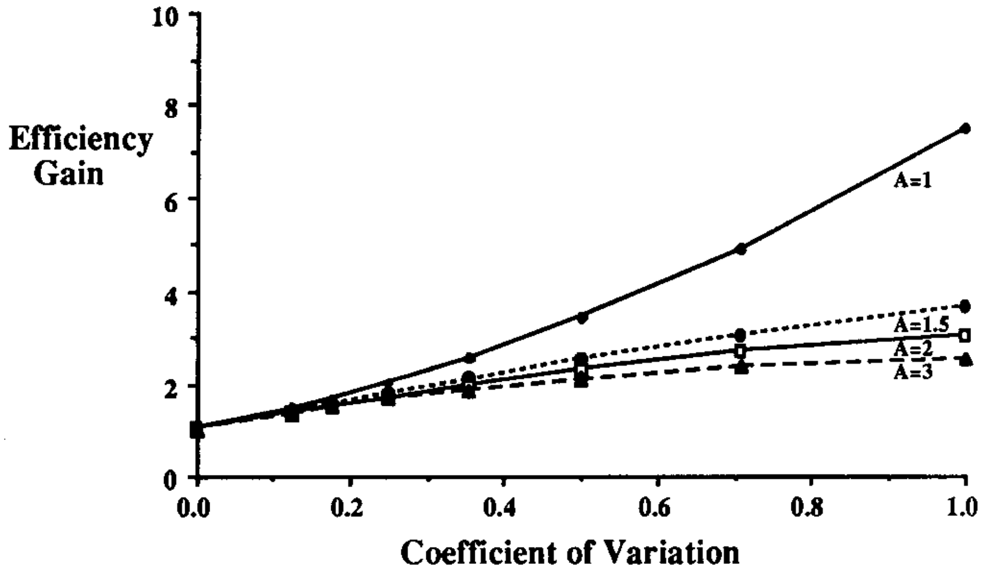


Figure 5.5: The Efficiency Gain for Few Ancestors over Infinitely-Many Ancestors (N=1024).

or as the coefficient of variation of the execution time distribution increases.

5.4 Location of Remote Operands

In Sec. 1.5.1, we made the assumption that the location of the remote operands is uniformly distributed among the processors, but in real cases, there are usually some fixed access patterns. Section 1.5.1 explained why the uniform distribution is a pessimistic assumption. From simulation, Fig. 5.6 shows the efficiency of the machine for various access patterns. We see that the efficiency

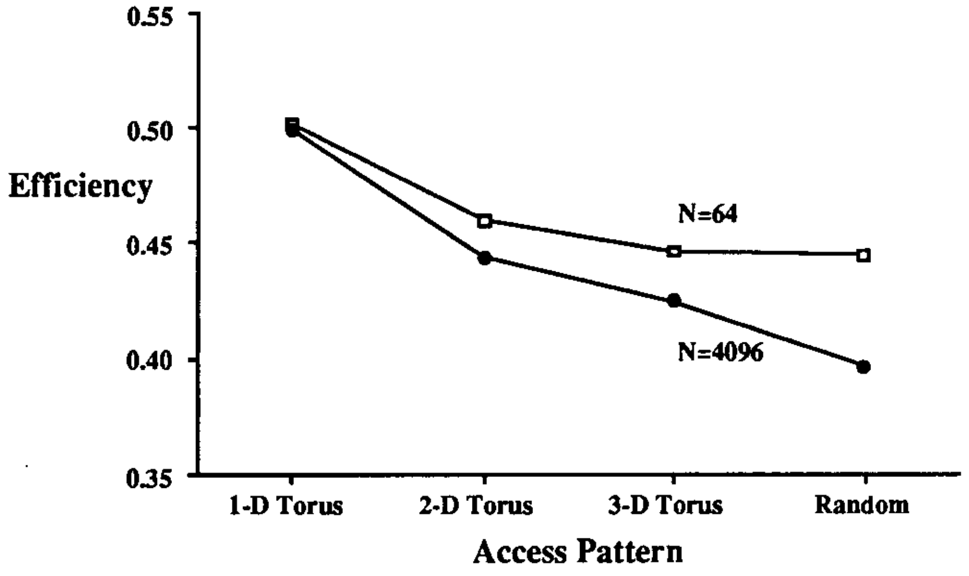


Figure 5.6: The Efficiency of the VT-DP Machine for Various Access Patterns of the Remote Operand.

of the random access pattern (i.e., uniform distribution among processors) is worse than that of more regular access patterns. We also observe that the efficiency changes only slightly for different access patterns. Hence, the uniform distribution assumption is reasonably valid.

5.5 Data Dependency Distance

The ancestors of an instruction are the processors from which the instruction fetches its operands. According to the SIMD semantics, the operands of the instruction at vt i refer to the values of these variables on its ancestors at the end of vt $(i-1)$. The data dependency distance between an instruction and its operand is defined to be the difference between the virtual-time of the instruction and the virtual-time at which the variable was last modified. The *sequential* SIMD semantics do not address the data dependency distance, and thus the data dependency distance is effectively one.

For example, Fig. 5.7.a shows the task graph of a SIMD program. A data dependency link in Fig. 5.7.a means that this instruction depends on *some* variable held by this ancestor *one* instruction earlier. When the granularity of data dependency is in terms of processors, waiting for a (tagged) variable on a processor at vt $(i-1)$ is equivalent to waiting for the processor to finish instruction $(i-1)$. However, the value of the *tagged* variable at the end of vt $(i-1)$ *may* be untouched during the execution of instruction $(i-1)$. For example, we see that the third instruction is independent of the second instruction. The odds of the above situation are often high[3] because there are many variables in one processor but usually only one of them is modified per instruction. In this example, the tagged variable, X, is not modified by instruction $(i-1)$,

[3]We are trying to take advantage of the *non-temporal-locality* of memory reference.

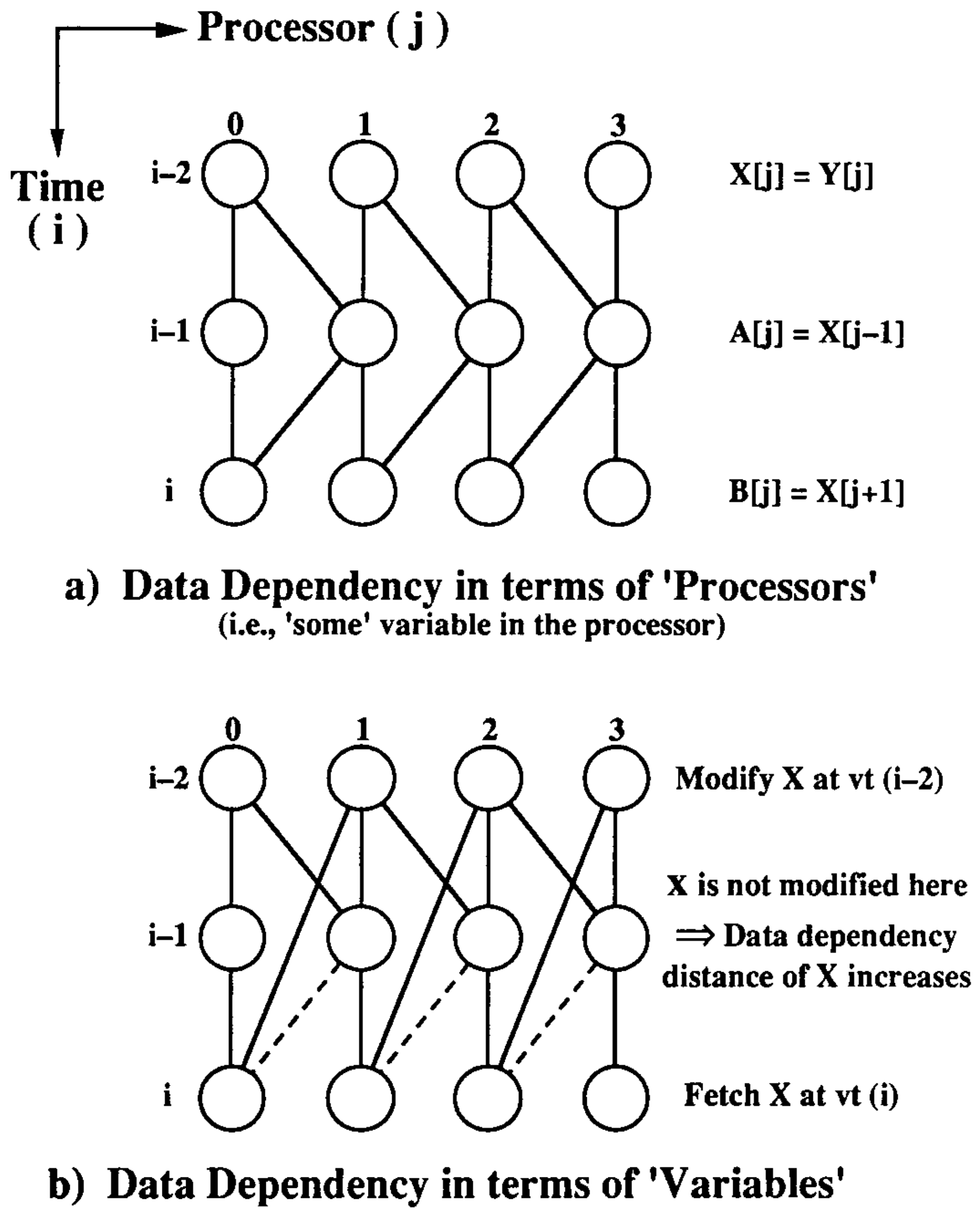

Processor (j)
Time (i)
0
1
2
3
i–2
i–1
i
X[j] = Y[j]
A[j] = X[j–1]
B[j] = X[j+1]
a) Data Dependency in terms of 'Processors'
(i.e., 'some' variable in the processor)
0
1
2
3
i–2
i–1
i
Modify X at vt (i–2)
X is not modified here
⟹ Data dependency distance of X increases
Fetch X at vt (i)
b) Data Dependency in terms of 'Variables'

Figure 5.7: Data Dependency Distance of Remote Operands.

and thus the value of this variable at the end of `vt` $(i-2)$ is the same as that at the end of `vt` $(i-1)$. In other words, the data dependency distance is increased by one. Figure 5.7.b shows the same task graph as Figure 5.7.a when the granularity of data dependency is in terms of variables. Note that the data dependency distance of the second instruction is one since its operand was modified during the previous instruction, while the data dependency distance of the third instruction is two since it was last modified two instruction ago.

The data dependency distance for a tagged variable at `vt` i can be further increased to d if it is last modified at `vt` $(i-d)$. When the data dependence distance is d, as long as the ancestor is less than d instruction behind, the remote operand is available and the processor can execute the instruction without waiting for the ancestor to catch up. The larger the data dependency distance, the more likely it is that the remote operand of an instruction is available, and hence, the less likely it is that the processor will be blocked.

From simulation, Fig. 5.8 shows the efficiency of asynchronous vs. synchronous SIMD machines for various data dependency distances (actually, the inverse of the data dependency distance), and Fig. 5.9 shows the efficiency gain of asynchronous over synchronous execution. From Fig. 5.8 and 5.9, we see that the efficiency can be improved up to 1 when the data dependency distance is very large, and the efficiency gain is significant even when the data dependency distance is small.

5.6 The Probability of No-Operation

One of the major deficiencies of synchronous execution is that it handles conditional instructions poorly. Figure 5.10 shows the different ways of implementing

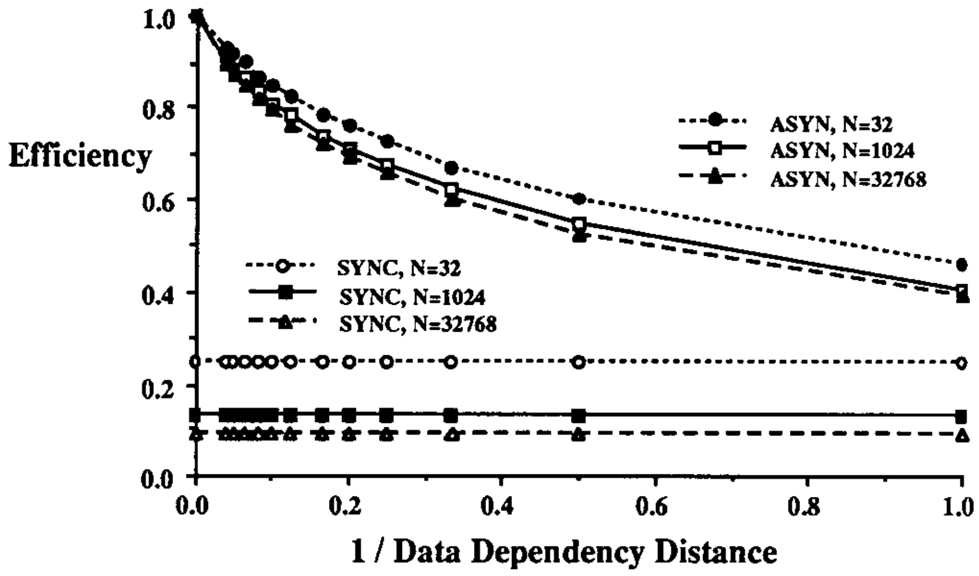


Figure 5.8: Efficiencies of the SIMD Machines for Various Data Dependency Distances – Asynchronous vs. Synchronous.

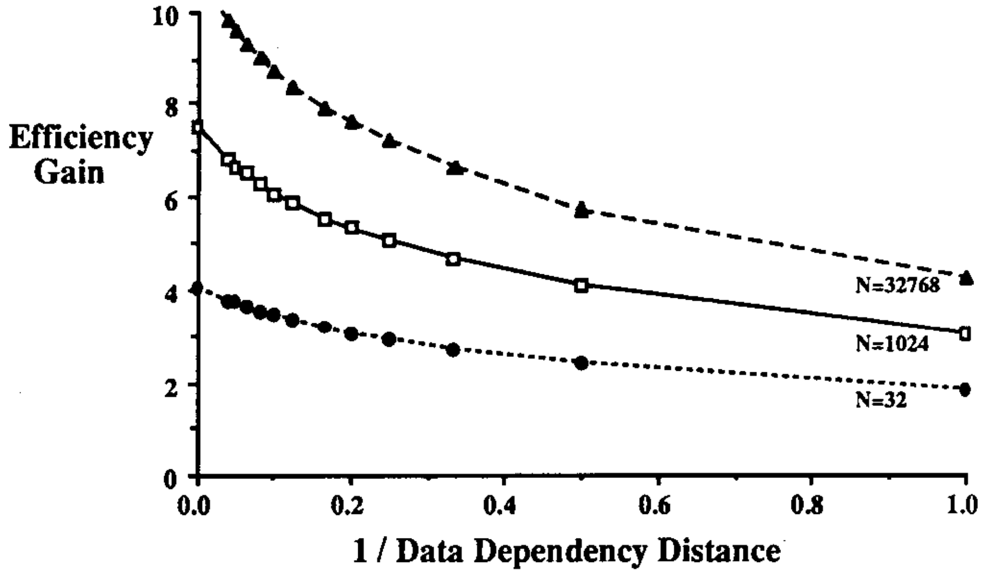
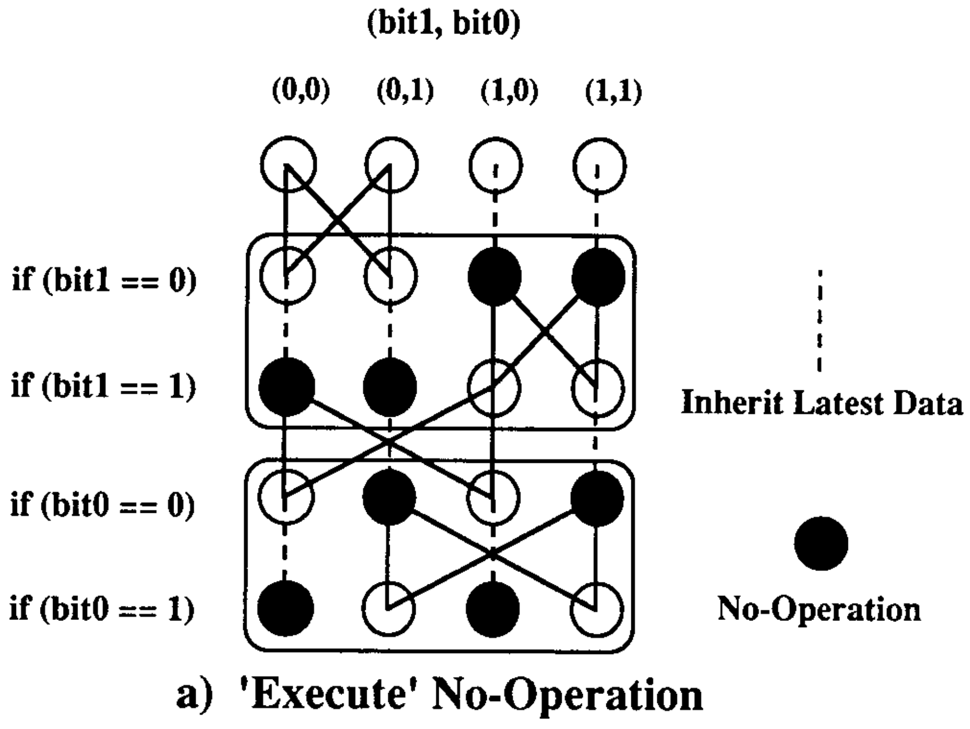


Figure 5.9: The Efficiency Gain for Various Data Dependency Distances Asynchronous over Synchronous.

the conditional instructions for synchronous and asynchronous execution. A no-operation denotes a disabled instruction (i.e., the condition of the instruction is false) for a particular processor. Synchronous execution (Fig. 5.10.a) *executes* the no-operations in the sense that a no-operation makes the processor wait while the other processors execute the instruction. Asynchronous execution (Fig. 5.10.b) *skips* the no-operation in the sense that a no-operation takes no time to finish, which is equivalent to skipping the no-operation.

From simulation, Fig. 5.11 shows the efficiency of asynchronous vs. synchronous execution as a function of the probability of no-operation for various numbers of processors, and Fig. 5.12 shows the efficiency gain of asynchronous over synchronous execution. From these figures, we note that the no-operations are a major source of performance degradation because of idle processors. Synchronous execution handles the no-operations poorly such that the efficiency drops significantly percentage-wise, while asynchronous execution retains good efficiency even though the probability of no-operation is high. The efficiency gain increases dramatically as the probability of no-operation increases or as the number of processors increases.

No-operations come from conditional instructions. If the condition is tight (i.e., it does not hold for most of the processors), then the probability of no-operation is high. Figure 5.13 shows a tree-reduction operation, which is a popular primitive operation in data-parallel programming. In a tree-reduction, the number of active processors is halved at each iteration until there is only one left. For N processors, a tree-reduction takes $\log_2 N$ iterations to finish. The average probability of no-operation is as high as $1 - \frac{1}{\log_2 N}$, which is approximately 94% for $N = 65536$.



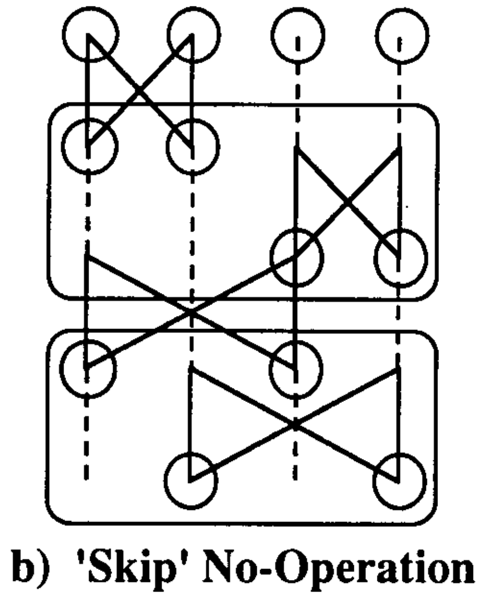


Figure 5.10: Implementation of Conditional Instructions in SIMD Architectures.

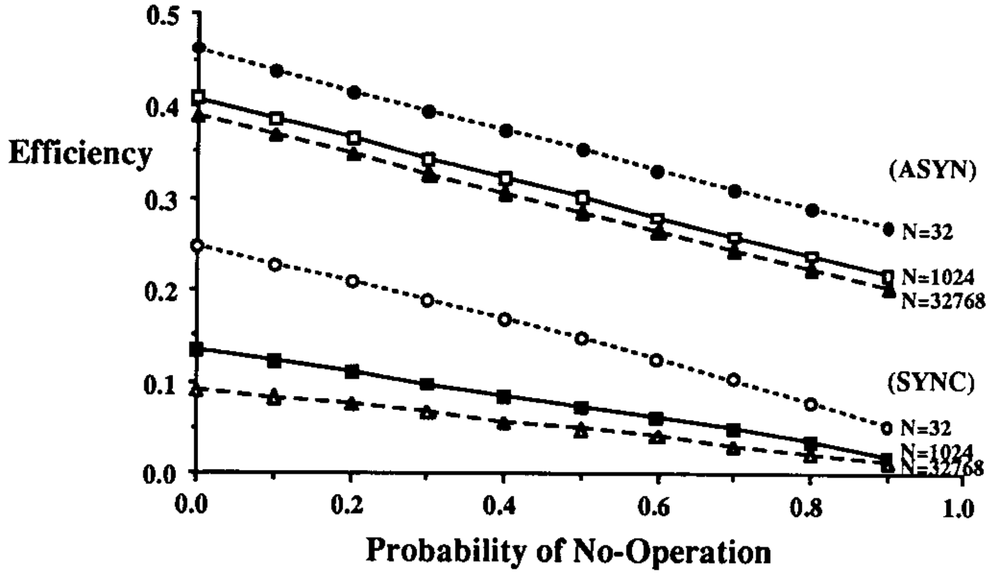


Figure 5.11: Efficiencies of the SIMD Machines in Handling the No-Operations – Asynchronous vs. Synchronous.

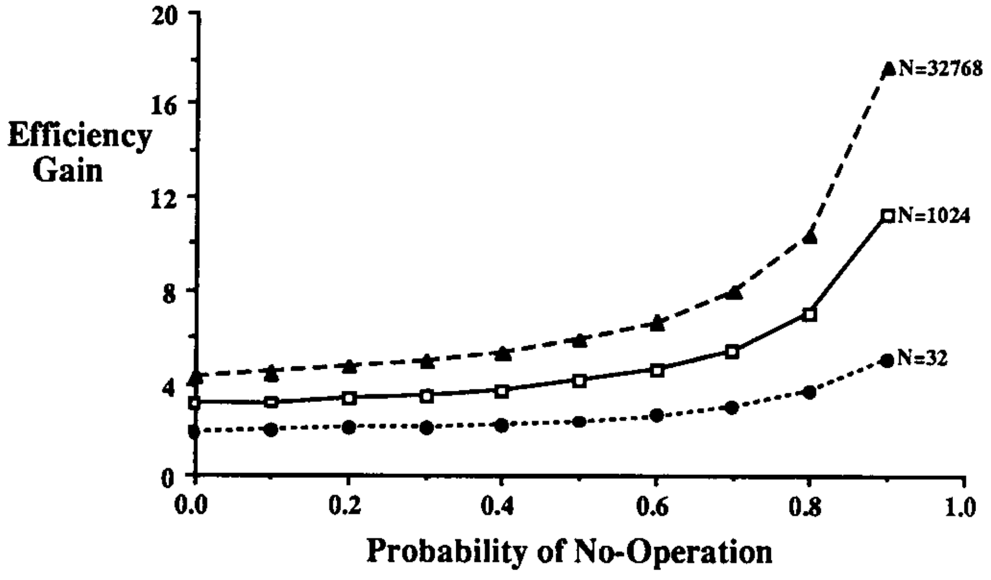


Figure 5.12: The Efficiency Gain in Handling the No-Operations – Asynchronous over Synchronous.

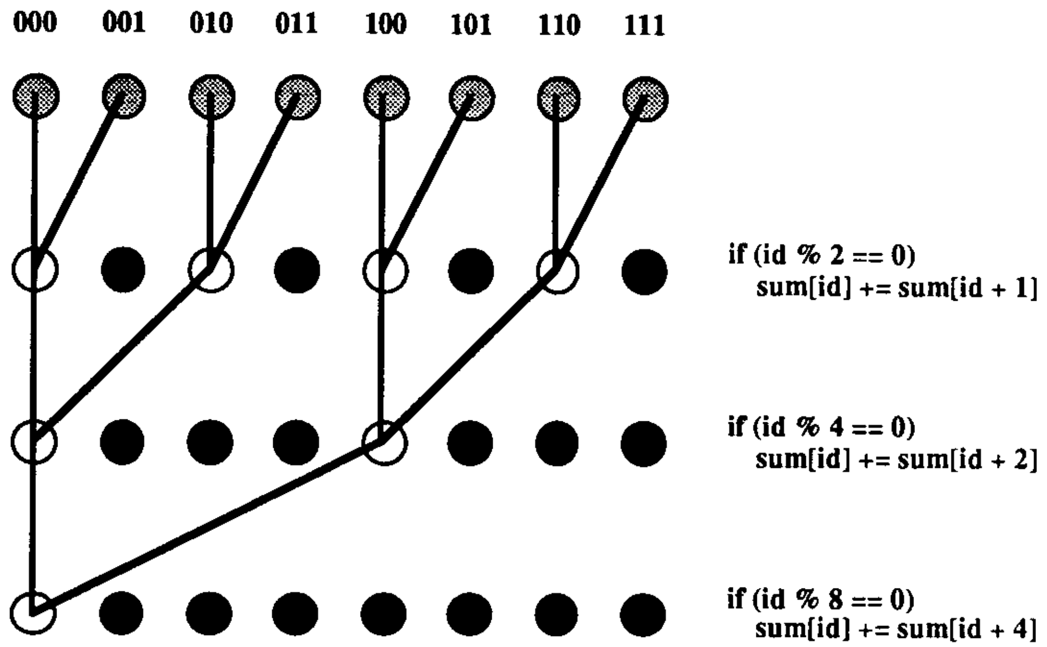


Figure 5.13: The Tree-Reduction Operation.

Figure 5.14 shows the efficiency gain of asynchronous over synchronous execution for $N = 1024$ when both the data-dependence distance and the probability of no-operation are considered. From Fig. 5.14, we see that asynchronous execution easily outperforms synchronous execution by one order of magnitude when both the data dependency distance and probability of no-operation are large.

5.7 The Two-Phase Write Algorithm

We have seen that the sequential semantics of SIMD programs adds an implicit ancestor to every instruction on every processor – the preceding instruction on the same processor. However, we also observed that the result of an instruction

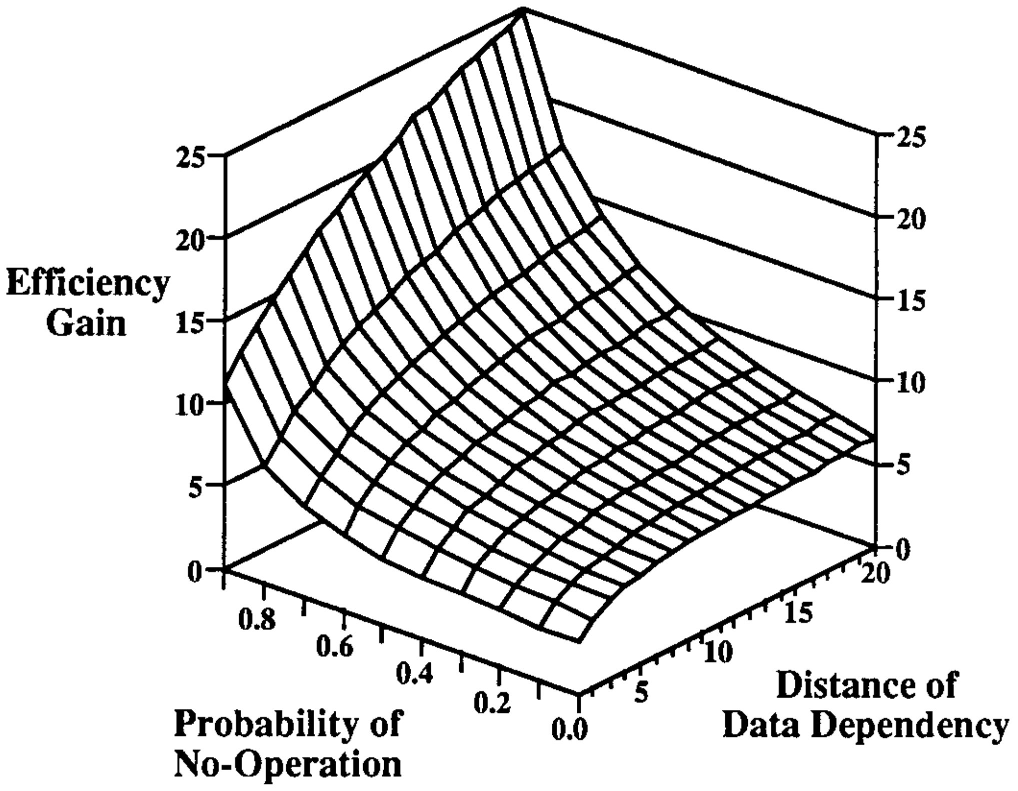


Figure 5.14: Efficiency Gain vs. Data-Dependency Distance and Probability of No-Operation (N=1024).

may not be used immediately by the next instruction.[4] If the current instruction is blocked (e.g., waiting for a remote operand), the execution of the next instruction can proceed without waiting for the current instruction to finish. Before the next instruction starts executing, the processor must schedule the execution of the current instruction and invalidate the variables that may be modified by the current instruction. Such a problem was addressed many years ago by the Tomasulo algorithm [Tom67]. This algorithm solved the problem

[4]Refer to Sec. 5.5 for more details.

that the IBM 360/91 machine had due to long pipeline delays in its floating point unit. It is one of the most sophisticated algorithms (even by today's standards) to convert sequential computation into data-flow computation within a small sliding window of instructions.

The main idea is to separate a `write` operation of a variable into two phases – the *logical* write and the *physical* write. The logical write is executed first before the content of the write operation is available; it invalidates the variable and assigns a unique identifier to the content of the write operation. From then on, all read requests to that variable (before the variable is overwritten) are transformed to waiting for that identifier. The next instruction can proceed after the logical write without waiting for the physical write. The physical write is executed when the content of the variable becomes available; it is sent to every processor waiting for the corresponding identifier. Once we adopt the two-phase write, then head-of-the-line blocking, which enforces sequential execution, is eliminated; at the same time, the sequential semantics are preserved. Thus we see that the techniques used to compensate for the long pipeline stages of floating point arithmetic units in sequential machine may now be used to compensate for the long (network) delays due to remote access in parallel processing systems.

The two-phase write can be easily implemented in the memory history by adding an extra *busy* bit to every outstanding update. A logical write sets the busy bit to one, representing the fact that an update is taking place, and the content is not available. A physical write resets the busy bit to zero, representing the fact that the data in this update is available. References to a busy update receive the time-stamped address of the update (which serves as the unique identifier), and then get blocked. When a physical write is executed, all references to the matching identifier are unblocked.

With the two-phase write, the Virtual-Time Data-Parallel Machine converts the SIMD computation from control-flow to data-flow (within a small sliding window of neighboring instructions). Data-flow execution *recovers* more threads of execution than control-flow [Pin85] [Pin86]; therefore increases the concurrency and improves the efficiency of the Virtual-Time Data-Parallel Machine. Thus we see that we can enjoy the benefits of data-flow execution while avoiding the current drawbacks of data-flow machines.

5.8 Load Balancing

There are some operations in data-parallel programming that put different loads on different processors depending on the `pid` (processor-id) of the processor. Take the tree-reduction as an example. For an N-processor system, the load on processor 0 is $(\log_2 N)$, while the load on processor 1 is zero. For tree-reduction, the load of a processor is equal to the number of trailing zeros of the processor's `pid`. When we have many tree-reductions, we would prefer to balance the load as fairly as possible across all processors such that the accumulated load of every processor is as even as possible.

Figure 5.15 shows the task graph of a data-parallel program consisting of a sequence of tree-reductions both with and without load balancing. Without load balancing (on the left-hand side), processor 0 has more work than the others and we can hardly take advantage of the no-operations on the other processors. With load balancing (on the right-hand side), the total amount of work is the same for all processors in which case skipping the no-operation

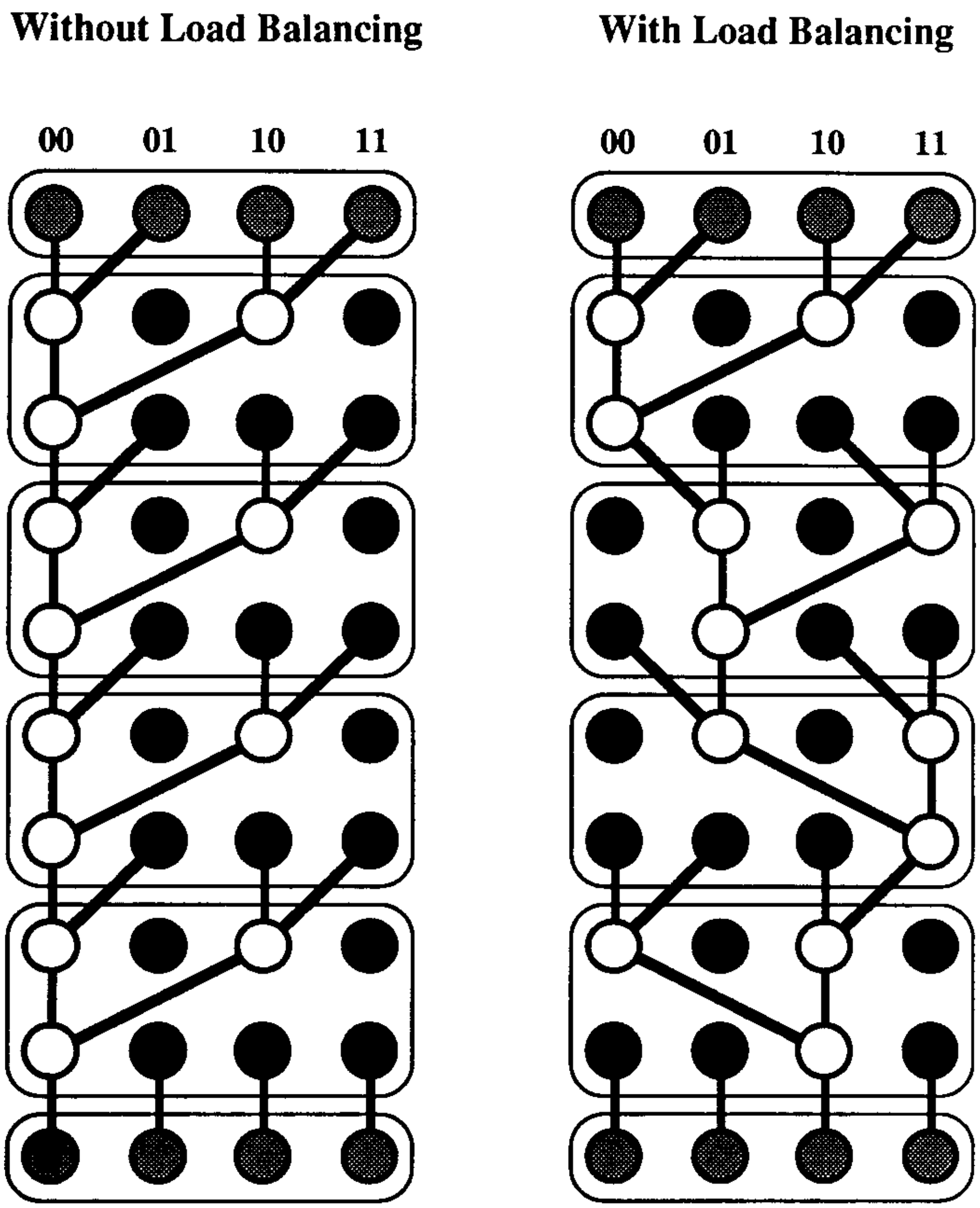


Figure 5.15: Load Balancing for a Sequence of Tree-Reduction Operations.

improves the performance a lot. If the data dependency distance is large, the longest path of Fig. 5.15 is no longer critical, that is, if the four segments of tree reductions in this figure were independent of each other, then the left hand side would still take 8 units of time whereas the right hand side takes 3 units of time for the tree reductions. Thus we see that proper load balancing is indispensable in order to make full use of the abundant no-operations and the large data dependency distance of SIMD programs.

5.9 Global Virtual-Time Algorithm

The algorithm for calculating GVT is very complicated for Time Warp because of roll-back [Bel90] [Dij80] [Cha82]. Since there is no roll-back in the Virtual-Time Data-Parallel Machine, the GVT algorithm is so simple that it can be easily implemented in hardware.

Recall that Global Virtual-Time is defined to be the minimum virtual-time among all processors. Because `min` is an associative operation, minimum-among-all can be divided into several phases. First, divide all into several partitions, then take the minimum inside each partition, and last take the minimum across all partitions. We may furthermore apply the rule recursively if desirable. Therefore, the GVT algorithm can be implemented by a combining tree, where the root of every subtree combines the minimum virtual-times of all its descendants. The GVT lag (defined in Sec. 3.3) is a function of the depth and of the branching factor of the combining tree.

5.10 Interconnection Networks

Interconnection networks play a critical role in parallel processing. Without proper design, they may become the performance and/or the economic bottleneck of parallel systems. Alleviating the load of interconnection networks improves the performance and (or) reduces the cost of the system.

In synchronous execution, the *worst* case of each instruction's execution time is important because processors which finish the instruction early must wait for the last one to finish. The variation of execution time mostly comes from the remote memory access instead of the computation inside the CPU. Therefore, the worst case of network delay becomes a primary criterion in the design of the interconnection network. In synchronous execution, the front-end processor triggers all processing elements to start executing the same instruction at the same time. If the instruction in execution needs a remote operand, then every processor issues a remote request at the same time, which puts a huge load on the interconnection network. On the other hand, if the instruction needs only local variables, there is no load on the interconnection network at all during the execution of this instruction. The interconnection network has either maximum load or no load at all, which results in either bad performance or low utilization. In summary, the emphasis in the design of the interconnection network for synchronous execution is on the worst delay at maximum load, regardless of the fact that the average delay or the average load may be much smaller.

In asynchronous execution, the *average* execution time is important. Therefore, the network design emphasizes the average, instead of the worst, network delay. Moreover asynchronous execution allows processors to execute different instructions at the same time; some instructions may need remote operands.

but others may not. The dispersion in instruction execution can smooth out the dramatic load fluctuation, such that the load of the interconnection network is close to the average load among neighboring instructions much of the time.

Asynchronous execution reduces the requirement (in terms of network delay) and the load of the interconnection network from the worst delay at the maximum load to the average delay at the average load. With asynchronous execution, we can consider more liberal medium access protocols on less expensive interconnection network topologies such as the ATM (asynchronous transfer mode) protocol [Han89] [Min89] on the DQDB (distributed queue dual bus) fiber optics network [New88] [IEEE89].

5.11 Conclusions

This chapter addressed some realistic issues of the Virtual-Time Data-Parallel Machine. The simulation results based on the generalized assumptions in this chapter are less elegant but more realistic, and thus more convincing than the analytical results in the last chapter based on some simple assumptions. The discussion in this chapter addresses the concern about the assumptions made in comparing asynchronous and synchronous execution; it also serves as a bridge from the abstract concept to the real implementation.

CHAPTER 6

Conclusions and Future Research

This dissertation has addressed the concept, characteristics, hardware support, performance analysis, and extensions of the Virtual-Time Data-Parallel Machine. Quantitative discussions are based on some simple assumptions, but in reality the results are expected to be application dependent. In Chapter 1, we illustrated that asynchronous execution of SIMD programs outperforms synchronous execution roughly by a factor of $(\ln N)$ where N is the number of processors in the system. In Chapter 2, we explained why the distribution of processors in relative virtual-time is independent of real-time, and then we showed, with the parallel-redundant model, how to derive the progress rate (r) by tracking the movement of global virtual-time. Hardware support for asynchronous execution was discussed in Chapter 3. A cost-effective FIFO priority cache was proposed to implement the incremental backup algorithm for the memory history, and an upper-bound on the size of the memory history was derived from the decoupled model. Several models were developed in Chapter 4 to analyze the progress rate of the instruction execution. The non-persistent model, the cold-start model, the roll-back model, and the infinite-processor model gave an upper bound, a lower bound, an approximation, and the limiting value of r, respectively. In Chapter 5 some extensions of the Virtual-Time Data-Parallel Machine were discussed, in particular, our earlier assumptions were relaxed and some miscellaneous issues were also addressed.

In summary, asynchronous execution outperforms synchronous execution regardless of the assumptions.[1] With low-cost hardware support for the memory history, it is quite reasonable to adopt asynchronous execution.

6.1 Architecture Simulator

An architecture simulator has been developed to verify the usefulness of the Virtual-Time Data-Parallel Machine on real programs. In this simulator, the assumptions regarding the program behavior in Sec. 1.5.1 (i.e., every instruction has one remote operand and the location of the remote operand is uniformly distributed among processors) are removed. Therefore, the performance as measured from the simulator is more convincing than are the results from the analysis. However, we still assume that the execution time of instructions is exponentially distributed. In general, the execution time distribution is problem and technology dependent. For simple and regular problems, the execution time distribution tends to be more deterministic than memoryless. Furthermore, the popular implementations of SIMD architectures are based on synchronous execution, which ignores those technologies that reduce the average execution time but increase the worst execution time (such as the cache memory).

From the simulator, we can demonstrate the *scalability* of the Virtual-Time Data-Parallel Machine. The scalability of data-parallel machines is defined differently from the traditional definition. The traditional definition of scalability is with respect to the speed-up of running the *same* program on an increasing number of processors. If the speed-up increases in proportion to the number

[1]Except for some extreme cases, synchronous execution is as good as asynchronous execution.

of processors, then the system scales-up well. This definition is not directly applicable to data-parallel machines where the number of processors is approximately on the same order as the intrinsic parallelism of the program. We cannot simply increase the number of processors alone because it breaks the balance between the number of processors and the intrinsic parallelism. When the number of processors is increased, the problem size must be increased proportionally such that the intrinsic parallelism is also increased proportionally. If the system scales-up well, then the execution time is almost constant; otherwise, the execution time increases as the system scales-up. Figure 6.1 shows the execution time required to solve a system of partial differential equations (i.e., Laplace's equation) for asynchronous vs. synchronous execution. This diagram shows that asynchronous execution indeed scales-up well because the execution time is almost constant, while the execution time for synchronous execution increases as the system scales-up. The above example does *not* imply that asynchronous execution is more efficient than synchronous execution in solving PDEs because the execution time of instructions is very deterministic for PDEs. This example illustrates that asynchronous execution is more *general* than synchronous execution in the sense that asynchronous execution achieves good efficiency even if the execution time is rather non-deterministic. Other than PDEs, complicated and irregular applications (e.g., VLSI simulation) that run poorly in synchronous execution *may* run efficiently in asynchronous execution.

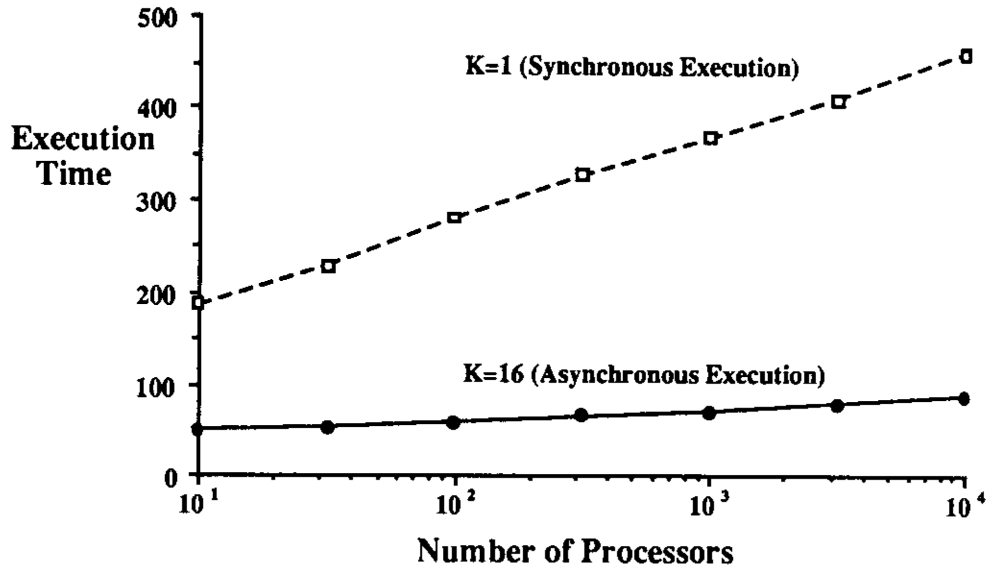


Figure 6.1: The Execution Time of a Program When the Number of Processors Scales-Up with the Problem Size.

6.2 Prototype

A prototype of the Virtual-Time Data-Parallel Machine seems to be the most obvious area for future work. This dissertation provided the analysis and simulation of the Virtual-Time Data-Parallel Machine, but building a prototype is the most *solid* proof of a good idea. The guidelines for the prototype might be as follows:

1. Choose a scalable interconnection network with a low average network delay for small fixed-size messages.

2. Add the hardware support for memory history (Sec. 3.2.2) to an existing but obsolete[2] RISC micro-processor to reduce the design cost as well as to make better utilization of the transistors.

3. Put as many processors as possible in one chip in today's technology to reduce the manufacturing cost.

A medium-size prototype with one thousand processors can be built within a reasonable budget.

6.3 Programming Environment

An assembler has been developed for the simulator to execute data-parallel programs written in assembly language. The assembler allows us to run various types of programs on the simulator, or to run the same program while changing the parameters of the simulator. Though the assembler serves well as a primitive programming environment for the simulator, a high level language compiler is mandatory to develop real programs or benchmarks when the prototype is built. A fancy compiler has become one of the most important survival kits for modern architectures. Advanced compiler technology is especially important for high performance computers where the effective use of the processors is mainly based on compile-time optimizations. Additionally, the design and standardization of high-level data-parallel languages is also important. With a well-designed standard data-parallel language as a baseline, customization for a particular implementation is hidden inside the compiler.

[2]An obsolete processor usually consists of much fewer transistors than the latest one. In terms of absolute processing power, the obsolete processor is inferior to the latest one, but in terms of processing power per transistor at a fixed clock rate, the obsolete processor is usually superior to the latest one.

Furthermore, a data-parallel language for multi-processor systems should work well on single-processor systems as well. Thus, the investment in the standard data-parallel language is protected across a broad range of architectures – from single-processor to various multi-processor systems.

6.4 Final Remarks

We proposed some minimal modifications to the architecture of the Connection Machine which converts the way it executes SIMD programs from synchronous to asynchronous. We have provided a basic but strong foundation for the understanding of both *why* and *how* to improve the efficiency of SIMD programs by allowing asynchronous execution. The investigation of the Virtual-Time Data-Parallel Machine is still in its infancy. While this dissertation serves as a starting point, there remains much work to be done.

REFERENCES

[Bai88] W. L. Bain and D. S. Scott, "An Algorithm for Time Synchronization in Distributed Event Simulation," *Proceedings of the SCS Multiconference on Distributed Simulation*, Vol. 19, No. 3, pp. 30–33, July 1988.

[Bar68] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes, "The ILLIAC IV Computer," *IEEE Transactions on Computers*, Vol. 17, No. 2, pp. 746–757, August 1968.

[Bat80] Kenneth E. Batcher, "Design of a Massively Parallel Processor," *IEEE Transactions on Computers*, Vol. 29, No. 9, pp. 836–840, September 1980.

[Baw84] Alan Bawden, *A Programming Language for Massively Parallel Computers*, Master Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1984.

[Bel90] Steven Bellenot, "Global Virtual Time Algorithms," *Proceedings of the SCS Multiconference on Distributed Simulation*, Vol. 22, No. 1, pp. 122–127, January 1990.

[Ble90] Guy E. Blelloch, *Vector Models for Data-Parallel Computing*, The MIT Press, Cambridge, Massachusetts, 1990.

[Bou72] W. J. Bouknight, Stewart A. Denenberg, David E. McIntyre, J. M. Randall, Amed H. Sameh, and Daniel L. Slotnick, "The ILLIAC IV System," *Proceedings of the IEEE*, Vol. 60, No. 4, pp. 369–388, April 1972.

[Bry77] R. E. Bryant, *Simulation of Packet Communication Architecture Computer Systems*, Ph. D. Dissertation, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1977.

[Cha79] K. Mani Chandy and Jayadev Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, Vol. 5, No. 5, pp. 440–452, September 1979.

[Cha81] K. M. Chandy and J. Misra, "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Communications of the ACM*, Vol. 24, No. 11, pp. 198–205, November 1981.

[Cha82] K. M. Chandy and J. Misra, "Termination Detection of Diffusion Computations in Communicating Sequential Processes," *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 1, pp. 37–43, January 1982.

[Cha91] C. S. Chang and R. Nelson, "Bounds on the Speedup and Efficiency of Partial Synchronization in Parallel Processing Systems," Research Report RC 16474, IBM Research Division, T. J. Watson Research Center, Yorktown Heights, January 1991.

[Chr83] David P. Christman, *Programming the Connection Machine*, Master Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1983.

[Dij80] E. W. Dijkstra and C. S. Scholten, "Termination Detection for Diffusion Computations," *Information Processing Letters*, Vol. 11, No. 1, pp. 1–4, August 1980.

[Fel90] Robert E. Felderman and Leonard Kleinrock, "An Upper Bound on the Improvement of Asynchronous Versus Synchronous Distributed Processing," In *Proceedings of the SCS Multiconference on Distributed Simulation*, Vol. 22, No. 1, pp. 131–136, January 1990.

[Fel91] Robert E. Felderman, *Performance Analysis of Distributed Processing Synchronization Algorithms*, Ph. D. Dissertation, Computer Science Department, University of California, Los Angeles, 1991.

[Fla77] P. M. Flanders, D. J. Hunt, D. Parkinson, and S. F. Reddaway, "Efficient High Speed Computing with the Distributed Array Processor," *Symposium on High Speed Computer and Algorithm Organization*, University of Illinois, Academic Press, pp. 113–128, 1977.

[Fuj88] Richard M. Fujimoto, Jya-Jang Tsai, and Ganesh Gopalakrishnan, "Design and Performance of Special Purpose Hardware for Time Warp," *Proceedings of the 15th International Symposium on Computer Architecture*, pp. 401–408, June 1988.

[Fuj89a] Richard M. Fujimoto, "Performance Measurements of Distributed Simulation Strategies," *Transactions of The Society for Computer Simulation*, Vol. 6, No. 2, pp. 89–132, April 1989.

[Fuj89b] Richard M. Fujimoto, "The Virtual Time Machine," *International Symposium on Parallel Algorithms and Architectures*, pp. 199–208, June 1989.

[Fuj90] Richard M. Fujimoto, "Parallel Discrete Event Simulation," *Communications of the ACM*, Vol. 33, No. 10, pp. 30–53, October 1990.

[Gho91] Kaushik Ghosh and Richard M. Fujimoto, "Parallel Discrete Event Simulation Using Space-Time Memory," *Proceedings of the 1991 International Conference on Parallel Processing*, Vol. 2, pp. 201–208, August 1991.

[Gro88] B. Groselj and C. Tropper, "The Time of Next Event Algorithm," *Proceedings of the SCS Multiconference on Distributed Simulation*, Vol. 19, No. 3, pp. 25–29, July 1988.

[Gro89] B. Groselj and C. Tropper, "A Deadlock Resolution Scheme for Distributed Simulation," *Proceedings of the SCS Multiconference on Distributed Simulation*, Vol. 21, No. 2, pp. 108–112, March 1989.

[Gup91] A. Gupta, I. F. Akyildiz, and R. M. Fujimoto, "Performance Analysis of "Time Warp" With Homogeneous Processors and Exponential Task Times," *Proceedings of the 1991 SIGMETRICS Conference*, Association for Computing Machinery, pp. 101–110, May 1991.

[Han89] Rainer Handel, "Evolution of ISDN Towards Broadband ISDN," *IEEE Network*, Vol. 3, No. 1, pp. 7–13, January 1989.

[Hay82] Leonard. S. Haynes, Richard L. Lau, Daniel P. Siewiorek, and David W. Mizell, "A Survey of Highly Parallel Computing," *IEEE Computer*, Vol. 15, No. 1, pp. 9–24, January 1982.

[Hil85] W. Daniel Hillis, *The Connection Machine*, The MIT Press, Cambridge, Massachusetts, 1985.

[Hil86] W. Daniel Hillis and Guy L. Steele, Jr., "Data Parallel Algorithms," *Communications of the ACM*, Vol. 29, No. 12, pp. 1170–1183, December 1986.

[Hor90] R. Michael Hord, *Parallel Supercomputing in SIMD Architectures*, CRC Press, Inc., Boca Raton, Florida, 1990.

[IEEE89] IEEE, *Draft of Proposed IEEE Standard 802.6 – Distributed Queue Dual Bus (DQDB) Metropolitan Area Network (MAN)*, IEEE, New York, New York, 1989.

[Jef85] David R. Jefferson, "Virtual Time," *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, pp. 404–425, July 1985.

[Kle75] Leonard Kleinrock, *Queueing Systems, Volume I: Theory*, John Wiley & Sons, New York, New York, 1975.

[Kuc68] D. J. Kuck, "ILLIAC IV Software and Application Programming," *IEEE Transactions on Computers*, Vol. 17, No. 2, pp. 758–769, August 1968.

[Kuc77] D. J. Kuck, "A Survey of Parallel Machine Organization and Programming," *ACM Computing Surveys*, Vol. 9, No. 1, pp. 29–59, March 1977.

[Lub89] B. D. Lubachevsky, "Efficient Distributed Event-Driven Simulations of Multiple-Loop Networks," *Communications of the ACM*, Vol. 32, No. 1, pp. 111–123, January 1989.

[Mad91] Vijay Madisetti, David Nicol, and Richard Fujimoto, editors, *Proceedings of the SCS Multiconference on Advances in Parallel and Distributed Simulation*, Society for Computer Simulation, Vol. 23, No. 1, January 1991.

[Mar91] M. Maresca and T. J. Fountain, editors, "Special Issue on Massively Parallel Computers," *Proceedings of the IEEE*, Vol. 79, No. 4, April 1991.

[Min89] Steven E. Minzer, "Broadband ISDN and Asynchronous Transfer Mode (ATM)," *IEEE Communications Magazine*, Vol. 27, No. 9, pp. 17–24, September 1989.

[Mis86] Jayadev Misra, "Distributed Discrete-Event Simulation," *ACM Computing Surveys*, Vol. 18, No. 1, pp. 39–65, March 1986.

[New88] R. M. Newman, Z. L. Budrikis, and J. L. Hullett, "The QPSX MAN," *IEEE Communications Magazine*, Vol. 26, No. 4, pp. 20–28, April 1988.

[Nic84] D. M. Nicol and P. F. Reynolds, Jr., "Problem Oriented Protocol Design," *Proceedings of 1984 Winter Simulation Conference*, pp. 471–474, December 1984.

[Nic88] D. M. Nicol, "Parallel Discrete-Event Simulation of FCFS Stochastic Queueing Networks," *SIGPLAN Not.*, Vol. 23, No. 9, pp. 124–137, September 1988.

[Nic91] David M. Nicol, "Parallel Self-Initiating Discrete-Event Simulations," *Transactions on Modelling and Computer Simulation*, Vol. 1, No. 1, pp. 24–50, January 1991.

[Par90] Dennis Parkinson and John Litt, editors, *Massively Parallel Computing with the DAP*, The MIT Press, Cambridge, Massachusetts, 1990.

[Pea79] J. Kent Peacock, J.W. Wong, and Eric G. Manning, "Distributed Simulation Using a Network of Processors," *Computer Networks*, Vol. 3, No. 1, pp. 44–56, February 1979.

[Pin85] Keshav Pingali and Arvind, "Efficient Demand-Driven Evaluation. Part 1," *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 2, pp. 311–333, April 1985.

[Pin86] Keshav Pingali and Arvind, "Efficient Demand-Driven Evaluation. Part 2," *ACM Transactions on Programming Languages and Systems*, Vol. 8, No. 1, pp. 109–139, January 1986.

[Pot85] J. L. Potter, editor, *The Massively Parallel Processor*, The MIT Press, Cambridge, Massachusetts, 1985.

[Rus78] Richard M. Russell, "The CRAY-1 Computer System," *Communications of the ACM*, Vol. 21, No. 1, pp. 63–72, January 1978.

[Sam85] Behrokh Samadi, *Distributed Simulation: Algorithms and Performance Analysis*, Ph. D. Dissertation, Computer Science Department, University of California, Los Angeles, 1985.

[Smi82] Alan Jay Smith, "Cache Memories," *ACM Computing Surveys*, Vol. 14, No. 3, pp. 473–530, September 1982.

[Su89] W. K. Su and C. L. Seitz, "Variants of the Chandy-Misra-Bryant Distributed Discrete-Event Simulation Algorithm," *Proceedings of the SCS Multiconference on Distributed Simulation*, Vol. 21, No. 2, pp. 38–43, March 1989.

[Tho61] James E. Thornton, "Parallel Operation in the Control Data 6600," *Fall Joint Computers Conference*, Vol. 26, pp. 33–40, 1961.

[Tom67] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal*, pp. 25–33, January 1967.

[Tri82] Kishor S. Trivedi, *Probability & Statistics with Reliability, Queuing, and Computer Science Applications*, Prentice Hall, Englewood Cliffs, New Jersey, 1982.