LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TR-425

CODE-MAPPING POLICIES FOR THE TAGGED-TOKEN DATAFLOW ARCHITECTURE

Gino K. Maa

May 1988

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

This blank page was inserted to preserve pagination.

Code-Mapping Policies for the Tagged-Token Dataflow Architecture

by

Gino K. Maa

Submitted to the Department of Electrical Engineering and Computer Science
on January 27, 1988 in partial fulfillment of the requirements
for the degree of Master of Science

Abstract

Multiprocessing seems to be the only viable way to gain significant speedup beyond that afforded by performance advances in semiconductor devices and hardware construction, which are beginning to face the limitations of physics. Although it is relatively easy to improve the "raw" computational performance of a system simply by adding more processors to it, the far more difficult task is to insure that the additional resources actually reduce a program's computing time. Thus, the ultimate success of multiprocessing as a means of increasing computation speed rests on the ability to parallelize computation: to partition, allocate, distribute, and schedule work efficiently amongst the large collection of available system resources, while maintaining a high rate of utilization of these resources.

To keep many processors busy with work, it is necessary that the program has enough concurrent operations to map to those processors. But given that there is much concurrency in many large programs, the onus lies on the code-mapping process to maintain as much concurrency as is feasible without compromising its efficiency. This goal maximizes the scalability of the system. Our study is based on a dataflow computer, the MIT Tagged-Token Dataflow Architecture (TTDA).

This study focuses on analyzing the effectiveness of the code-mapping policies for the TTDA. It develops metrics and a practical method for evaluating the effectiveness of mapping policies on a given benchmark program and proceeds to apply it to a variety of readily implementable schemes. This approach provides answers to the following important questions:

- What is the maximum gain achievable by any code-mapping strategy and, therefore, whether it is worthwhile to seek a more sophisticated strategy?
- If some of the processors are idle, does it mean that the program lacks sufficient parallelism, and therefore either should be rewritten or perhaps is entirely unsuitable for a multiprocessor, or that the code-mapping process is too inefficient?
- What is the effect of communications latency, an inherent part of all multiprocessor systems, on the performance of the system and the code-mapping strategy?

Thesis supervisor: Dr. Arvind
Title: Associate Professor of Electrical Engineering and Computer Science

Keywords: Dataflow, Multiprocessors, Code Mapping, Program Partitioning, Processor Allocation, Scheduling, Granularity, Multiprocessor Performance Scalability, Parallel Computing.

Acknowledgments

I am grateful to a host of people whose generous support and contributions made this work possible. I thank my advisor, Arvind, for offering me this wonderfully enriching opportunity to participate in a challenging, revolutionary research, and for imparting his enthusiasm, wisdom, and guidance on this work.

I am obliged to David Culler and Beruz Vafa for bringing my attention to the multiprocessor resource mapping problem initially, Ken Traub for making the Id compiler spiffy, Natalie Tarbet for generously proofreading and editing this report, and all other members of the Computation Structures Group past and present for developing the essential tools and pioneering concepts that this work bases on. I thank Andrew Chien, who has patiently put up with all my gripes along the years, and acknowledge the efforts of Andy Boughton and Natalie Tarbet, who have built the vital infrastructure that holds this research group together.

I am forever indebted to my parents and my brother, Ray, and sister, Anne, for their unconditional sacrifice, love, understanding, and support. I thank my dear friends John Chang, Mark Tapley, and Joey Chang, for their interest in my endeavors and their moral encouragement.

This research was supported by the Defense Advanced Research Projects Agency under the Office of Naval Research contract N00014-84-K-0099.

Contents

1 Introduction 1

2 Dataflow Parallel Computing 5

2.1 The Dataflow Paradigm of Computing 6

2.1.1 The Dataflow Instruction Execution Mechanism 7

2.1.2 The Parallelism Profile of a Dataflow Graph 8

2.1.3 Generating Dataflow Graphs 11

2.2 Speedup and Utilization of a Parallel Computer 12

2.3 Why dataflow is a good vehicle for studying mapping policies 14

2.4 The SIMPLE Code: an Application Kernel in Id 15

3 System Simulation Issues and Techniques 21

3.1 Program Graph and Data Value Representation 22

3.2 The Tagged-Token Dataflow Architecture 24

3.2.1 The Instruction-Set Processor Pipeline 24

3.2.2 The PE Controller 27

3.2.3 The I-Structure Memory 27

3.3 The TTDA Simulator 29

3.4 The Graph Interpreter Models 33

3.4.1 The Idealized GITA 35

3.4.2 The "Finite-Processor" GITA with Latency 36

3.4.3 The Multiple-Queue GITA 40

3.5 A Validation of GITA 40

3.5.1 Experimental Specification and Results Using the Simulator 41

3.5.2 Correlating the Experimental Results from GITA 43

4 The Effects of Code-Mapping Policies on Performance Scalability **47**

4.1 The Code-Mapping Process of the TTDA 49

4.2 Dataflow Task Scheduling 49

4.3 Task-Allocation Policies 53

4.4 Task Granularity 57

4.4.1 Choices of Task Granularity 60

4.4.2 Program Parallelism under Different Task Granularities 61

4.5 The Impact of Task-Allocation Policies on System Speedup 65

4.6 The Impact of Latency on System Speedup 68

5 Conclusion **75**

5.1 Summary of Significant Results 76

5.2 Future Research 77

List of Figures

2-1 Program Graph and Parallelism Profile for Inner Product, $n = 3$ 10
2-2 Compiler-Output Program Graph for Inner Product 13
2-3 Block diagram of SIMPLE. 17
2-4 Code-block-size Distribution of SIMPLE. 18
2-5 Parallelism Profile of SIMPLE (3 iterations, 20 × 20). 19
2-6 Parallelism Profile of SIMPLE (1 iteration, 32 × 32). 20

3-1 Representation of an Instruction 23
3-2 Split-Phase I-Structure Memory Reference 24
3-3 The Tagged-Token Dataflow Architecture 25
3-4 I-structure Memory Cells 28
3-5 I-structure Memory Controller 28
3-6 Simulation Analogue of the TTDA 31
3-7 Simulation Analogue of the Interconnection Network 32
3-8 Structure of the Graph Interpreter 34
3-9 $(n = 1000, l = 0)$ Parallelism Profile for SIMPLE (1 iteration, 32 × 32). 37
3-10 Speedup in the Presence of Latency for SIMPLE (1 iteration, 32 × 32). 39
3-11 Effect of Latency on Speedup of SIMPLE (1 iteration, 10 × 10). 44

4-1 The TTDA Code-Mapping Process. 50
4-2 Anomaly in Instruction Scheduling 53
4-3 Work Distributed to Processor Groups by a Static Allocation Policy. 58
4-4 Ideal Parallelism Profiles of SIMPLE(32) under Various Task Granularities. . . 63
4-5 Speedup of SIMPLE(32) as Computed from Its Parallelism Profiles. 64
4-6 The Efficiency and Scalability of Some Allocation Policies. 69
4-7 Effect of Latency on Speedup of SIMPLE(32). 72
4-8 Effect of Latency on Speedup of MatrixMul(20). 73

This empty page was substituted for a blank page in the original document.

Chapter 1

Introduction

Recent advances in technology have made computers smaller, cheaper, more reliable and energy efficient. These developments afford us the opportunity to exploit the potentials of networking several computers together over operating each as a stand-alone machine. An immediate benefit of networked computers is sharing of both information and resources. Data residing or programs running on one computer may be accessed by another; hardware resources such as printers or special-purpose processors connected to one computer can be made available to others. The advantage of modularity is inherent in a network of specialized computers where each computer is designed and programmed to serve a particular subset of operations that are required of the system. This modularity may simplify the development and maintenance of very complex systems. Multiple computers can cooperate to work on one large problem, with the goal of reducing the solution time significantly from that achievable with only one computer. Multiple computers may also be interconnected to provide redundancy in operations. The replicated resources can insure system availability even when some of the computers should fail. This concept can be further extended to provide fault tolerance, meaning that a system would be able to detect failure of some component, recover, and bypass the fault to continue operating correctly. These are some very attractive motivations for exploring networking and multiprocessing.

The objective of the research described here is to exploit the potential of using multiple computers to speed up the execution of a program, thus allowing large, complex problems to be solved faster and some of those problems which are beyond the capabilities of today's supercomputers to be tackled at all. Multiprocessing seems to be the only viable way to gain

significant speedup beyond that afforded by performance advances in semiconductor devices and hardware construction, which are beginning to face the limitations of physics. Although it is relatively easy to improve the "raw" computational performance[1] of a system simply by adding more processors to it, the far more difficult task is to insure that the additional resources actually reduce a program's computing time. Thus, the ultimate success of multiprocessing as a means of increasing computation speed rests on the ability to parallelize computation: to partition, allocate, distribute, and schedule work efficiently amongst the large collection of available system resources, while maintaining a high rate of utilization of these resources.

In order to exploit multiprocessing to speed up computations, there must initially be sufficient concurrency in the program to make use of redundant hardware. But the subsequent process of actually parallelizing the computation, or mapping the code to processor, must preserve enough of the initial concurrency of the program to saturate the machine with work. We shall assume that there is sufficient concurrency in large programs and concentrate on examining the latter problem in detail. Our study of the specific code-mapping issues is based on the dataflow machine and the dataflow programming paradigm because of the elegant simplicity it offers in accounting explicitly for the costs and benefits of various aspects of the code-mapping strategies.

This study mainly concentrates on the "effectiveness" part of an entire cost-effective analysis needed to determine the exact code-mapping policy for the TTDA. As such it develops metrics and a valuable method for evaluating the effectiveness of mapping policies on a given benchmark program and proceeds to apply it to a variety of readily implementable schemes. The "cost" part of the analysis of code-mapping policies is mostly neglected here, since many operating-system related implementation details have not yet been explored adequately to provide specific timing and throughput parameters needed to complete that phase.

Chapter 2 introduces the principles of dataflow computing and presents an abstract interpreter of dataflow graphs. It establishes some metrics for characterizing program parallelism under the abstract interpreter and for evaluating the performance of a multiprocessor. Then it describes a large computational physics kernel known as SIMPLE, coded in a dataflow language, Id. This kernel will be used almost exclusively throughout the rest of the dissertation.

[1]The "raw" performance being the MIPS rate of one processor multiplied by the number of processors in the system, which gives the absolute upper-bound performance that the given hardware can ever attain.

Chapter 3 documents the MIT Tagged-Token Dataflow Architecture, the target machine of this code-mapping study, and the operating conditions and assumptions under which it is conducted. We show the program development and execution processes and the simulation and emulation tools used.

Chapter 4 presents the results of this dataflow code-mapping study. Three fairly orthogonal phases can be identified in the code-mapping process: task partitioning, task allocation to processors, and instruction scheduling. Finding the optimal strategy for each of these phases is an NP-complete problem[18, 13]. Instead, we shall show, for a particular program, how to obtain the theoretical best-case scenario for each code-mapping phase, and use it as a comparison with what some realistic strategies can achieve. This approach provides answers to some important code-mapping questions, including the following:

- What is the maximum gain achievable by any code-mapping strategy and, therefore, whether it is worthwhile to seek a more sophisticated strategy?
- If some of the processors are idle, does it mean that the program lacks sufficient parallelism and, therefore, either should be rewritten or perhaps is entirely unsuitable for a multiprocessor, or that the code-mapping process is too inefficient?
- What is the effect of communications latency, an inherent part of all multiprocessor systems, on the performance of the system and the code-mapping strategy?

Chapter 2

Dataflow Parallel Computing

Many approaches have been suggested to exploit the potentials of parallel computing. An algorithm for solving a particular problem or a class of problems can be implemented directly in hardware, so that the concurrency in replicated hardware structures may be fully utilized. Recent advancements in VLSI technology and CAD (computer-aided design) tools promote this approach. They can be found from dedicated chess computers [1] to systolic arrays and FFT and other sorting networks. Alternatively, algorithms can be designed to repeat some simple, regular pattern of execution during run-time, such as a regular data access pattern. Then a machine can be instructed either to use its *vector* hardware to pipeline this repeated pattern more expediently by overlapping several iterations at once or, instead, to use many processing elements which work simultaneously, each operating on one iteration of the execution using distinct data, to complete the task quickly without repetition. Vector supercomputers and SIMD, single-instruction multiple-data, systems such as the Crays and the Connection Machine are manifestations of such an approach, which works well for algorithms whose parallelism is well organized into these distinct (in data to be processed), yet similar (in execution procedure) loops. Of course, an entire program can be manually decomposed into concurrent processes which communicate with each other via messages, as in CSP or Ada, or it can be infused with explicit *fork* and *join* operations to create multiple subprocesses and synchronize their computations. Lastly, a method for automatically exposing a large degree of unorganized parallelism inherent in most programs can be adopted to spare the engineer the task of designing special

[1] The Hitech chess computer developed at Carnegie Mellon University, for example, uses special parallel hardware to search and evaluate state space rapidly.

hardware for each problem, as in the first approach, or the programmer the task of finding the algorithm optimally suited for a particular problem/hardware combination, as in the second approach.

To this latter end we may start by adopting a functional language discipline (*e.g.*, FP [10], ML [23],) which permits one, whenever there is a function application, to evaluate all of its arguments concurrently. This parallel evaluation yields results identical to a conventional top-down, left-right sequential evaluation because the rewrite semantics of functional languages guarantees side-effect-free execution [20]. Since programs written in a functional language consist entirely of compositions of more primitive functions, there is much concurrency in most programs. One execution paradigm to exploit the parallelism offered by functional languages is a data-driven style of computing known as *dataflow* [15].

In this chapter, we introduce the dataflow paradigm of program development and execution under some abstract interpreter models, then define some important metrics such as the parallelism profile, speedup, and utilization for characterizing aspects of the program and the performance of a multiprocessor[2]. These definitions will be used throughout the rest of the dissertation. We then argue why the dataflow approach is a much more appropriate basis for attempting to understand and analyze the issues of code-mapping for a multiprocessor than the von Neumann style of programming. Lastly, we present a large kernel written in the dataflow style as the benchmark for our study. Relevant statistics such as its dynamic instruction count, instruction mix, and parallelism profile are noted for reference. Certainly, the conclusions of any set of experiments are as dependent on the choice of the benchmark programs as they are on the computer itself, but we believe the choice of a large, realistic kernel will produce a more balanced view of all aspects of the system.

2.1 The Dataflow Paradigm of Computing

Dataflow computers execute a program by directly interpreting a dataflow graph consisting of nodes, which represent machine instructions, connected by directed arcs, which represent the data dependencies, and hence the partial execution ordering, among the nodes. This section begins with descriptions of the dataflow instruction execution mechanism followed by a short

[2]As we shall see in Chapter 3, there are functions relating these program and performance metrics.

example program graph and the derivation and meaning of its *ideal parallelism profile.* Then we shall show the actual graphs generated automatically from the source program by the Id compiler.

2.1.1 The Dataflow Instruction Execution Mechanism

In the conventional, sequential computing model, computation is performed by the underlying hardware and various levels of firmware and software emulating a fetch-decode-execute cycle. First the machine fetches an instruction from the memory; it is decoded to determine which operation is needed and where its operands are. Then the operands are fetched; the specified operation is carried out on the operands and the results stored back. The cycle thus repeats to process the next instruction, which defaults to the one logically succeeding in the instruction stream. Lying at the heart of this computing model is the notion of a control sequence which is passed from one instruction to the next: the currently active instruction chooses its successor based on the state of the system. Although multi-threaded program execution can be accommodated by special fork and join operations, they require operating-system-level support because the machine-instruction level scheduling does not support the necessary synchronization and context switching to make this style of operation efficient. Inherent in this computing model also is the relation that the instruction dictates its operand fetches, so that the operand fetches are a part of the primitive fetch-decode-execute cycle and they must complete before further work can resume.

A basic machine instruction cycle begins when all the operands required by an activity are present at the processor, then the activity becomes *enabled.* The instruction corresponding to an enabled activity is then fetched out of the program memory, decoded, and executed using the available operands. The results of the operation are delivered to the succeeding activities which may cause them to be enabled for execution. In the dataflow model, instruction scheduling is based upon the availability of operand data to an activity, thus freeing from the single-threaded control structure of sequential computers. The dataflow scheduling mechanism provides machine-instruction level synchronization and context switching to provide efficient multi-threaded computation. In this model the program instruction fetch, instead of the operand fetches, becomes part of the primitive fetch-decode-execute cycle and it must complete before further work can be initiated.

If we view data and code as two distinct types of input required by a machine for execution, (*i.e.*, the Harvard architecture view,) then there is an important contrasting point worth noting: program code is static and completely defined prior to runtime. As such, it is more amenable for distribution to and caching by several processors without concerns for synchronization or consistency problems. Data, on the other hand, is continually changing (or in the functional programming paradigm, is continually being defined.) The data store must thus constantly be updated to reflect the state of the computation. Given these differences, then, it is obvious that instruction fetch should be dictated by available data and so become part of the execution cycle. Since the program can be distributed safely among processors, the program fetch will always be local and therefore immune to network access latency, resulting in a *non-blocking* processor pipeline.

2.1.2 The Parallelism Profile of a Dataflow Graph

Dataflow graphs have a well-defined meaning without any timing assumptions, and dataflow execution is generally viewed as asynchronous. Nonetheless, in order to simplify the characterization of parallelism in dataflow programs, we shall consider a synchronous execution model.

The *parallelism profile* for a dataflow graph on a given input is a function $pp(t)$ which gives the number of instructions executed at each step t on an abstract machine. The abstract machine has the following characteristics:

- All enabled instructions are immediately executed.
- Each operator takes unit time to execute.
- Results of an instruction are available to its successors instantaneously.
- Unbounded computation resources.

We illustrate the method for generating parallelism profiles through an example. Figure 2-1 shows a program graph which computes the inner products of two vectors, `A` and `B`, of size `n`. Initially, a token corresponding to **sum** with value zero is input to the left switch and a token corresponding to `i` with value one is presented to the right switch and the $\leq$ predicate. The value `n`, as well as descriptors for the two vectors, are *loop invariants* and thus can be

considered to be embedded in the graph. Assume that the value of n is 3. In step 1, instruction 1 (*i.e.*, the instruction $\leq$,) fires, producing tokens with value TRUE for the control inputs of the two switches. Instructions 2 and 3 fire in the second step and produce the tokens carrying the value of **sum** for instruction 8 and i for instructions 4, 5, and 6. In step 3, instructions 4, 5, and 6 execute while the token for **sum** waits for the other input to instruction 8. Firing of instruction 6 provides input to the predicate and the right switch. In step 4, the value of **sum** that has been waiting is added to the result of instruction 7 while the new value of i passes through the switch. Note in step 5 of the parallelism profile in Figure 2-1 that the second iteration has begun while the first is still active. Execution continues in this manner until step 14 when it produces a token on the false side of the switch in instruction 2. The pattern in steps 4 to 6 covers all eight instructions and repeats for every iteration. Note, a node is fired at the step corresponding to the maximum of the times of its input tokens.

The step beyond which $pp(t)$ is uniformly zero is called the *critical path length*, that is, the length of the longest chain of data-dependencies in the program. The area under the curve $pp(t)$ gives the total number of operations executed. The ratio of these is the average parallelism. It can be seen from the parallelism profile of the inner-product n = 3 shown in Figure 2-1 that the critical path length is 12 and the total number of operations is 27. Note, this describes the parallelism for this particular method of computing the inner-product and does not imply that this is the maximum achievable parallelism. However, other methods with more parallelism would be described by different graphs.

The fine-grained scheduling underlying a dataflow computer treats each instruction as an independent process, complete with its own context and environment. We define a single *thread of computation* as a set of instructions which can never execute concurrently. In Figure 2-1, instructions 1, 3, 4, 7, 8, for instance, constitute a thread of computation and instructions 4, 5, and 6 are in different threads of computation. The number of *active threads of computation*, then, is the number of instructions competing for processor usage at a given time[3]. This notion is analogous to that of the *concurrent processes* for sequential computers.

[3]This is only true if pipeline effects are ignored. Given a pipelined machine, the actual number of competing tasks would be the number of active threads of computation minus the number of pipeline stages.

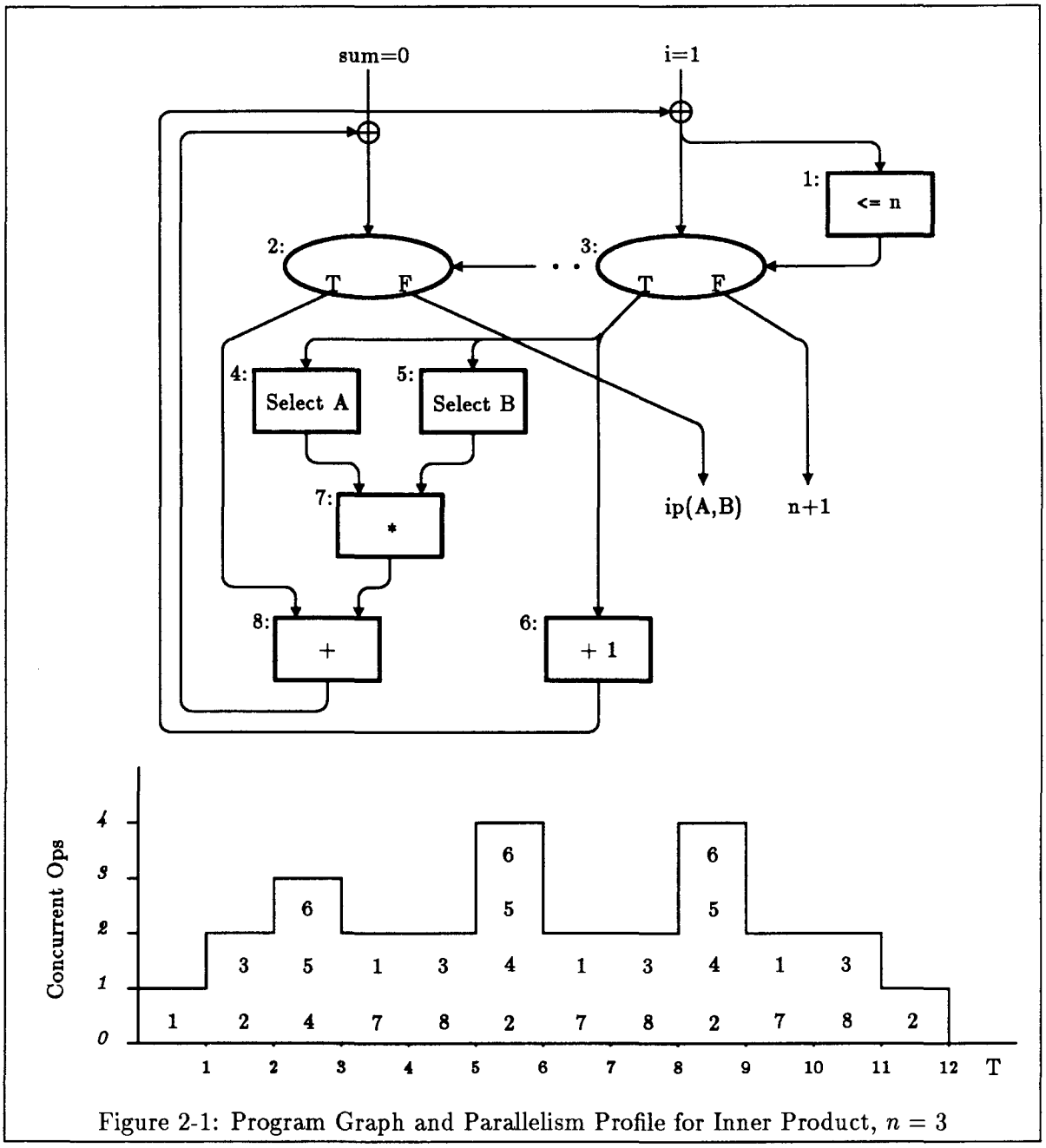

Figure 2-1: Program Graph and Parallelism Profile for Inner Product, $n = 3$

2.1.3 Generating Dataflow Graphs

Although dataflow graphs provide a precise representation of parallel execution, the utility of the model is limited by our ability to generate the graphs themselves. We shall consider only graphs generated by a reasonably sophisticated compiler for the language Id. Id is a functional language extended with I-structures to provide efficient array manipulation. A broad class of algorithms are easily expressed in Id, although it is difficult, perhaps impossible, to express certain types of algorithms in Id efficiently. This section explores the implications of the language and the compiler on parallelism profiles.

The inner product function may be expressed in Id as follows.

```
Def ip A B = {(l,h) = bounds(A) ;
              s = 0
              In
                {For j From l To h Do
                   Next s = s + A[j] * B[j]
                 Finally s }} ;
```

The dataflow graphs produced by the Id compiler are based on a fixed set of schema and rules for composition [27] which ensure deterministic behavior under all execution orders. Arithmetic expressions and let-blocks are described by acyclic graphs. Conditional expressions are constructed using switch instructions to steer values to the appropriate arm, based on a predicate, allowing portions of the conditional to execute before all the inputs of the expression are present. Iteration is captured by a loop schema, which permits arbitrary overlap of iterations, unless constrained by data dependencies. Values which are arguments to an invocation of a loop but are constant over all iterations are not circulated, but are explicitly stored in a constant area [8] in the loop preamble. User-defined functions and loops are compiled into code-blocks, which are invoked by an application schema, permitting arbitrary recursion. The class of graphs generated in this manner is deterministic and self-cleaning [6]. Furthermore, graphs are embellished so that each code block receives a trigger to enable nodes with constant input and produces a signal indicating that all nodes without outputs have fired [27].

The graph generated by the Id compiler for `ip`, the inner product program, contains 31 instructions. Most are for setting up and cleaning up the loop and are thus executed only

once. The graph for ip shown in Figure 2-2, though stylized, captures the essential features of the compilation for drawing the parallelism profile. The output of the Id compiler can be executed on the TTDA simulator or GITA [25], the Graph Interpreter for the Tagged-token Architecture, which generates parallelism profile graphs as a part of its runtime statistics reports. As can be seen in the profile in Figure 2-2, the compiler-generated graph executes five more instructions per iteration than the graph in Figure 2-1. These additional instructions are generated for tag manipulation and control of loop unfolding, which are discussed further in [2]. Graphs generated by the current Id compiler incur roughly 150% overhead in terms of the number of instructions executed beyond the essential computation in order to allow maximal parallelism [16] while preserving determinacy.

The dataflow graph for a program is divided into units called *Code-Blocks*. Typically there is a separate code-block for each separately compiled function and for each loop.

2.2 Speedup and Utilization of a Parallel Computer

Some basic concepts and definitions of metrics must first be established before the performance of a multiprocessor can be evaluated. One of the most important metrics for a multiprocessor system is its *speedup* in running a program, defined as

$$S(n) = \frac{t(1)}{t(n)},$$

where $t(n)$ is the execution time required for a configuration of n processors. The speedup factor predicts the throughput of the system as a function of its hardware size and also the degree to which its performance can be cost-effectively scaled up, simply by modularly increasing the hardware. $t(1) \leq nt(n)$; otherwise, we shall be able to emulate the multiprocessor by n-way interleaving the uniprocessor's instruction stream with the n-way partitioned code and reduce $t(1)$ thus.

The fact that during the course of a computation some processors may be idle gives rise to the concept of the *utilization* or *efficiency* of a multiprocessor system,

$$\eta = \frac{S(n)}{n} = \frac{t(1)}{nt(n)}.$$

In general, $\eta < 1$ because of synchronizations, both explicit and implicit. Explicit synchronizations are those dependencies dictated by the program logic — the flow, anti, and output

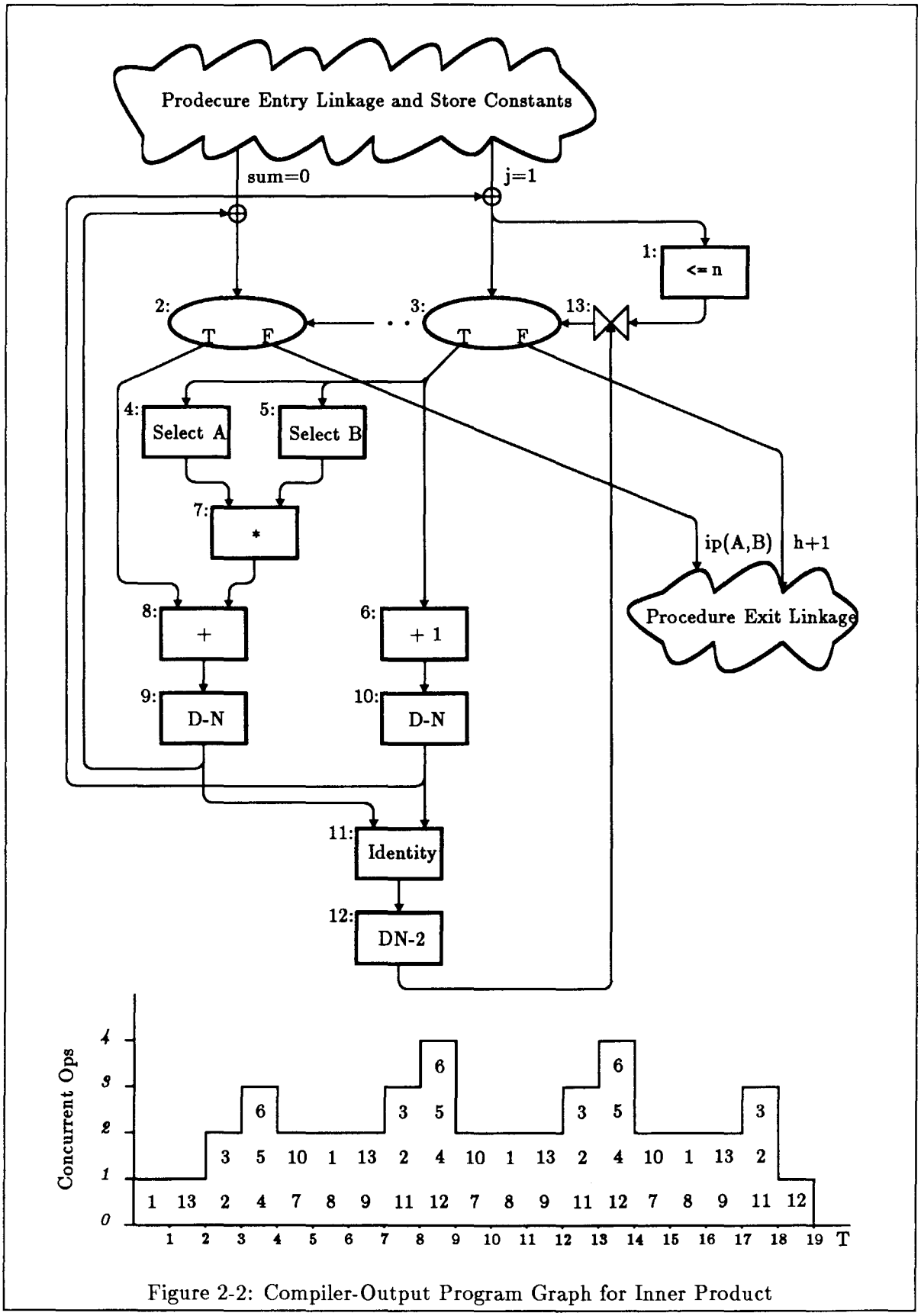

Figure 2-2: Compiler-Output Program Graph for Inner Product

dependencies described by Kuck in [22], and the macroscopic-level data structure producer-consumer relationships. In von Neumann parallel computing, this cost is often ascribed to the overhead of parallelizing or decomposing an algorithm. But in dataflow computing, this is more properly unified under the overhead of synchronization. Of course, there may be large variations in the amount of explicit synchronization among different algorithms computing the same problem. We assume that the problems amenable to multiprocessor solution are either inherently parallel or some parallel algorithm can be found. Implicit synchronizations — bus contention or collision in the routing network and memory bank access conflicts — are the artifacts of the architectural specification and hardware implementation. These synchronizations force processors to defer execution of certain instructions until their prerequisites are met.

2.3 Why dataflow is a good vehicle for studying mapping policies

Dataflow is an excellent vehicle for studying code-mapping policy issues confronting multiprocessor systems because the costs and benefits of each choice at each phase of the code-mapping process can be accounted for explicitly and thus evaluated easily. Unlike parallel von Neumann programs, which are sequential code annotated with extra instructions to switch contexts and to synchronize or fork computation, dataflow programs perform context switching, synchronization, and forking at every instruction. Each dataflow instruction executes entirely context independent of all other instructions in the program, its results determined solely by its inputs. Efficiency arguments aside for now, this scheme already includes the full cost of parallelizing a program, so that when task granularity is varied or more processors are added to the system, no further instructions or work is needed to run the program in the new configuration. Any gain in execution speed or loss in processor utilization can be readily attributed either to a lack of parallelism in the program itself, or the algorithm, or to an inefficient code-mapping strategy. Most von Neumann parallel programs would have to be recompiled and many even completely restructured and rewritten if they were to run on a system with, for instance, ten times more processors. The very high cost of forking and joining parallel processes and synchronizing computation on von Neumann computers, in addition to the tedious explicitly-specified decomposition, results in these programs' being parallelized only as much as they are necessary for the target system. This approach, unfortunately, destroys the metric for any

valid scalability comparison so that we are not able to discern the effects of program parallelism and code-mapping strategies in accounting for scalability anomalies.

By using the dataflow parallelism profile, it is possible to quantify precisely the amount of parallelism available in the program itself, as we have already seen. As the program is actually mapped onto a set of processors and executed, its characteristics can be determined at each phase of the mapping process. Since no more instructions are executed[4] nor is the program structure altered, the cost and effectiveness of that phase can be evaluated in terms of the amount of parallelism it preserves.

2.4 The SIMPLE Code: an Application Kernel in Id

In order to characterize the behavior and performance of a computing system, we must unfortunately introduce a highly variant agent, the test or benchmark program. Any choice of benchmark program necessarily biases the emphasis placed on particular aspects of the system design and therefore influences the outcome of the experiments. We eschew most small, single function kernels: inner product, linear combination, FFT, convolution, *etc.* , precisely because they tend to spotlight particular features and thus give a more artificial characterization of the system as a whole. In searching for a candidate, we considered complete application kernels which are mostly computation-bound — in particular, the computational physics problems which account for a large proportion of the load of many supercomputers in service. These problems need to solve numerically a set of partial differential equations, which describe a certain physical system, usually by either the finite-element, finite-difference, or Monte Carlo methods. A large program based on each of the finite-difference and Monte Carlo methods has been written in Id as benchmarks for the TTDA. Since both the finite-difference and the finite-element methods are reduced to a large sparse matrix inversion problem, the finite-element method has not been represented. We have chosen the SIMPLE code, an instance of the finite-difference method, as the primary benchmark for this study. It is a large, 1200-line program with complex data and control structures, exhibiting non-trivial producer/consumer relationships[5].

[4]This means that potential speedup is directly proportional to the average parallelism.

[5]The Monte Carlo benchmark is an electrodynamics application (PIC) using a particle-in-cell approach.

The SIMPLE code solves a two-dimensional hydrodynamics and heat conduction problem. It has been studied extensively both analytically [14] and empirically. It has become one of the standards for benchmarking high-performance computers, so performance data from many commercial and research systems exist. A detailed discussion of the program appears in [5][6]. Its top-level structure, main subcomponents, and dataflow dependencies are depicted in Figure 2-3. With the exception of the procedures **world** and **generate**, which set up the program constants and generate the initial conditions, the entire program is iterated once for each simulated time step. Each iteration starts with calculating the positions of the border zones surrounding the grid boundary by reflecting the adjacent interior point across the boundary, and by calculating their physical attributes (*i.e.*, mass density, viscosity, and pressure.) The problem is then partitioned into a hydrodynamics and a heat conduction phase. During the first phase, the velocity and position of the nodes are incremented based on the acceleration vector at each node. Then, new values of area, volume, and density at the new positions of the nodes are computed along with their intermediate values of pressure and temperature. The second phase transfers energy between adjacent zones to account for the heat conduction. This involves solving a system of linear difference equations by the alternating direction implicit (ADI) method to determine the final pressure and temperature of each zone. For a grid size of $n \times n$, both of the major phases consist of doubly nested loops generating $O(n^2)$ work for each simulated time step.

Figure 2-5 shows the parallelism profile of three iterations of the outer loop (three time steps) of SIMPLE on a 20×20 mesh, while a typical simulation run would perform 100,000 iterations on 100×100 mesh. The critical path is 4371 and the instruction count is 2,338,792. The profile of the first iteration is biased by the initialization procedures. But as can be seen from the identical latter two of the three distinct sections of the profile, there is no significant parallelism among the outer loop iterations (*i.e.*, successive time steps) of SIMPLE; the profile for n iterations can be obtained by repetition of a single-iteration profile. Table 2.1 lists a breakdown of instruction mix classes for the same run. To show how the profile changes with the size of the problem, we have drawn the profile for the 32×32 mesh in Figure 2-6. (Critical path = 2393, instruction count = $2,207,156$.) It is noteworthy that the potential parallelism

[6]This reference actually describes a version of the program written in a more recent dialect of Id, Id-Nouveau [24]. Except for its more flexible syntactic style, much of the semantics of the program still remains the same as, or has close counterparts in the original Id version that is used throughout this study.

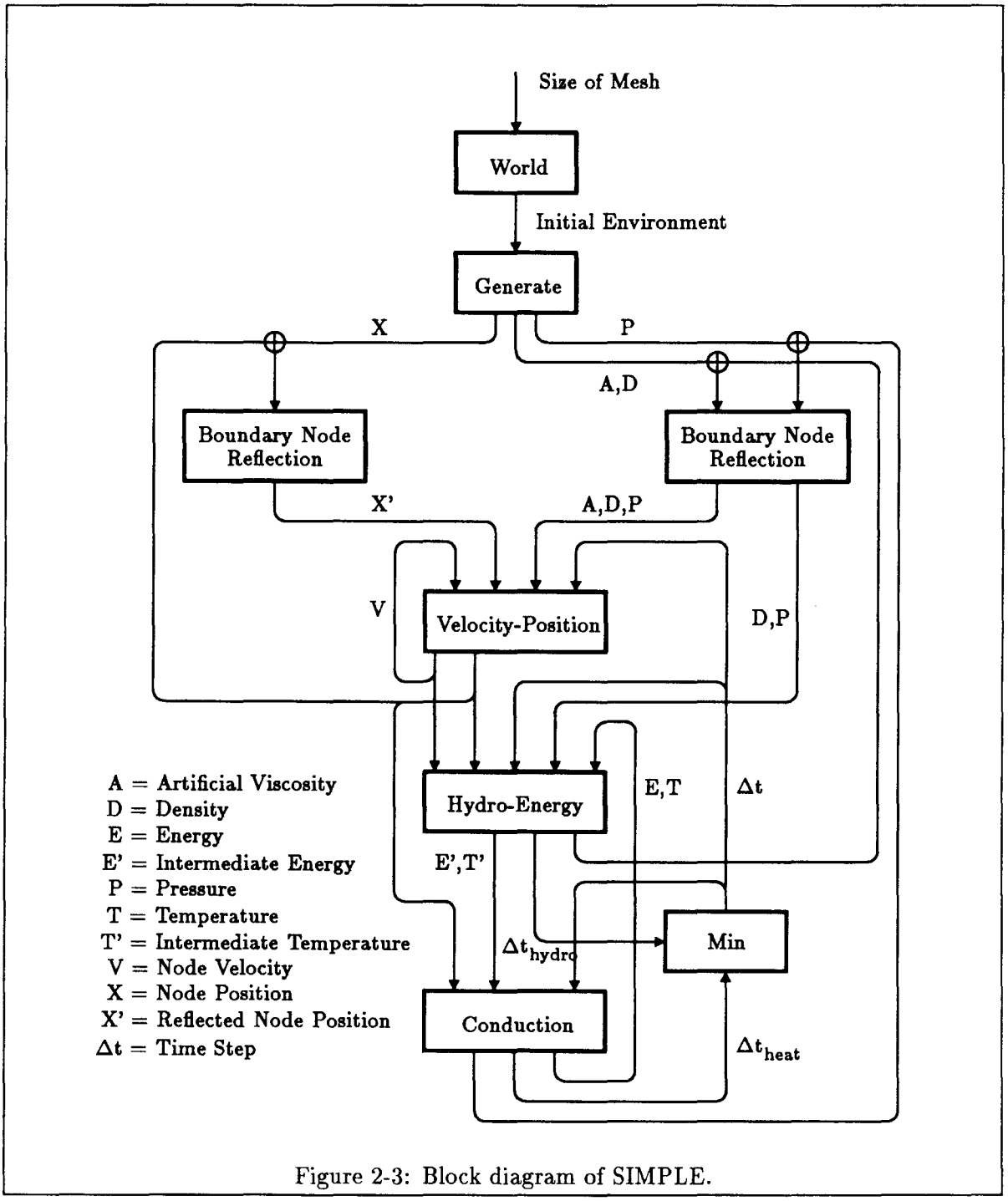

Figure 2-3: Block diagram of SIMPLE.

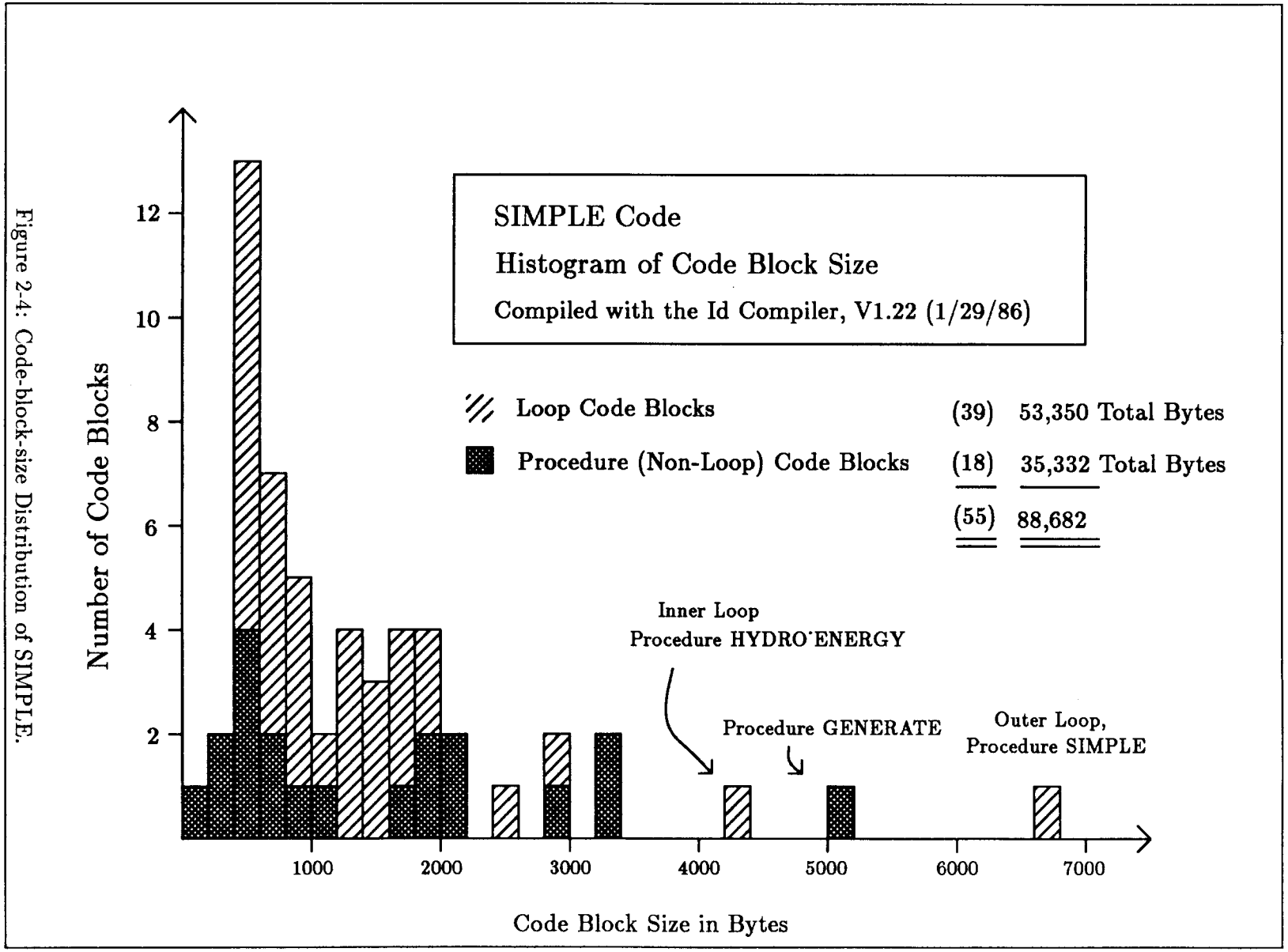

Figure 2-4: Code-block-size Distribution of SIMPLE.

varies tremendously during execution, a behavior which in our experience is typical of even the most highly parallel programs. We believe that any large program that runs for a long time must have sufficient parallelism to keep hundreds of processors utilized; several applications that we have studied support this belief.

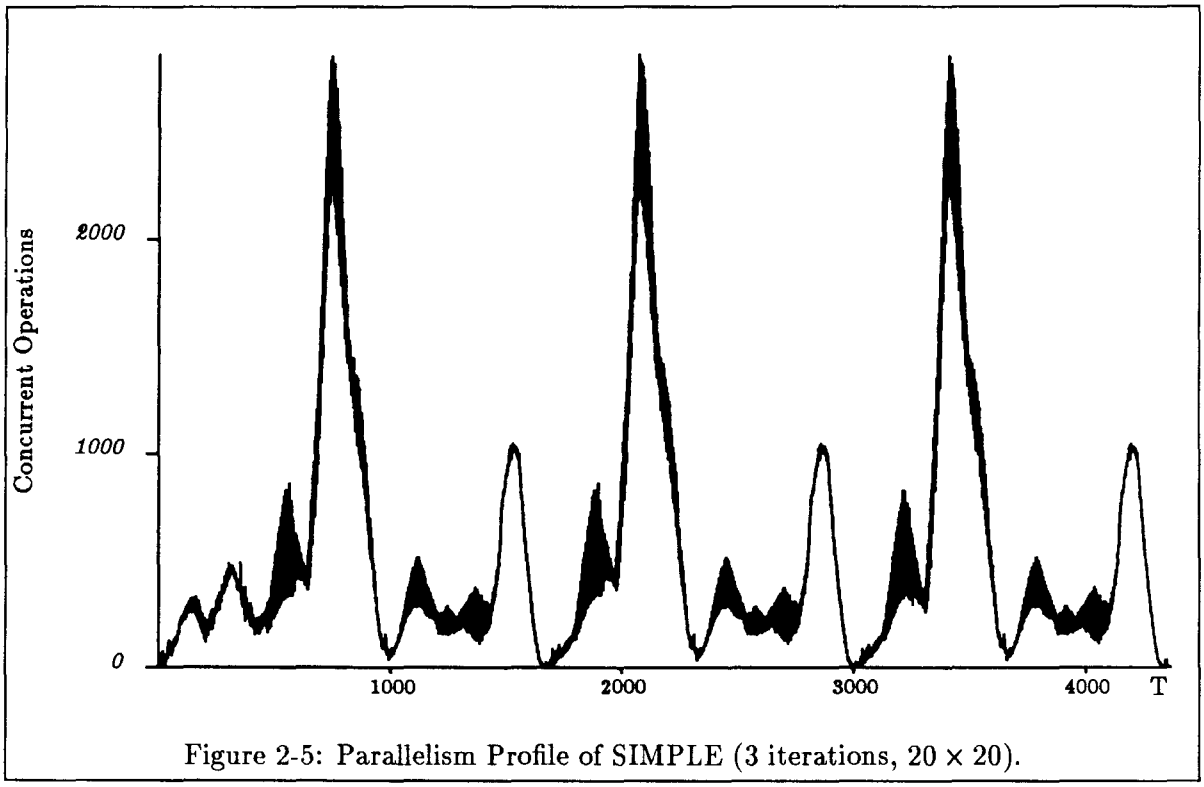

Figure 2-5: Parallelism Profile of SIMPLE (3 iterations, 20 x 20).

Type	Count	Percentage
IDENTITY	643,520	27.52%
I-FETCH	622,746	26.63%
FP-ARITHMETIC/LOGICAL	313,893	13.42%
I-STORE	180,430	7.71%
INT-ARITHMETIC/LOGICAL	168,522	7.21%
MISC-TAG-AND-CONTROL	98,190	4.20%
SWITCH	87,301	3.73%
EXPAND-COMPRESS	84,193	3.60%
DEC-RC	65,490	2.80%
USE	34,496	1.47%
D	32,232	1.38%
SUPER-ARITHMETIC	2,403	0.10%
MISC-I-STRUCTURE	798	0.03%

Table 2.1: Dynamic Instruction Mix of SIMPLE (3 iterations, 20 x 20).

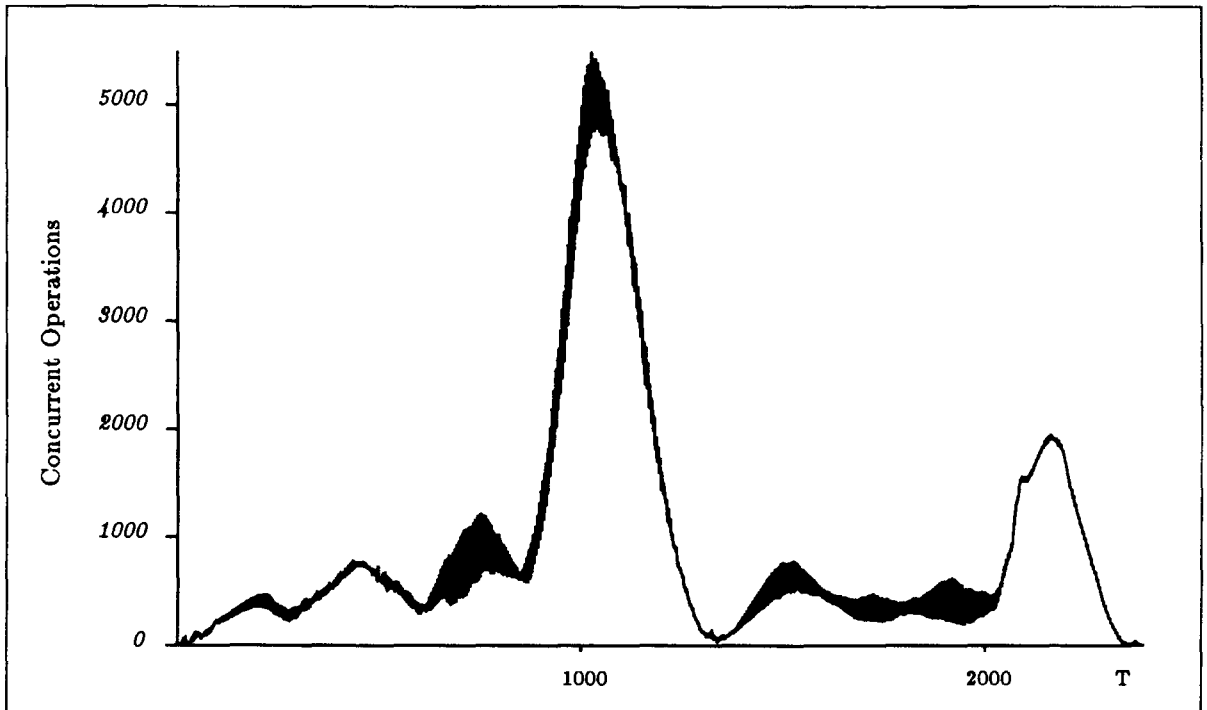

Figure 2-6: Parallelism Profile of SIMPLE (1 iteration, 32 x 32).

Chapter 3

System Simulation Issues and Techniques

Developed by the Computation Structures Group at MIT, the Tagged-Token Dataflow Architecture (TTDA) [3] is a multiprocessor system based on the dataflow principle of program execution. It is a mechanization of the U-interpreter abstract model of computation [6]. Its distinguishing feature, compared to Dennis's pioneering static dataflow architecture [15], is that every token in the machine is tagged with a field which encodes the *activity name*, or the context, of the token. It exposes more parallelism in the graph by enabling tokens from different contexts to coexist on any arc of the graph, which has the identical effect as replicating the actual graph once for each active context.

The TTDA is a conceptual prototype dataflow system; it has never seen an actual hardware implementation. It is intended to guide our investigation into the requirements and rewards of dataflow computing and to focus our attention on the relevant issues. The definition of the TTDA includes a complete specification of its machine instruction set, the various token and data types and value representation, the processor pipeline structure and the functional specifications of each stage, and the I-structure memory controller operations. With this definition it is possible to construct high-level-language compilers producing dataflow machine instructions and instruction-set interpreters emulating or simulating candidate processor designs.

Although the instruction processing aspect of the TTDA has been fully specified, the higher-level operations issues of code-mapping policies have not been explored and those of resource

management [1, 2] have not been settled. In order to study the behavior and investigate these undecided issues and to evaluate the merits and deficiencies of the TTDA, before actually committing it to some hardware realization, some emulation facilities have been constructed to allow these experiments to be performed. We shall first present the encoding of the dataflow program graphs and data tokens, and then a block-diagram-level description of the TTDA, which processes instructions and data in these representations. We then discuss the salient features and justify the design decisions of two emulation tools: the TTDA simulator, which faithfully follows the proposed hardware functional specification of the TTDA, and several variants of GITA, the graph interpreter, which is an abstract implementation of a TTDA instruction interpreter. Both of these will be used later to conduct experiments in evaluating code-mapping policies for the TTDA. Finally, we shall verify GITA as a valid simulation tool for the TTDA by establishing a correspondence between the results from the two facilities, so that GITA results can be cited with confidence, and normalized and meaningfully compared to the simulator results.

3.1 Program Graph and Data Value Representation

A dataflow source program is translated into a set of *code-blocks*, each of which corresponds to an Id function or a loop construct. A code-block is a piece of dataflow graph with some number of inputs and outputs. The graph for the code-block is encoded as a linear sequence of instructions to be stored in the Program Memory. The address for each instruction in the linear sequence is chosen arbitrarily[1]. Instructions in the graph are encoded as shown in Figure 3-1. The *literal/constant* field may be a literal value or an offset into the constant-area. The *destinations* are merely the addresses of the successor instructions in the graph. To facilitate relocation, addressing within a code-block is relative to the beginning of the code-block.

A specific invocation of a code-block is determined by a *context*, which identifies two registers: the *Code-Block Register* (CBR) which points to the base address in Program Memory for the code-block's instructions, and the *Data Base Register* (DBR) which points to the base address in Constant Memory for the constant area in Constant Memory. At code-block invocation time, the system manager must therefore allocate space in Program and Constant

[1] Except for certain conventions as to where input tokens to a code-block are received.

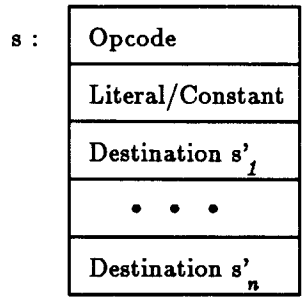

Figure 3-1: Representation of an Instruction

Memory for the designated function and allocate and initialize a CBR/DBR pair to point to the instruction- and constant-base addresses.

There are two classes of tokens in the TTDA: data-value tokens and system-services tokens. The value tokens correspond exactly to the tokens traveling along the arcs of the dataflow program graph. Each value token has a tag field and a data field. The tag of each value token specifies an instance of an instruction activity to which it is destined. Each tag consists of five sub-fields:

- Processor number — the destination processor.
- Context register number — the context of the token.
- Instruction offset — instruction within the code-block.
- Token Arity — whether token needs a partner.
- Operand port number — which instruction operand the token is for.

The data field consists of a type tag and an encoding of the value itself.

The system-services tokens have no counterpart in the dataflow graph; they are manifestations of the architectural implementation. These appear in I-structure operations (for example an I-fetch, which is actually a split-phase transaction as shown in Figure 3-2) as the request and reply/acknowledgment tokens traveling along the dashed arcs between the I-structure memory units and the program instructions. Other instances are system-manager requests (*e.g.*, function invocation/termination, memory allocation/deallocation.) System-services tokens carry processor- or memory-controller-specific information.

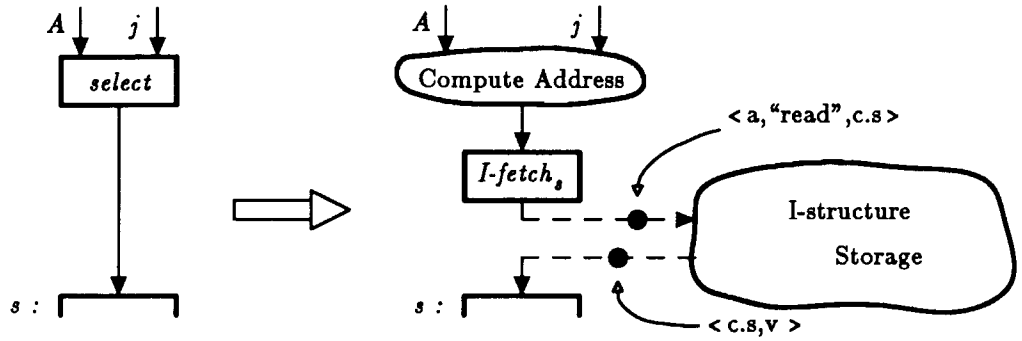

Figure 3-2: Split-Phase I-Structure Memory Reference

3.2 The Tagged-Token Dataflow Architecture

The Tagged-Token Dataflow Architecture incorporates a set of processing nodes interconnected by a short-distance, high-bandwidth packet-switching communications network. Each node consists of a processing element (PE) and an I-structure memory module, and the packets that are routed by the network consist of dataflow tokens or service request and acknowledgment tokens. Although this underlying implementation uses message-passing communications protocols, the uniform address space of the system and the machine-instruction-level packetization presents only a shared-memory architecture to the programs. The specifications of the network are deliberately left nebulous in the TTDA definition in order not to preclude any innovative developments in network topology and design.

As shown in Figure 3-3, the processing element is a dataflow computer which is radically different in organization from its standard von Neumann counterpart. Within each PE, there are three parallel paths for incoming tokens: the instruction-set processor, the processor-controller, and the I-structure memory.

3.2.1 The Instruction-Set Processor Pipeline

The instruction-set processor branch comprises a pipeline of asynchronous stages coupled by staging FIFO buffers:

Input Stage: An input token is extracted from the incoming packet and checked to determine whether it is a data token, an I-structure memory request, or an auxiliary service request/acknowledgment token. For data tokens, those which do not require a partner to

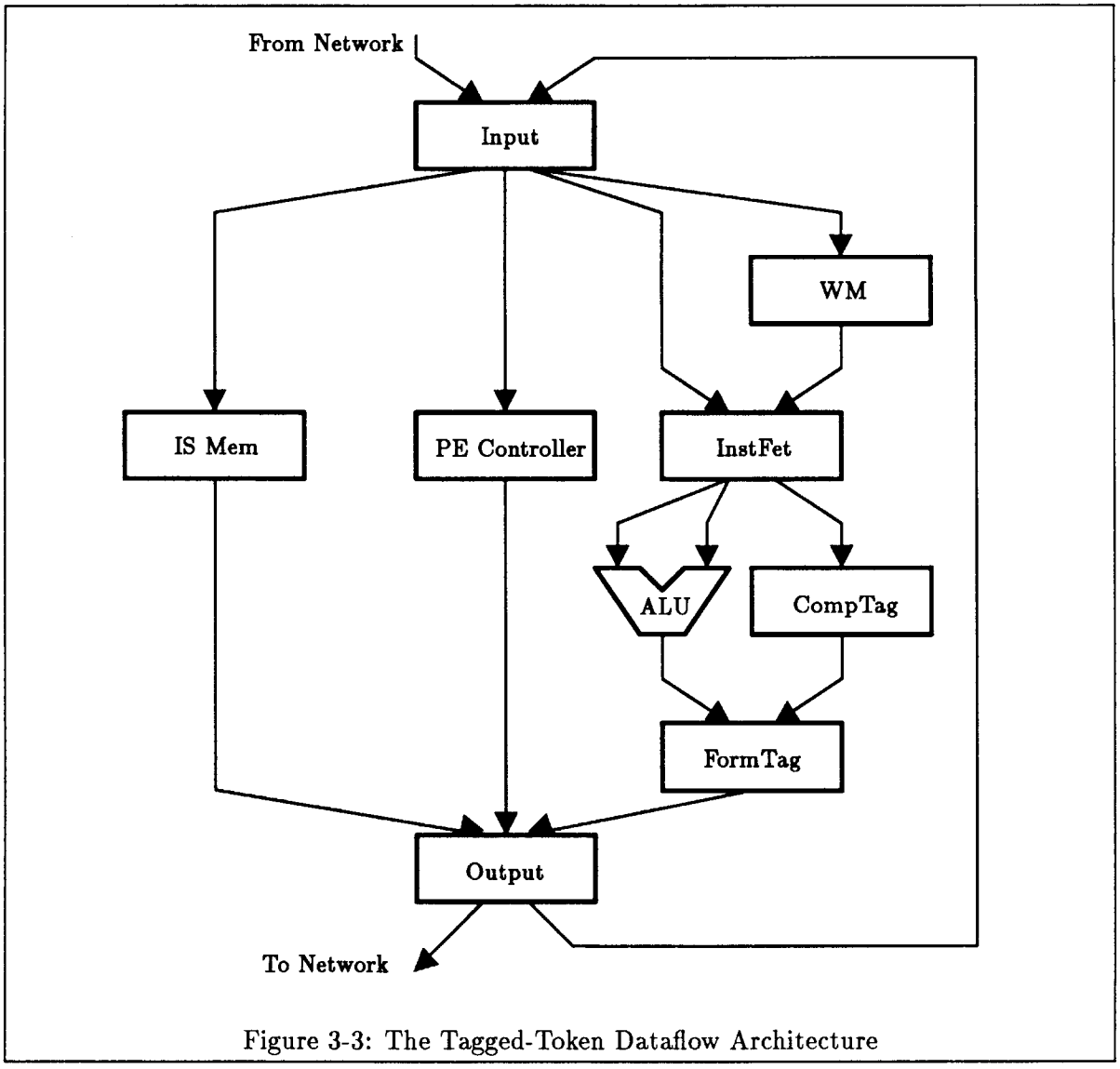

Figure 3-3: The Tagged-Token Dataflow Architecture

become active[2] are passed directly to the instruction fetch stage, while those which do will enter the wait-match stage to seek its partner. I-structure memory requests are forwarded to the I-structure memory controller stage and auxiliary service request/acknowledgment tokens are forwarded to the PE controller.

Wait-Match Stage: The associative token store is searched in attempt to locate the partner of the input token. Partner tokens have identical tag field, except for the instruction port number. If a partner is found, then it is extracted from the store and the pair of operands are then forwarded to the instruction fetch stage. If a match cannot be found, then the incoming token must wait in the wait-match store for its partner to arrive eventually. The Wait-Match unit serves two distinct functions: It provides the hardware mechanisms for synchronization by deciding whether to store or fetch an operand, and it provides for the actual storage of the operands. Obviously, the wait-match associative store must have sufficient capacity to hold all incoming unmatched tokens, otherwise the overflowing tokens can never hope to get matched and thus exit out of this stage. The scenario potentially leads to a deadlock of the system.

Instruction-Fetch Stage: The context base register and the destination instruction address in the tag of the token are used to fetch the instruction from program memory. The tokens' data fields are sent to the ALU and their tags are forwarded to the compute-tag stage, along with the necessary instructions to control the processing function of these two succeeding stages.

ALU: The workhorse of the pipeline. It receives its operands and function codes from the instruction-fetch stage.

Compute-Tag Stage: The tag sub-fields of the tokens are manipulated, according to the type of executing instruction, to produce output tags that will direct the result tokens to their intended destinations.

Form-Token Stage: The results of the ALU and compute-tag stages are combined as the data and tag fields in building the output tokens.

[2]Note that there is no correlation between the arity of an operator and whether a token destined to that operator requires a partner. This is because a dyadic operator may have a constant operand and a monadic operator may need a signal trigger. See the instruction-set definition [7].

Output Stage: The results from the three branches of the PE are serialized to be delivered to the network.

3.2.2 The PE Controller

The PE controller branch serves to provide a "backdoor" access to the entire state of each processing element to facilitate diagnostics and block-code and -data transfer to or from any of its internal memory units. It also provides basic input-output capability, and non-deterministic merging for stream and resource management operations.

3.2.3 The I-Structure Memory

Memory operations in the TTDA are all split-transaction processes: a request for memory service is sent to the memory module to initiate a transaction. An acknowledgment is eventually delivered to the sender to complete the memory operation. The salient characteristic of this type of memory access cycles is that even though there may be an arbitrary time delay between the request and its corresponding acknowledgment, the processor issuing the request is immediately freed to start work on another thread of computation.

The I-structure memory [9] supports this split-transaction memory operation by requiring each request to specify a *continuation*, a forwarding destination, so that upon completion of the operation, the results or an acknowledgment can be delivered to the forwarding address, and further processing can resume on that thread of computation.

To synchronize the read and write requests to a memory location, the I-structure memory controller manipulates the *presence bits* that are associated with each location, as in Figure 3-4. The presence bits indicate the state of a memory cell: *present, absent*, or *deferred.* Writing into an *absent* location puts it into the *present* state, and future reads would obtain copies of the written content as expected, although any writes would cause an error. Reading from an *absent* location puts it into a *deferred* state, in which all read requests are remembered, until the write data arrives. A write into the *deferred* location will, in addition to setting it to the *present* state, also have to reply to each entry of the pending-request list. A simplified state transition diagram for the I-structure memory controller is in Figure 3-5. The hardware enforcement of the write-once behavior of the I-structure is consistent with the single-assignment semantics of Id, and is necessary to guarantee determinacy of computation.

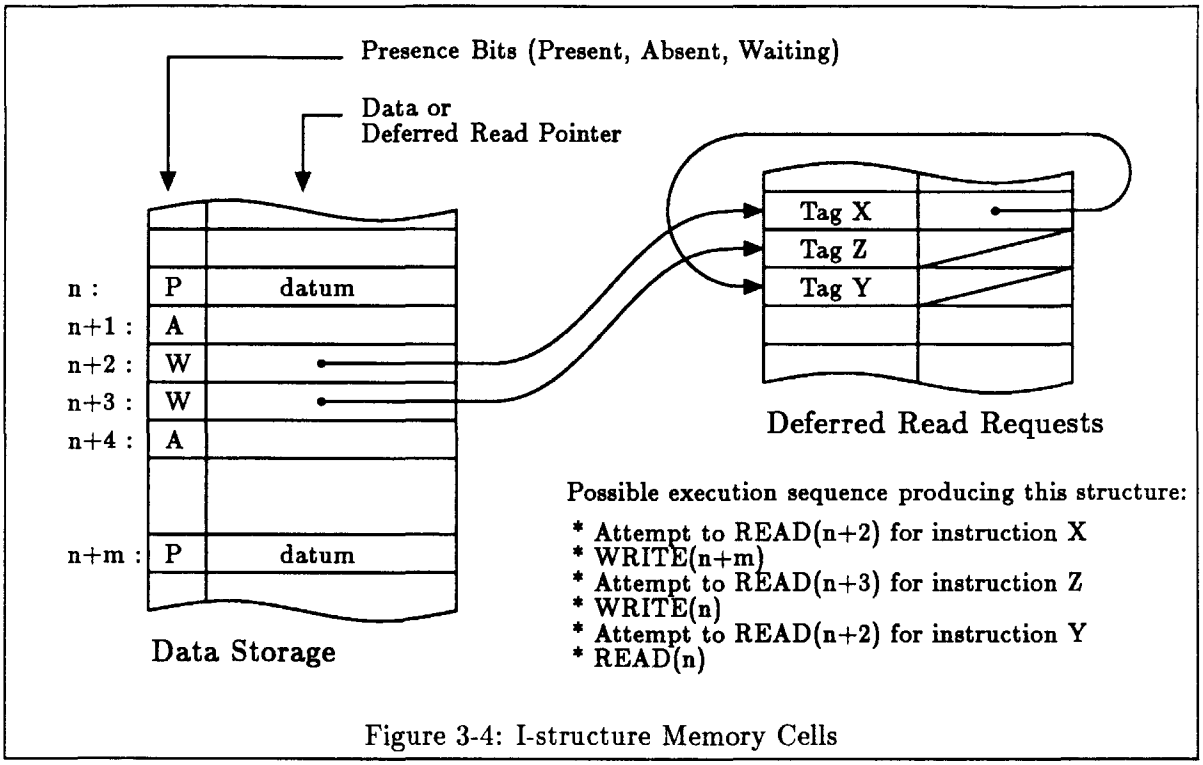

Figure 3-4: I-structure Memory Cells

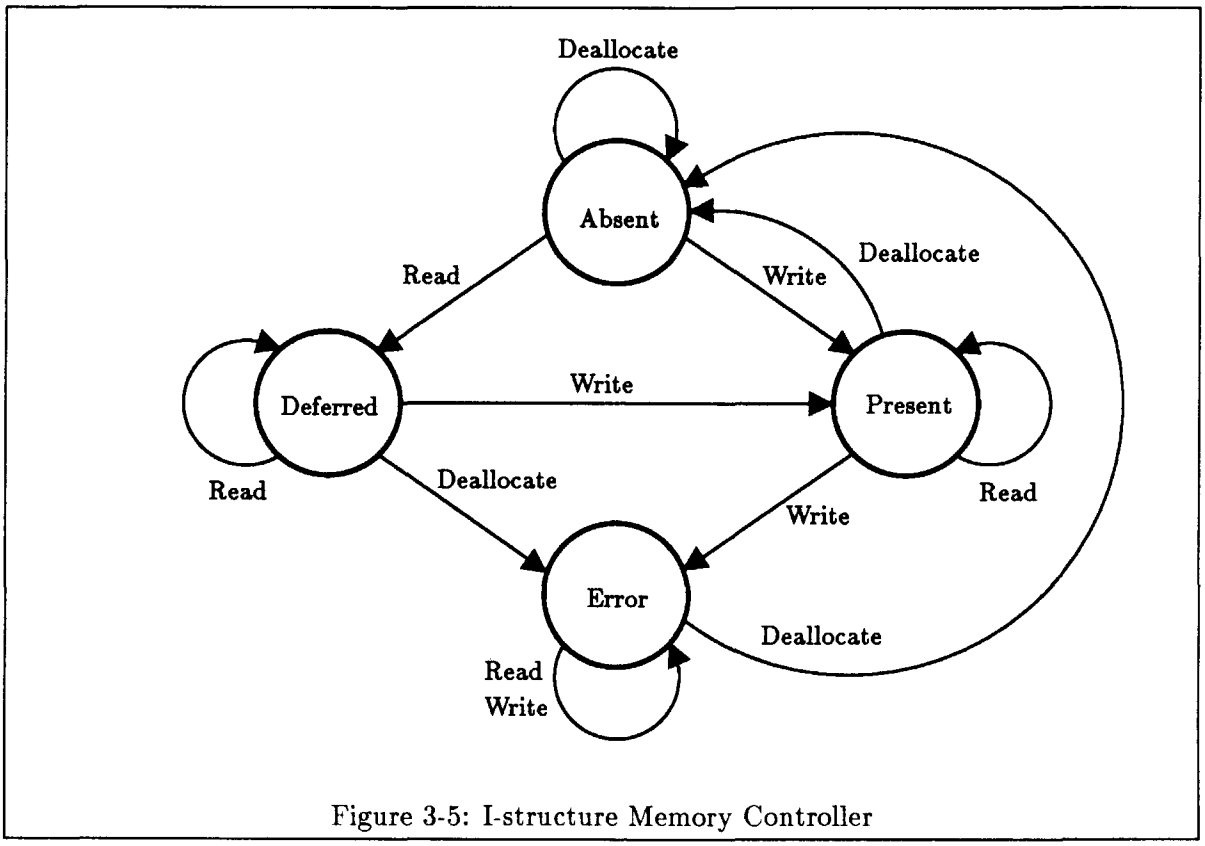

Figure 3-5: I-structure Memory Controller

The I-structure memory implementation does incur some overhead compared to a conventional memory system. The operational overhead includes the space needed to remember the deferred-read list and the extra processing time when a deferred-read is satisfied. If we assume unit time accounting, then each read demands one τ of work to get a response value or to be placed on a list. Similarly, a write normally requires one τ of work to update the memory location, plus, if there is a deferred-read list, one more τ to respond to each pending read. Hence, if all reads becomes deferred, then there is a 100% degradation in throughput, whereas when no reads are deferred, then the I-structure memory performs on a par with conventional memory. The management overhead includes the costs of initializing, allocating, and deallocating blocks of I-structure memory. Conventional memory also needs such management work, but since each of its locations can only be written once, the I-structure memory probably needs to be recycled much more often. Reference counts or other storage reclamation methods are needed to recycle inaccessible cells.

3.3 The TTDA Simulator

The simulator models, at the register transfer level, the time behavior as well as the input-output behavior of the system and executes the TTDA instruction set described in [7]. It simulates a system of dataflow processors and I-structure storage elements, communicating through a multistage (butterfly) interconnection network of size $n \times (1/2)\log_2 n$, where n is the number of processors plus the number of storage elements[3].

There are three fundamental aspects to the simulation: modeling the functional components, modeling the finite intermediate staging buffers, and scheduling the activities throughout the system. At the core of the simulator is an event-driven scheduler. It schedules time-stamped packets to be processed by a collection of target-architecture-dependent *stations* and *finite buffers*, each of which increments the timestamps and may also update the contents of the packets according to its functional descriptions. When a buffer becomes full, it blocks upstream stations from producing output and the scheduler from enabling those stations. The

[3]This facet of the simulator differs from the TTDA machine configuration, in which a processing element *and* an I-structure module are packaged together as a node of the network, making all nodes homogeneous. The separation of the memory from the processor is intended to facilitate studies of memory capacity and throughput requirement issues. The original case of a one-to-one coupling of memory to processor can be easily simulated by this more general scheme.

simulated analogue of the architecture is described in Figure 3-6. The ovals correspond to the hardware components and are the *stations* in the simulation. The boxes correspond to the *finite buffers* on the inputs to the various components. The resource manager in the simulator is currently modeled as a hardware component. A precise description of the construction and operation of the simulator's scheduler and finite buffers, and full functional and timing specifications for its *stations* are found in [11].

The aspect of the system specification that is of particular relevance to our central topic of code-mapping is the interconnection network, so its design choice needs some further justification. For the purpose of the following discussion, interconnection networks can be divided into two broad classes according to topology: static and dynamic. Static topologies (excluding complete connections, *i.e.*, full cross-bars,) such as tree, mesh, and n-cube networks, provide different length geodesic paths between different pairs of nodes, thus giving rise to "neighborhoods" within the network. (The n-neighborhood of a node is the set of all nodes which have a path with length not more than n from the given node.) The locality among nodes imposed by these networks can conceptually be used to guide the mapping policy to make better use of the available bandwidth by assigning highly interacting tasks to neighboring processors. But if the network topology were ignored by the mapping policy, the usable network bandwidth could be drastically reduced by forcing a large fraction of packets to traverse a great many links to arrive at their destinations. Dynamic topologies, such as shuffle-exchange, baseline, banyan, and butterfly multistage networks, route directly from any node to any other node without going through a third node. Routing between any pair of nodes in these networks takes equal effort and expends equal amount of network resources.

Although the TTDA definition gives no specifications for the design and topology of the communications network, the simulator's processor and I-structure storage elements are all interconnected to each other by a multistage packet-switching network. A multistage butterfly network is chosen as a representative from the dynamic topology class. Consequently, the mapping policy can no longer attempt to match locality in programs with locality in network topology, resulting in non-optimal communications bandwidth and latency. Although it shall later be argued that optimal mapping for a static topology is not likely to be achievable anyway, we shall for now assume the network has enough bandwidth to support the computation, which is not difficult as long as the network traffic distribution is fairly random. Further, we contend, and shall also later demonstrate, that the TTDA tolerates a large degree of communications

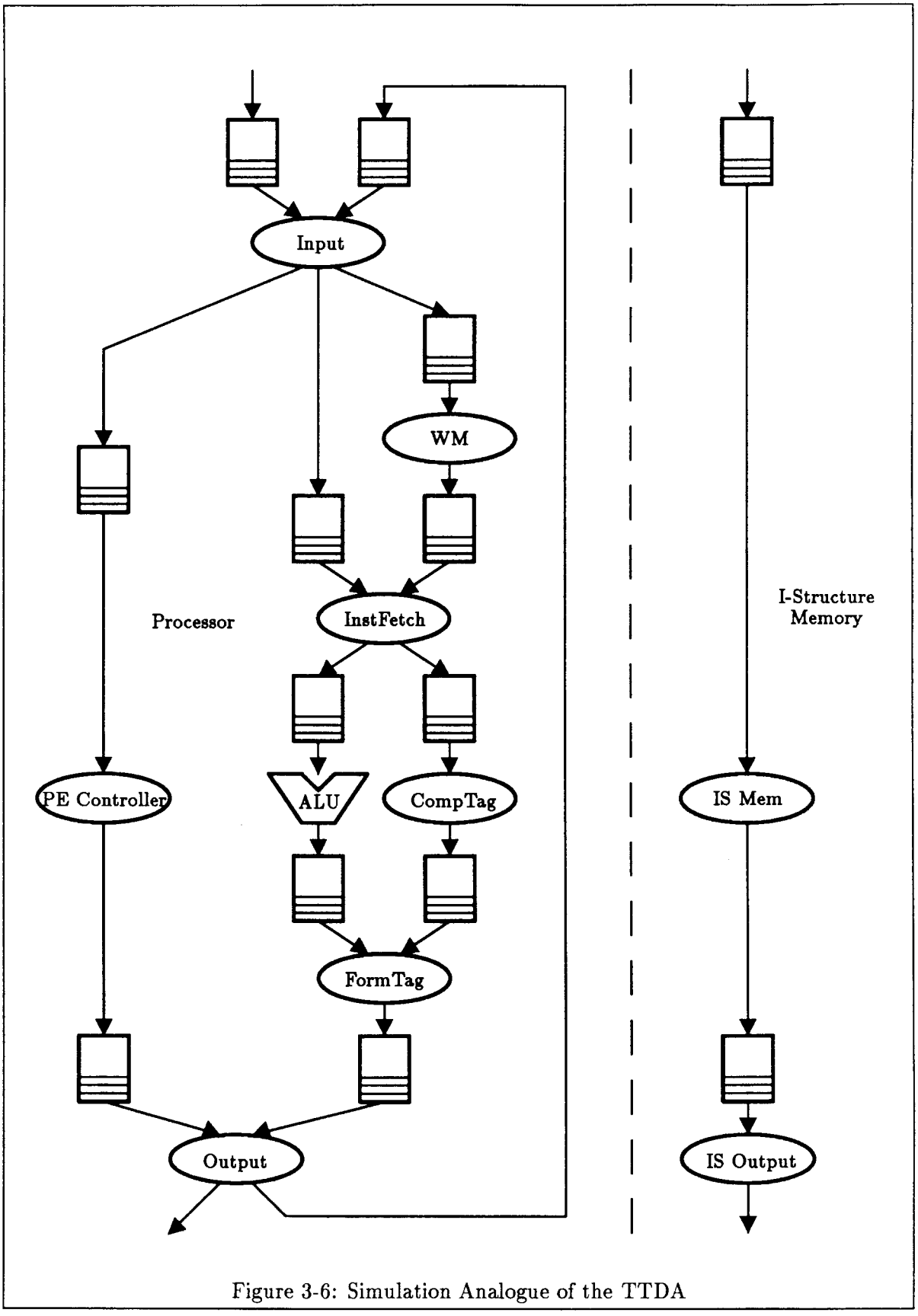

Figure 3-6: Simulation Analogue of the TTDA

latency without incurring a significant performance loss. Given these premises, the selection of a multistage butterfly network, also known as an FFT network, as the network model for the simulator would seem appropriate. It consists of $\log_2 n$ stages of $n/2$ 2×2 cross-bar switches, where n is the total number of nodes (*i.e.*, the sum of the number of processing and storage elements.) Figure 3-7 shows an 8-node network and the simulation analog of each of its 12 2×2 switches. The *packet-switching* attribute refers to the store-and-forward switch operation, which mandates data buffering in the switches.

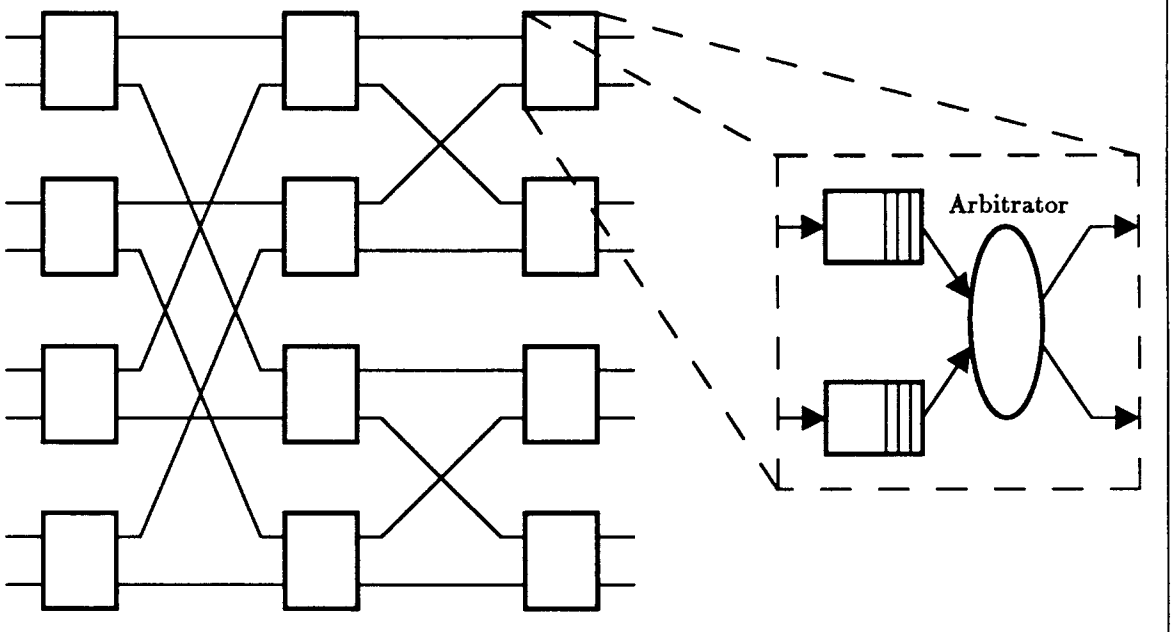

Figure 3-7: Simulation Analogue of the Interconnection Network

The simulator, including data collection instrumentation and user interface facilities, consists of over 70,000 lines of Pascal code. The level of simulation detail forces the simulator to demand a remarkable amount of resources: due to the size of the available address space (12 Mbytes) of the IBM 4381/VM host and the simulation speed ($\approx$80 ops/sec), the practical upper limit for the simulated system size is around 16 processors and for the SIMPLE kernel is about one iteration on a 10 by 10 grid, which generates just over 200,000 dataflow instructions. The intent is to observe the behavior of a relatively small workload[4] on a fairly small system and to extrapolate the probable behavior of a life-size problem running on a full-scale TTDA

[4]The computation itself is by no means more trivial; the *small workload* only refers to the reduced number of iterations of the algorithm in order to limit the simulation time and resources needed.

multiprocessor. We believe in the validity of such extrapolation of the results implicitly when using the simulator because the dataflow paradigm of execution gives us a very clean way of decomposing a program, without having to rewrite the code or to introduce extra instructions or any other overhead save that of the traffic in the network. Fortunately, the bandwidth of a multistage interconnection network is automatically scaled up linearly as the number of processors and storage elements is increased. Note also that its latency, however, grows as $O(\log n)$, which is a primary reason why it is essential that a multiprocessor must be able to tolerate long latency.

3.4 The Graph Interpreter Models

GITA [25], the Graph Interpreter for the Tagged-token Architecture, is designed to execute a dataflow program graph by tracing the progression of data-value tokens along its arcs, modifying their contents as specified by the operator nodes in accordance with the semantics of the TTDA instruction set. It does not try to simulate the operations of a specific machine implementation model, as the TTDA simulator does. Rather, its only goal is to produce result values given the program graph and input values, and to do so quickly enough and to maintain sufficient context information for program debugging as to be useful for program development. GITA operates in one of two modes: idealized or "finite-processor" mode.

GITA consists of three operating components: a TTDA instruction interpreter, an I-structure memory, and a systems manager, each being fed by a FIFO queue of tokens, organized as in Figure 3-8. The top-level interpreter loop scans the queues for tokens, then calls the appropriate operating component to process the token. Often, this process generates output tokens, which are then placed onto the respective queues. The queues are strict FIFO, thus enforcing an eager FIFO instruction scheduling policy.

In contrast to the TTDA simulator, GITA is much less involved with architectural details, thus enabling its operations to be greatly simplified and accelerated. Supported further by the Multiprocessor Emulation Facility (MEF) [4], which links up to 32 Lisp machines together to alleviate both time and storage constraints, GITA can execute much larger dataflow programs than the simulator is able to handle. There are some fundamental differences in the design of GITA, however, that are irreconcilable with the characteristics of the simulator, which make a

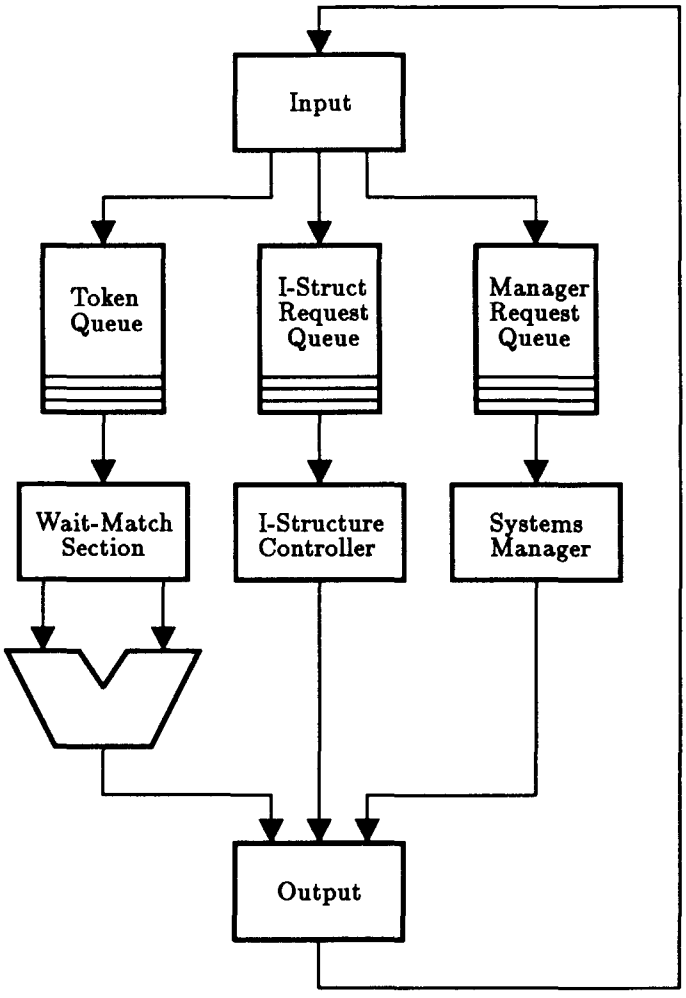

Figure 3-8: Structure of the Graph Interpreter

direct comparison of their results difficult, if not impossible: the execution engine of GITA is not pipelined and there is no concept of a network. There is only one token queue in GITA and its "finite-processor" mode merely adds a counter to restrict the number of operations that can be executed in one time step, although it places no bounds on the number of wait-match operations per time step. The execution time is accounted for in an abstract *step time*, which charges the same for an identity operator as it does for a floating-point function. I-structure accesses and manager requests are implemented differently and are also only charged one step time. Because of this step-time accounting, the "performance" results are not meaningful by themselves, but only significant as ratios, or speedups, in going from m to n processors. But with an augmentation to this basic interpreter, we shall show in Section 3.5 that the multiple-queue GITA can be a very useful tool in studying resource mapping issues in the large, for which the simulated environment necessary for the experiments to be meaningful would completely overwhelm the capacity of the simulator.

3.4.1 The Idealized GITA

To study some global aspects of the program graph, a simple timestamping scheme is used to provide such information as the critical path and the parallelism profile of a program run, both of which are concepts defined previously. This scheme models an ideal scenario based on the following assumptions:

- All enabled tokens are immediately processed in the current time step.
- Each operation takes unit time to execute, *i.e.*, all result tokens produced in the i-th time step are labeled $i + 1$.
- There is no communications delay for delivering result tokens to their destinations.
- Unbounded resources are available.

With the exception that we are counting tokens instead of instructions[5], these assumptions exactly mirror the specifications of the abstract machine defined in Section 2.1.2. The critical path, then, is simply the largest of all timestamps issued and the parallelism profile the number of tokens carrying each distinct timestamp. Since the full program graph is dynamically

[5]There is either one or two tokens for every dataflow instruction.

spliced from individual code-block graphs based on runtime conditions, these global characteristics cannot be computed from the static compiler output graphs. The instrumentation on GITA also collects the dynamic instruction mix and other execution statistics: wait-match and I-structure storage usage, I-structure reference frequency, and profile of active contexts, none of which, because of the ideal timing assumptions, necessarily reflect corresponding statistics from a real machine accurately. For actual resource usage issues, the TTDA simulator is still the penultimate validation vehicle, next to some real hardware; although often the GITA statistics do agree, at least qualitatively, with the simulator's results. The parallelism profile obtained from GITA, however, is enormously useful in characterizing a program and explaining its observed runtime behavior on a realistic system. It provides a means by which the properties of the programs can be isolated from the properties of the machine when analyzing the performance of a benchmark system.

3.4.2 The "Finite-Processor" GITA with Latency

The first step towards a realistic execution model is to bound the amount of available processing resources. Instead of allowing an arbitrary number of tokens to be processed during each step, a maximum limit of n tokens during each step is imposed. This represents n processors only in a very abstract sense. We do not assign activities to processors, but rather choose up to n input tokens in each step. It can be viewed as representing a dataflow system where the mapping of activities on to processors is perfect in that there can never be a situation where n tokens are ready, but not all of them can be processed because some are on the same processor. In addition, we introduce communications latency by assuming that the output of every instruction takes l steps to be delivered to its destination. This is consistent with the view that activities are distributed arbitrarily over all the processors. This "finite-processor" latency model has the following attributes:

- *Not more than* n tokens are processed per step,
- *"Fair" (FIFO) scheduling.*
- Each operator takes unit time to execute.
- *Fixed communications delay* for delivering result tokens to their destinations.

- Unbounded resources are available.

Obviously, for a program whose parallelism profile is less than n throughout, this restriction will not alter its behavior at all. But when it exceeds n, then the effect of execution on n processors may be visualized by drawing a horizontal line at n on the parallelism profile and then "pushing" all the tokens which are above the line to the right and below the line. Figure 3-9 shows the profile for SIMPLE, with the same parameters as those used for Figure 2-6, generated under this model with $n = 1000$, slightly greater than the average parallelism. The length of the critical path is increased from 2393 to 3252.

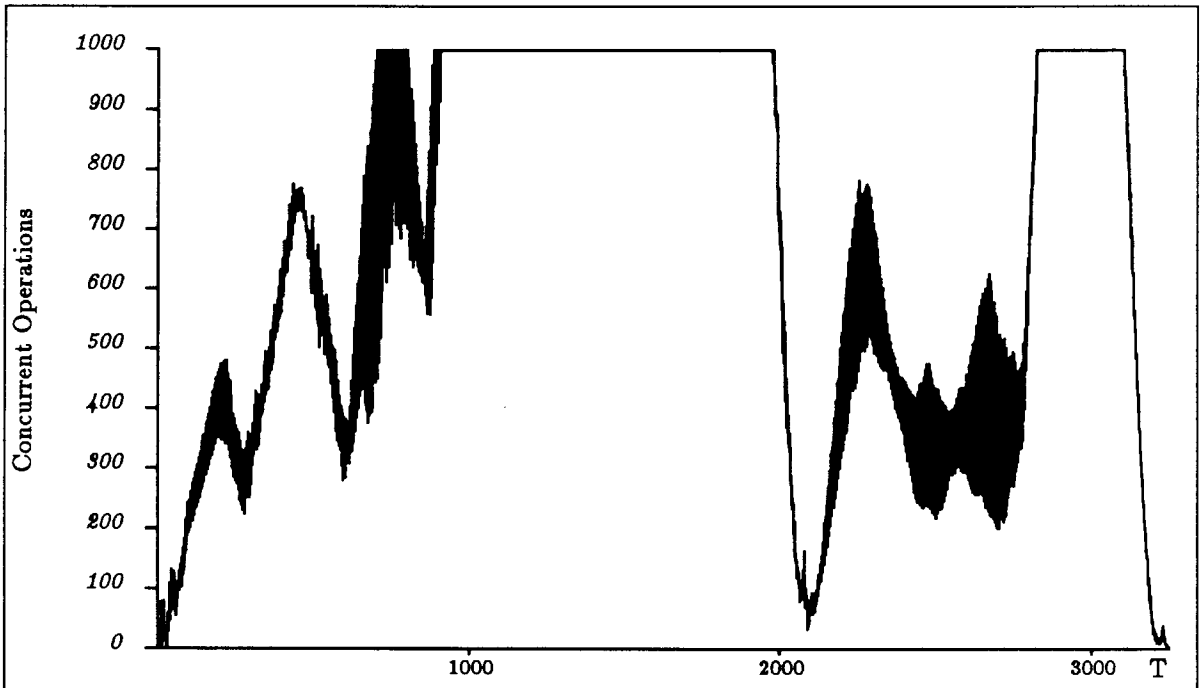

Figure 3-9: ($n = 1000, l = 0$) Parallelism Profile for SIMPLE (1 iteration, 32 x 32).

The "finite-processor" model is somewhat imprecise as a subset of the input tokens must be chosen in each step and different choices may result in different profiles. A "fair" choice is assumed, by which we require that if activity S1 precedes S2 under the ideal model, then it does so under any "finite-processor" model. Instruction scheduling will be addressed in Chapter 4.

The discussion thus far has assumed no communications latency. When latency is introduced in the model, the results of an I-structure operation (for example an I-fetch, which is a split-phase transaction as shown in Figure 3-2) is available at the destination node $2l + 1$ units after the later of the I-fetch and the I-store for the particular elements.

Speedup and Utilization

The amount of parallelism available in a program in the context of the "finite-processor" model can be expressed in terms of speedup and utilization as follows. Let $t(n,l)$ be the number of steps required to execute the program with at most n tokens per step and fixed communications latency of l units.

$$S(n,l) = \frac{t(1,0)}{t(n,l)} \qquad\qquad \eta(n,l) = \frac{t(1,0)}{nt(n,l)},$$

$t(1,0)$ is simply the total number of tokens processed, *i.e.*, the area under the parallelism profile. These numbers indicate the limits to improved performance imposed by data dependencies in the algorithm itself, modulo influences of instruction scheduling. For example, for 3 iterations of SIMPLE (20×20), $S(100,0) = 97$, and $\eta(100,0) = 97\%$. Thus, even on an idealized machine, *i.e.*, one with instantaneous communication and synchronization, it is not possible to utilize all the processors all of the time.

Estimating Speedup on Finite Machines from Parallelism Profiles

Without actually running the program repeatedly under the "finite-processor" model for each n and l, an estimate for $t(n,l)$ can be made from the ideal parallelism profile as follows. For any τ, if $pp(\tau) \leq n$, we process all $pp(\tau)$ tokens in one step. However, if $pp(\tau) > n$, then we assume it will take the least integer greater than $\frac{pp(\tau)}{n}$ steps to process $pp(\tau)$ tokens. If $l \leq \left\lceil \frac{pp(\tau)}{n} \right\rceil$, then computation of $pp(\tau+1)$ may start immediately after the last token for $pp(\tau)$, because the earliest results from $pp(\tau)$ have already arrived at their destinations. Otherwise, it must wait l steps from the start of the first n operations of $pp(\tau)$ to begin computing $pp(\tau+1)$. This estimate is optimistic, for consider if all the tokens of $pp(\tau+1)$ depended on the last operation performed for $pp(\tau)$. Assuming that dependencies are uniform, then

$$t(n,l) = \sum_{\tau=1}^{t(\infty,0)} \max(l, \left\lceil \frac{pp(\tau)}{n} \right\rceil), \tag{3.1}$$

where $t(\infty,0)$ is the length of the critical path. Figure 3-10 shows the speedup curves derived in this manner for SIMPLE. The points in the figure show the speedup measured under the "finite-processor" model for various settings of n and l.

As these curves show, the estimate of $t(n,l)$ tends to be conservative because some of the operations from the next step may be performed along with the last few operations of $pp(\tau)$.

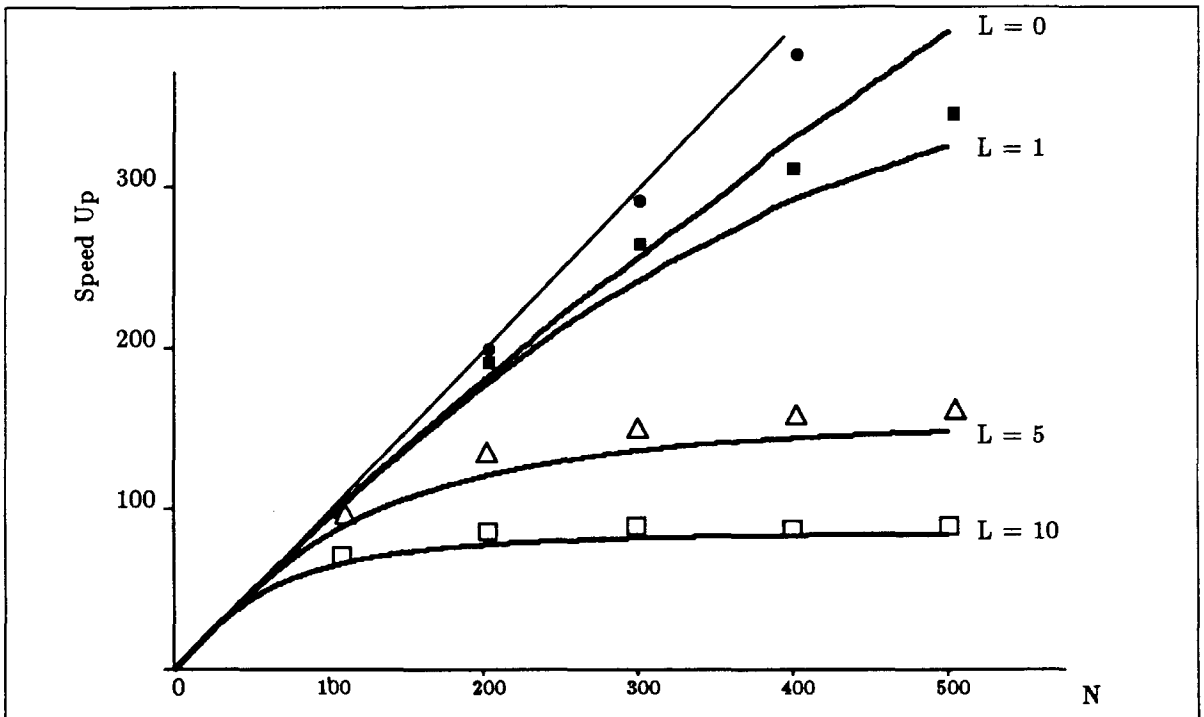

Figure 3-10: Speedup in the Presence of Latency for SIMPLE (1 iteration, 32 x 32).

The dynamic instruction scheduling capability of dataflow machines can be partially accounted with the following, more optimistic formula.

$$t(n,l) = \sum_{\tau=1}^{t(\infty,0)} \max(1, l, \frac{pp(\tau)}{n}).$$

"Finite-processor" execution gives $t(1000,0) = 2763$ and $t(100,0) = 15,176$, while the estimation technique shows them to be 2551 and 15,120, respectively. The accuracy of the estimate is better at lower values of n, which is the range of greater interest. Note that the speedup estimation equation and the "finite-processor" GITA model both provide us functions for mapping parallelism profiles into speedup and utilization quantities.

For realistic architectures the latency is determined by the depth of the processor and storage-controller pipeline, as well as the number of stages in the communications network, which is expected to grow as the log of the number of processors. This suggests that we should consider fairly large latencies. At the same time, since our data comes from an artificially small problem size, we should consider a fairly small number of processors, and extrapolate to realistic problem sizes on more processors.

3.4.3 The Multiple-Queue GITA

One of the fundamental differences, cited earlier in the introduction, between GITA and the TTDA simulator is that GITA has only a single token queue, with which the only way to model an n-processor system is its rather artificial "finite-processor" mode. It is thus unable to simulate the necessary environment for experimenting with code-mapping issues at all, and of course unable to match the behavior of the TTDA, as will be demonstrated shortly. A solution is to incorporate multiple token queues in GITA and then apply the following operational rules:

- *At most one token per step per queue.*
- "Fair" (FIFO) scheduling *within each queue.*
- *No more than n queues may execute at each step.*
- Unit time per operation.
- Fixed communications delay for delivering result tokens to their destinations.
- Unbounded resources.

The multiple-queue model requires an accompanying mapping scheme to assign tokens to queues, the crux of which is the main topic of Chapter 4. This model, with the appropriate extensions, will be the basis for much of the experimental support cited for our study of the code-mapping issues. The immediate implication of the multiple-queue model is that it can no longer exploit parallelism within a queue in ways that the original GITA is able to do. Several enabled operators within a queue can at best be fired in sequence, but not concurrently as GITA is allowed, although it is still able to use the parallelism to mask latency effectively. This restriction is, of course, also present in the simulator and in a real parallel machine.

3.5 A Validation of GITA

GITA is much more ideal and abstract than a realizable implementation of the Tagged-Token Dataflow Architecture. Before the results obtained on GITA can be accepted with any degree of certainty as those actually reflecting the behavior of a TTDA machine, it must first be validated against the simulator, the penultimate vehicle for verifying all conjectures about the

TTDA. The following experiment to show the TTDA's tolerance of an extraordinary amount of communications and memory access latency is thus devised to show also that GITA, without actually modeling the processor pipeline and the network structure, can still provide veritable predictions of the TTDA's behavior. It is therefore an important link in establishing the validity of the discussions of the subsequent chapter, whose supporting results are all drawn from experiments based on GITA rather than the TTDA simulator.

3.5.1 Experimental Specification and Results Using the Simulator

In an attempt to verify the effects of communications and memory access latency on the performance of the TTDA, the TTDA simulator is used to run SIMPLE and the collected run-time statistics are analyzed to assess the extent to which the processor utilization rate can be sustained in the presence of large latencies. To simulate varying amounts of access latency, delay elements are inserted at the inputs to each processor and storage element. The effect of this arrangement is that by inserting k delay elements, each access (*i.e.*, a read operation or a write plus an acknowledgment) is actually deferred by $2kt_{delay}$ since it involves two passes through the network. To avoid pipeline imbalances caused by mismatched timing which could produce certain congestions in the network arising only as a result of inserting these delay elements, t_{delay} is always set equal to the network switch's arbitration time. The artificially induced latency affects only network-bound tokens, which are related to I-structure access or manager requests, but not to the majority of the tokens which circulate, via an internal data path, back to the input of the local processor pipeline. For a run of SIMPLE, the ratio of network-bound to locally-circulated tokens is approximately one to four.

The timing parameters for the simulation are set so that all stations have 1τ of processing time and 1τ per word of memory access. This has been done to simplify the analysis of the results of the simulations slightly, and to facilitate comparison with the GITA results. Assuming that $\tau = 100nsec$, for example, means that each processor in the model at best could process 10 million tokens per second, or, assuming the ratio of monadic to dyadic instructions is approximately one to one for SIMPLE, around 7.5 million dataflow instructions a second. Since the processor pipeline itself is 6τ for dyadic instructions and 5τ for monadic instructions, which bypass the wait-match station, the average latency incurred by every token would be

(for large k, the number of unit-delay elements inserted)

$$\begin{aligned} L &= \frac{3}{4}5.5\tau + \frac{1}{4}(2k + 5.5)\tau \\ &= (5.5 + \frac{k}{2})\tau \\ &\approx \frac{k}{2}\tau \end{aligned}$$

The code-block allocation policy selected is round-robin, with all domain sizes set uniformly to one. This of course means that a code-block instance is executed entirely on one processor and that iterations of a loop all unfold on the same processor. The I-structure allocation policy and access method selected is the interleaved-address mode, which maps consecutive I-structure addresses horizontally across all the storage elements. This scheme better equalizes allocation and utilization among the storage elements as compared to the old round-robin vertical allocation in which each I-structure array is allocated entirely on one processor.

The results of the simulation runs are most succinctly presented as a set of speedup curves, shown in Figure 3-11. The horizontal scale is the number of processors actually participating in computing the task; the curves are sampled at two, four, six, eight, twelve, and sixteen processors. The vertical scale is the performance of the simulated system in million dataflow operations per second, based on the timing parameters given to the simulator. Although the program was never run with only one processor because each simulated processor lacks sufficient resources to run the whole program, it is still possible to normalize the performance based upon the two-processor performance. The latency is varied from $L = 0$ to $L = 100$ (i.e., 0 to 100 delay elements per network port) through this set of runs. It is evident from the curves that the TTDA can indeed sustain an extraordinary amount of latency while still retaining much of its speed. The significance of this accomplishment must be viewed in light of the relatively small size of the run (200,000 instructions, or roughly one second of execution on *one microcomputer.*) Based on these parameters for the simulator, a system with 16 dataflow processors only incurs a 50% loss in performance with an introduction of 200τ of delay on every network-bound token. A smaller system, which is much more realistic given the size of the run, fares even better. In comparison, a single von Neumann processor with blocking memory references would expect to experience $3/4 + 1/4 \times 200 \approx 50$ times reduction in speed given a similar arrangement[6]. The reason that von Neumann processors fare so poorly in the presence

[6] Assuming that a 25% mix of the von Neumann machine's instructions executed require a memory reference

of memory latency is that such processors must idle during an entire memory reference. On the other hand, the dataflow instruction scheduling mechanism will issue a memory request, then immediately schedule another instruction from a parallel thread of computation belonging to either the same or an entirely different code-block. Thus, the dataflow processor is busy doing useful work between the request and its completion, rather than idling uselessly as a von Neumann processor would. In this way dataflow processors can take full advantage of the pipelining of the processor and the network, whereas von Neumann processors can do so only to a very limited extent.

3.5.2 Correlating the Experimental Results from GITA

As a second confirmation of this property, the same experiments were performed using GITA. The latency specified applies to all tokens, not just the network-bound ones. So the latency setting for the GITA runs are varied from $L = 0$ to $L = 50$. The resulting curves bear the same general trend as those from the simulator, but whose absolute speedup values do not resemble each other at all. The induced latency affects the behavior of GITA much more than it does the simulator, but only because GITA performs so much better under little or no latency. This, as we shall see next, is due to GITA's ability to exploit all the parallelism not expended in masking latency to speed up its own execution.

We are indeed able to get a very close match in the speedup curves with the multiple-queue version of GITA, with each queue modeling a single processor. The code-block invocations are assigned queues according to a round-robin[7]. The speedup curves are normalized by the same constant as that used for the "finite-processor" GITA curves, which defines the two-processor GITA as having a speedup of two. It is obvious that this set of curves is very similar to that of the simulator. The slightly irregular results manifest in the intersection of the curves are due to the variations in the round-robin mapping assignment: When there are relatively few tasks, such as when the run size is small and the code-blocks are large, mapping assignments start to have a greater impact on the completion time, as we shall examine in detail in the next chapter. Results similar to these have increased our confidence in the validity of GITA,

across the network, and that instructions are fetched locally on the processor. And although 200 delay units for global reference is excessive except for the largest of systems, a high-performance computer, the Cray 2, for example already requires 52 clocks for each memory access, and that is only a bus transaction which doesn't even include network delays yet.

[7]The issues of allocation policies will be discussed at length in Chapter 4.

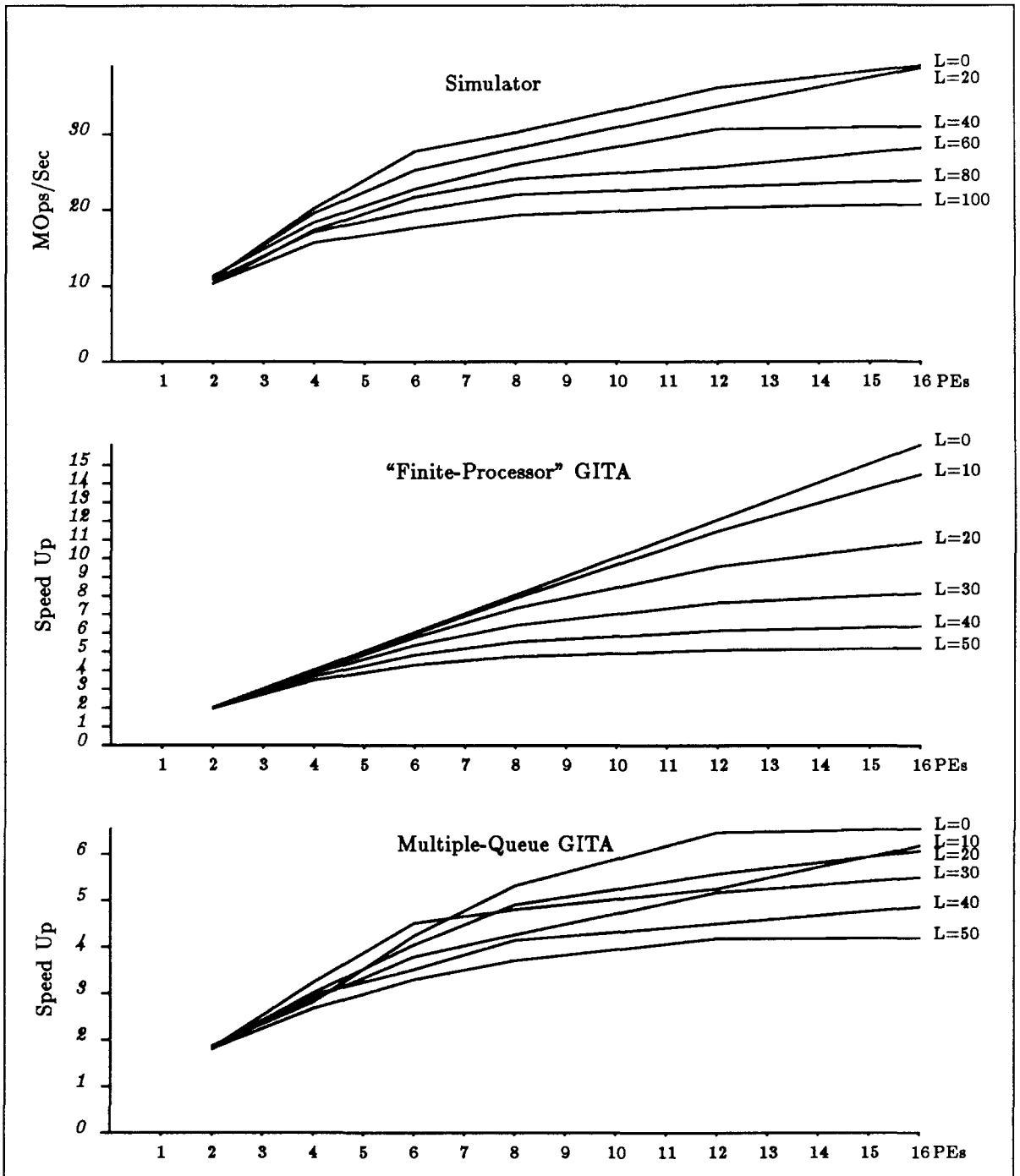

Figure 3-11: Effect of Latency on Speedup of SIMPLE (1 iteration, 10×10).

as not just an abstract graph interpreter, but as a useful tool in predicting the performance impacts of various resource-mapping factors on the TTDA itself.

Chapter 4

The Effects of Code-Mapping Policies on Performance Scalability

The ultimate success of multiprocessing as a means of increasing computation speed beyond that afforded by advances in semiconductor device technology rests on the ability to partition, allocate, distribute, and schedule work efficiently amongst the large collection of available system resources. Optimizing the efficiency of this mapping process can be viewed as minimizing required overhead to parallelize the computation while maximizing system utilization. The problems of partitioning, allocating, distributing, and scheduling tasks optimally are NP-complete, but we shall present a method for examining these code-mapping problems which will show that some very simple strategies indeed work surprisingly well relative to the theoretically optimal strategy.

This chapter examines some policy issues governing the partitioning, allocation, distribution, and scheduling of tasks on the MIT Tagged-Token Dataflow Architecture. Some of the major issues that arise in formulating effective code-mapping policies for a multiprocessor system include the following:

- Manual *vs.* automatic: The partitioning, allocation, distribution, and scheduling can be specified either manually with explicit program annotation or automatically by program-development tools or an ubiquitous run-time supervisor.
- Static *vs.* dynamic allocation: Many other mapping-policy choices are affected by whether tasks are assigned to processors by some compiler phase or during run-time.

- Fine *vs.* coarse granularity: Dataflow graphs allow very fine-grained task-mapping, but in order to establish the proper trade-off between communications overhead and exploitation of parallelism, the granularity of the tasks must be adjusted by grouping many dataflow operators into a single task because there is no more benefit in favoring a finer granularity, once there are enough tasks to keep all the processors utilized.

- Intelligent *vs.* oblivious policy: The allocation policy is considered *intelligent* if its decisions are dependent functions of processor load or network topology or some other run-time environmental factors, so that it appears to be actively attempting to balance and regulate processor activity or to minimize network delay and traffic. An oblivious policy does not try to sense or regulate the run-time environment.

- Balance *vs.* locality: Although the ideal policy is one that achieves both balanced processor load *and* maximized locality, in reality these are conflicting goals and striking the proper compromise seems to be an elusive art.

- Data *vs.* program locality: In the attempt to exploit computational locality to minimize network delay and traffic, the allocation decisions based on program locality are often incompatible with those based on data locality.

Our predilection is for an automatic code-mapping process which needs very little or no programmer instruction or intervention. Our contention is that manual program decomposition for MIMD computers will discourage most programmers from ever using more than a single processor at a time, as usage statistics from most Cray X-MP and other multiprocessor supercomputer installations have borne out.

Faced with an array of complex and interdependent issues, we shall start this chapter by clarifying the code-mapping process of the TTDA with a simple model. It identifies some mutually independent phases of the process, which allow us to begin unraveling the factors shaping the final observed behavior of the dataflow machine. A further significance of this process model is that for each phase, a theoretical, optimal performance scenario can be determined for a given program, thus allowing the effectiveness of many allocation policies and partitioning granularities to be evaluated in absolute terms. Lastly, we examine the impact of communications latency on code-mapping strategies and on system performance.

4.1 The Code-Mapping Process of the TTDA

The code-mapping process of the TTDA can be conceptualized as a controlled progression of tasks through a series of states. Figure 4-1 describes this code-mapping process logically; the solid lines indicate paths of task status transitions while the dashed lines indicate paths of choices controlling the transitions. At the start of the computation, all tasks are in the dormant *wait state.* An initial task is enabled and enters the *ready state.* The processor-allocation policy chooses an execution host for each enabled task and moves it into the *run state* associated with the target host. From the pool of runnable tasks, the priority scheduler selects a single favored task for execution. At the completion of the execution, some succeeding tasks are then enabled. This process continues until all the processors are idle and there are no more enabled tasks. There are, then, three individual phases to the entire code-mapping process: task-partitioning, -allocation, and -scheduling. In the next few sections, we shall first determine the lower-bound performance of any eager scheduling discipline, such as the one implied by the FIFO token queue, because a proof exists for establishing a lower bound and because its effect pervades throughout the empirical investigation of both the partitioning and the allocation policies. Then, we shall discuss the goals and issues involved in the task-partitioning and -allocation and develop some abstract models to characterize the ideal, upper-bound performance of any such policies. These abstract models also serve to separate the mutual interactions between partitioning and allocation concerns and help to identify the relevant factors affecting each phase of the code-mapping process. Although we cannot devise the optimal policy and demonstrate its performance, the observed behavior of some realistic and readily conceivable policies will be presented in light of the upper bounds given by these abstract models so that their effectiveness may be evaluated.

4.2 Dataflow Task Scheduling

We first establish an abstract scheduling model and use it to prove some bounds on anomalies inherent in the scheduling process. We shall then turn to examine the scheduling problem for the TTDA in terms of this abstract model and demonstrate the effect of the scheduling anomalies on the behavior of the TTDA.

A scheduling model consists of a set of system resources, some task systems, sequencing

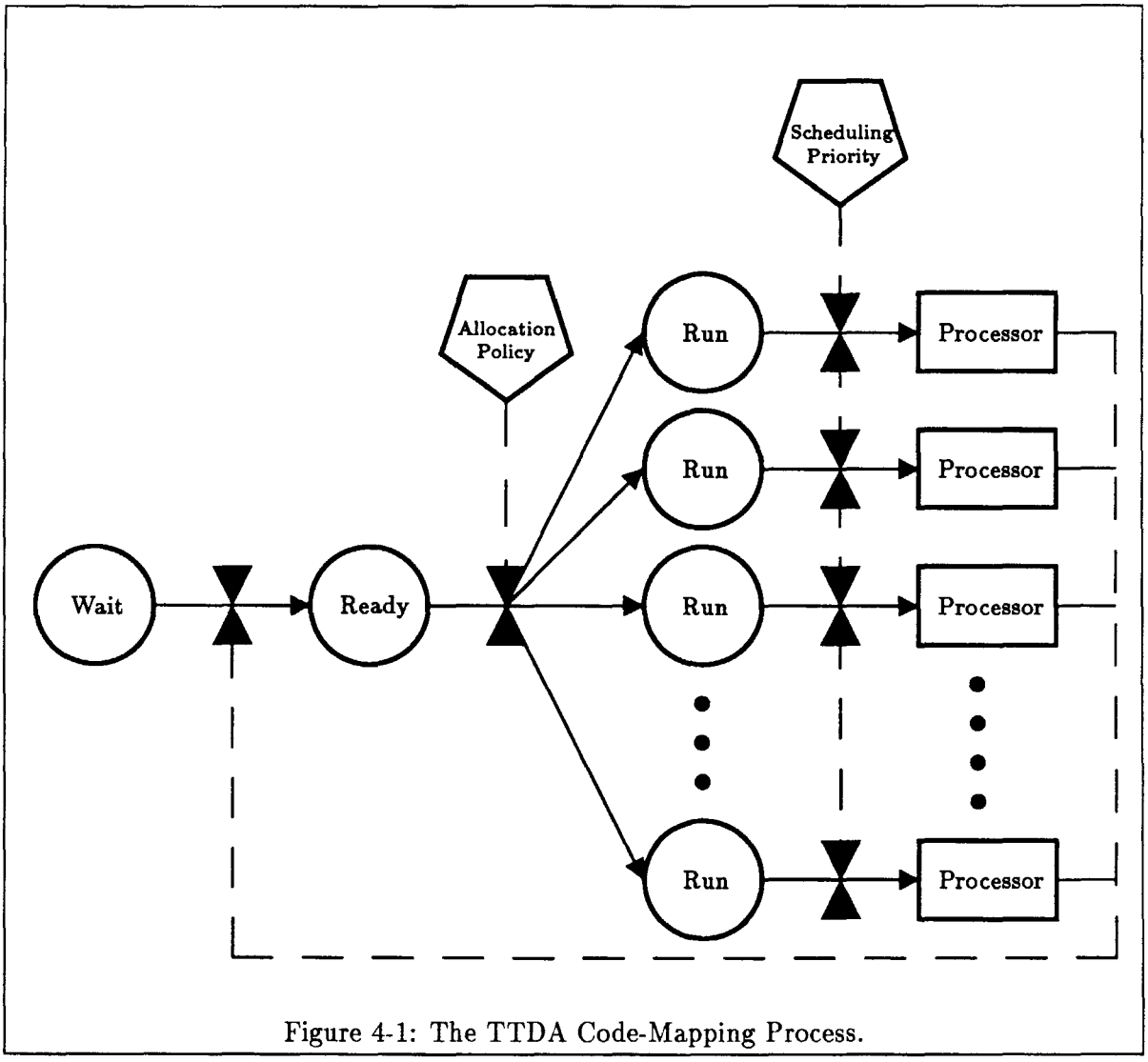

Figure 4-1: The TTDA Code-Mapping Process.

constraints, and a set of criteria the execution must meet. Of the system resources, only homogeneous processors, $P = \{P_1, P_2, \ldots, P_n\}$, are actually considered for our scheduling purposes. Each task system is a three-tuple, $(\mathcal{T}, \prec, \mu)$, where

- $\mathcal{T} = \{T_1, T_2, \ldots, T_m\}$ is the set of tasks constituting the program to be executed.
- $\prec$ is a partial-order relation defined on $\mathcal{T}$ such that $T_i \prec T_j$ denotes the precedence constraint that T_j cannot start executing until T_i has terminated first.
- μ is an m-vector of task execution times. $\mu_i > 0$, $1 \leq i \leq m$, is the time a processor takes to execute task T_i.

For now, assume the precedence constraint relation $\prec$ is acyclic. The task system can then be represented by a directed acyclic graph, where the tasks form the nodes and the precedence relation forms the directed arcs. We shall also assume the scheduling discipline is non-preemptive; a task, once scheduled, cannot be suspended to be resumed later. The scheduler always assigns a ready task, one that has satisfied the precedence constraint relation, to an idle processor at the earliest moment, *i.e.*, right when the last of its predecessors have completed. This is known as *eager scheduling*. When there are more ready tasks than idle processors, the scheduler consults a (possibly dynamic) priority list to decide which of the tasks are scheduled first. The rest are left pending until some processors become available, during which time the priority list is used to serialize the pool of ready tasks again and the favored tasks are then scheduled. When a processor becomes idle and there is no ready task for it to execute, it is said to be executing an empty task ϕ_i until the next time the scheduler is invoked by the system. After which, if it is still idle, it will execute another idle task ϕ_j, $i \neq j$.

Consider the case when two different priority lists, L and L', are used to serialize ready tasks for an n-processor system. We wish to determine how much variation there is between their completion times, $t(n)$ and $t'(n)$. For a valid schedule $\mathcal{S}$, first let ϕ_{i_1} be the last empty task to finish. There may be several, but they will all have the same starting and finishing time. Let T_{i_1} be the corresponding non-empty task which finishes at that same time. There must be one by the definition of the scheduler; if there are several, then arbitrarily choose one. Then let ϕ_{i_2} be the last empty task to complete before the starting of T_{i_1}, and proceed to define T_{i_2}. Inductively repeat this process until there are no earlier empty tasks left. Let T_{i_r} be the last task picked thus. $T_{i_k} \prec T_{i_{k-1}}$ for $1 < k \leq r$ since there is an idle processor doing ϕ_{i_k}

that would otherwise be able to do $T_{i_{k-1}}$. So there is a chain of tasks $T_{i_r} \prec T_{i_{r-1}} \prec \ldots \prec T_{i_1}$ such that whenever a processor is idle, some other processor is executing one of these tasks. Thus

$$\sum_{\phi_i \in \mathcal{S}} \mu(\phi_i) \leq (n-1) \sum_{k=1}^{r} \mu(T_{i_k}).$$

So

$$t(n) = \frac{1}{n}\left(\sum_{T_i \in \mathcal{T}} \mu(T_i) + \sum_{\phi_i \in \mathcal{S}} \mu(\phi_i)\right) \leq \frac{1}{n}\left(\sum_{T_i \in \mathcal{T}} \mu(T_i) + (n-1) \sum_{k=1}^{r} \mu(T_{i_k})\right).$$

Since

$$t'(n) \geq \frac{1}{n} \sum_{T_i \in \mathcal{T}} \mu(T_i) \quad \text{and} \quad t'(n) \geq \sum_{k=1}^{r} \mu(T_{i_k}),$$

substituting for $t'(n)$,

$$t(n) \leq \frac{1}{n}(nt'(n) + (n-1)t'(n))$$

and rearranging terms,

$$\frac{t(n)}{t'(n)} \leq 1 + \frac{n-1}{n} \approx 2, \quad \text{for large } n.$$

This result indicates that the performance of *any* eager-scheduling choice would be less than a factor of two worse than the optimum schedule. A more general proof for bounds on scheduling anomalies when $\prec$, μ, and n are also varied is given in [19].

To see how the scheduling choice affects the parallelism of a program, we show two different parallelism profiles, based on two scheduling policies: FIFO and LIFO. FIFO scheduling is fair, as defined in Section 3.4.2. With LIFO scheduling, the most recently enabled instructions are considered before any enabled activities carried over from earlier steps. Figure 4-2 shows the results of applying the scheduling orderings to the inner product program graph introduced in Section 2.1.2, with only two processors, executing in the style of the "finite-processor" mode described in Section 3.4.2. In general, when scheduling a large number of small tasks, as is the case for dataflow instructions, the variance due to scheduling anomalies is not significant. "Finite-processor" mode completion times using FIFO and LIFO scheduling for some kernels are shown in the following table:

Kernel	Policy	Number of "Processors"		
		100	500	1000
SIMPLE 32	FIFO	—	7014	4114
	LIFO	—	7503	4352
MatrixMul 20	FIFO	2805	709	—
	LIFO	2859	795	—

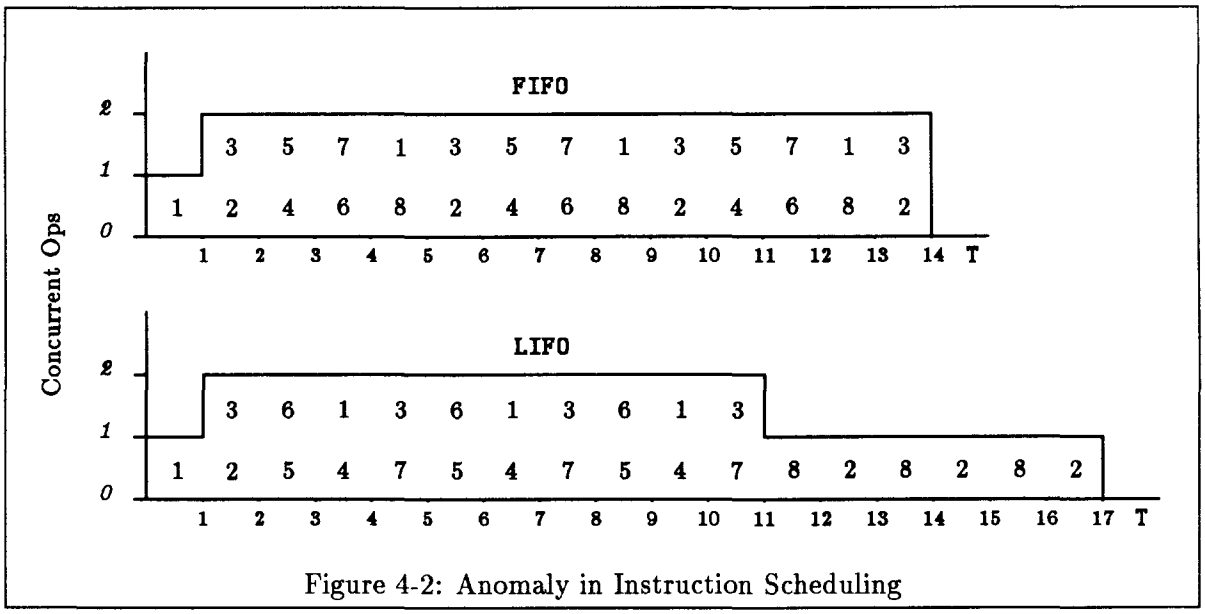

Figure 4-2: Anomaly in Instruction Scheduling

4.3 Task-Allocation Policies

Since every control in the code-mapping process affects the ultimate performance of the system and also affects the optimal choice of other controls in the processing loop, picking a proper starting point for the study could only be arbitrary. The first control point we shall examine is the effect of task-allocation policies on execution performance. This cursory study is done with the other controls and parameters set to what are deemed reasonable, but still arbitrary, choices. To remedy this fallacy, in a later section the task-allocation issue will be revisited, after other issues have been considered more fully.

The program graphs for the first dataflow machines proposed were statically allocated onto the processors. This approach necessarily forbids recursive function calls in the source language. Other than this limitation, it was thought that complex and intelligent allocation

strategies should be implemented in the post-compiler phase [17, 12]. The entire program graph, with all function calls expanded, must be traversed and nodes annotated with firing times, based on a timing chart of instruction execution and assumed communications latencies. The code graph can be partitioned to maximize concurrency by attempting to assign nodes with similar firing time to distinct processors, and to minimize interprocessor communications cost by attempting to assign preceding and succeeding nodes to the same processor.

This type of dataflow graph analysis is intractable for several reasons. Unlike the usual task scheduling problems which work with procedures or even entire processes, the number of fine-grain dataflow operators that has to be scheduled for each realistic program overwhelms most scheduling algorithms. Also, unlike coarse-grain task-scheduling problems which are often plagued by inadequate concurrency, dataflow program graphs give too much concurrency. Although parallel threads of computation can be executed independently, there is no reason that they should be allocated to different processors unless they will actually be scheduled simultaneously. But determining the simultaneity of events requires, besides just knowing the topology of the graph, much run-time information as well. The status of the processing pipeline, (*e.g.*, the fill level of the buffers, and the number and variety of the other instructions in the pipeline,) contributes to variations in the timing of each instruction. The network's performance is similarly critically dependent on dynamic traffic load. Under heavy traffic, the communications latency increases as more packets are in transit state within the network and even the network throughput may decrease as congested nodes could block upstream nodes from delivering packets to elsewhere [26]. Furthermore, the asynchronous instructions such as memory fetches and manager requests can take an arbitrary amount of time. The execution semantics only guarantees that, for any logically-correct program, these operations will eventually complete. Indeed, it is impossible to tell *a priori* how long an I-structure fetch will take in general, as that depends on when the corresponding I-structure store completes, which cannot be inferred from the text of the program. The I-structure memory provides dynamic arcs between stores and fetches and the exact topology of these arcs is dictated by the array indices computed at run-time.

Thus, in searching and exploring candidate task-allocation policies, we eschew automatic static partitioning on these grounds. The viable alternatives, then, are manual-static and automatic-dynamic partitioning. Manual-dynamic partitioning is obviously infeasible since a programmer would be overwhelmed by the volume of run-time allocation decisions.

To survey the range of possibilities, the following task-allocation policies have been chosen and implemented on the emulation facilities:

Round-robin At run-time, successive procedure invocations are dispatched to processors based on a global round-robin processor pointer. This policy is fair, assuming procedure code-blocks are of approximately the same size, in the total amount of work allocated to each processor and also in the temporal distribution of work among the processors.

Random At run-time, procedure invocations are assigned to processors chosen by a pseudo-random number stream. This policy is fair only *statistically* but does not require global synchronization.

Hashing Each procedure is allocated on every processor in the system. During run-time, a hashing function on the tag field of the tokens destined for an instruction determines the actual processor that will receive the tokens and thus execute the given instruction. This is the classic scenario for dataflow machine instruction mapping.

Load-leveling At each procedure invocation time, the *load factor* of each processor is queried and the procedure is assigned to the processor with the lowest load factor. The load factor of a processor is determined by the amount of active code (*i.e.*, the sum of the sizes of all the active code-blocks on that processor, where a code-block is active if an invocation of it is outstanding.)

Static The programmer specifies a semi-static assignment of procedures to processors. Currently there are four hierarchical procedure-mapping categories and three processor groups for the SIMPLE kernel:

1. Outer-level procedures and loops are allocated on group one processors on the round-robin basis.
2. Procedures and loops called by outer-level loops are distributed by the outer loops' context indices. Thus, all activities spawned off by the first loop iteration are dispatched to the first processor in group two, and so on.
3. Procedures and loops called by inner loops are distributed by the inner loops' context indices to group three processors.

4. Short-procedure calls are effectively "inlined" by invoking them on the same processor hosting the caller.

Obviously, this policy maps a program well only onto a certain number of processors, depending on the problem size.

The following table shows system performance and variations in processor utilization (one iteration of SIMPLE 10 x 10):

Policy	*Execution Rate (MOps/sec)*	*PE Utilization Rates (%)*
Round-Robin	38.197	56, 49, 59, 41, 54, 57, 56, 35, 49, 64, 40, 38
Random	37.772	56, 43, 47, 44, 54, 47, 53, 55, 41, 69, 38, 44
Hashing	19.147	30, 26, 26, 26, 26, 26, 26, 27, 26, 26, 26, 26
Load-Leveling	44.940	61, 58, 55, 55, 54, 63, 60, 60, 59, 56, 62, 61
Static	32.885	45, 57, 41, 39, 39, 40, 42, 37, 48, 46, 40, 40

These results seem to indicate that, in the absence of complete information about the topology of the runtime program graph and instruction timing, the simple, oblivious dynamic allocation policies actually perform up to par against either a more intelligent static allocation policy, which imposes a specific processor assignment based on knowledge of the structure and behavior of the program, or a load-leveling allocation policy, which senses the "load-level" of the processors in the system in determining the processor assignment.

A real implementation of the load-leveling policy requires the resource manager either to probe the status of processors in the system to determine their load factor or to maintain knowledge of all outstanding task invocations. Either of these methods would force the system to do some global operations, which will be inherently non-scalable[1]. Furthermore, sensing load factor can produce undesirable results: An allocated task can actually be inactive, waiting for

[1]The SIGMA-1 has addressed this scalability problem by implementing a scheme in which distributed load-leveling is performed through the communications network [21].

data from another task. It is nevertheless counted the same as an active task by our scheme. Or, if a more direct method to examine the processor token queue is implemented, an underutilized processor may be swamped with tasks before its load status starts to reflect that.

The intelligent static policy, which divides the processor pool into three disjoint groups to support the logical mapping described, showed low utilization for all the processors. This results from the difficulty in balancing the workload evenly among the processor groups at compile-time without knowing the time domain behavior (*i.e.*, the footprint) of the computation, which often concentrates in one group, thus leaving other groups idle, and then moves on to another group leaving the previously busy processor group idle. Figure 4-3 shows the activities of each group over the time domain. The run is a 16 by 16 SIMPLE using GITA, with 16 PEs in each processor group. Note how well groups 2 and 3 could be effectively merged with little impact on execution time, and how underutilized group 1 is. In fact, when the two groups are merged, so that PEs 0 to 15 host group 1 and PEs 16 to 31 host groups 2 and 3, the execution time only degraded from 59268 steps to 71802 steps, a 20% increase. When all three groups are merged onto one set of 16 processors, the execution time becomes 89016, a 50% increase over that for the original 45 processor system, with only 36% the computing resources.

System Size	*Execution Time*	*% Increase*	*Relative Utilization (%)*
45	59268	—	100
31	71802	20	175
16	89016	50	422

4.4 Task Granularity

Not a part of the runtime code-mapping process, but nonetheless an influential factor in it is the task granularity, the amount of work parceled into each unit of task that the allocation process distributes to each processor. The granularity of a program is chosen for the compiler before it can generate the object code graph. The criteria for choosing an optimum task granularity are to balance the system load (which favors fine-grain) and to minimize the overhead of parallel operation (which favors coarse-grain.) These are highly dependent on the characteristics of the machine as well as on that of the programs themselves.

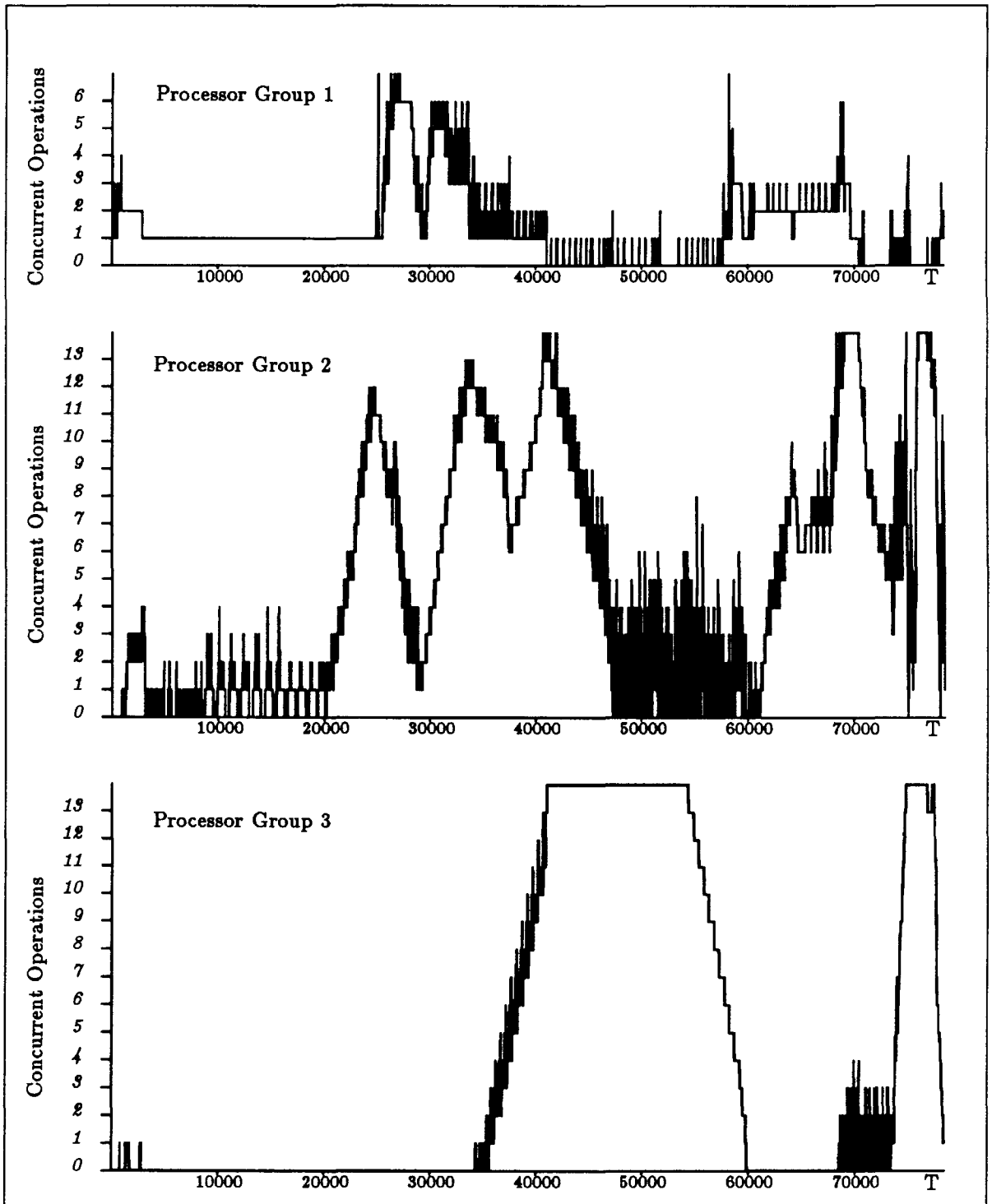

Figure 4-3: Work Distributed to Processor Groups by a Static Allocation Policy.

Machine-hardware-related factors such as network throughput (bandwidth) and latency, processor throughput and the number of its pipeline stages, and memory operation speed, dictate how much parallelism each processor needs to keep itself busy and where the trade-off point is for a processor executing an instruction itself, versus sending it to another processor for execution, thereby incurring the communications cost through the network. Having more processors in a system would, no doubt, favor a finer partition granularity in general. On the other hand, constants within a large task can be stored in one or a few processors as context information, and thereafter used repeatedly without paying the distribution cost. Other systems-related factors, such as the number of task-allocation managers and the complexity of task-allocation policies and operations, determine the rate at which the system can process task invocations and terminations and how much overhead is associated with each task allocation. Lastly, the program itself presents factors to complicate the case. When more parallelism is available in a program, there is more to gain from a fine-grain partition; whereas, when the program is largely serial, a similar approach becomes less profitable. Furthermore, the amount of I-structure memory references and the *inherent locality* of the logic of the program impose a minimum (best-case) traffic load on the network. When this requirement is light, *i.e.*, the program activities are very local, the mapping policy can afford to be more generous in trading communications bandwidth for finer granularity.

Note that, on parallel machines consisting of a collection of von Neumann processors, once a task has been allocated to a processor, it is eventually scheduled to gain processor attention and complete in one, or a few, continuous time spans. Thus, it is convenient to refer to *the* task granularity. Unlike these von Neumann parallel computers, for the TTDA we need to distinguish between the distribution, or allocation, granularity and the scheduling granularity. Although a real code-mapping decision would inevitably group many dataflow instructions into a single task in order to increase the code distribution granularity to reduce the immense overhead that would otherwise be incurred in a fine-grained task allocation system (as the hashing policy results had indeed verified,) the underlying fine-grained dataflow *instruction scheduling* is still invaluable in providing the intra-processor parallelism that keeps the pipeline saturated with activity and enables the system to tolerate large communications latencies. The *task granularity* of interest here refers to the distribution granularity, whose indirect constraint on instruction scheduling is that, at any given time step, at most one instruction from each task can be executing. Coarse-grain tasks thus reduce the parallelism available for multiprocessing

(as opposed to the intra-processor parallelism.)

4.4.1 Choices of Task Granularity

Since the machine- and systems-related factors affecting choice of granularity cannot be determined until a real hardware implementation of the TTDA is available, setting these to some arbitrary values on the emulation facility to obtain runtime data in support of a particular granularity is futile. Instead, we shall attempt to investigate the effects of task granularity along more broad, qualitatively distinct lines of division: at the instruction, iteration, and code-block levels, and evaluate their relative merits. We have already seen that instruction-level partitioning destroys locality and places heavy demand on the network bandwidth, but it will be examined again for the sake of having a reference point at the very end of one spectrum that will make interpolations possible. The code-block invocation, which corresponds to an instance of an Id function or loop, also provides a convenient granularity for partitioning. There are enough such invocations in most complex programs to yield a sufficient number of tasks for distribution among the processors, and the code-blocks also form natural boundaries of localized activities and minimized communication, the arguments and results of the invocation. In between the instruction and the code-block task granularities is iteration task-allocation granularity. There are two motivations for distributing iterations of a loop code-block:

- Consecutive iterations of a loop do not have to compete against each other for processor time, as they are allocated onto distinct processors. This is significant for two reasons. First, the activities of the iterations of a loop are largely concurrent, consecutive iterations lagging only by the loop index calculation and context change, so that distributing iterations is almost always worthwhile. Secondly, the loop index calculation and context change operations within a loop are invariably part of the local, if not global, critical path of the program, so that distributing iterations, and thus increasing the priority of the loop code-block if the system is lightly loaded, often improves the execution time by the same virtue that critical-path scheduling is often better than plain-list scheduling. As a result, the distributed loop unfolds quicker and thus exposes more parallelism, within its body, and generates more concurrent activities.

- Distributing iterations makes the task size roughly proportional to the textual length of the source code-blocks (*i.e.*, functions and loops.) Since the modern programming

practice of top-down, modular design discourages writing long routines, the size of most tasks thus partitioned should be approximately the same, within an order of magnitude. Whereas, should loops be allowed within a task, the task size may become arbitrarily large, depending on the number of iterations specified.

A partitioning policy can also implicitly assign priority to different components of a program. Assume a particular section of code with k concurrently active threads of computation has been partitioned so that it maps optimally onto n processors, each of which is already preoccupied with l threads of computation. The partitioned version of this code thus receives

$$\frac{k+l}{\frac{k}{n}+l}$$

times the processing time relative to the unpartitioned code. When l is small, as would occur when the program is trivial or the number of processors is large for the program, we could expect the partitioned section of code to run up to n times faster. When l is large, however, and each new code-block invocation only marginally increases a processor's load (*i.e.*, $l \gg k$) as would happen in most situations where the problems one wishes to solve are always hopelessly larger than the available computing resources can support, the partitioning would change the behavior little.

4.4.2 Program Parallelism under Different Task Granularities

The series of experimental results that will be used to study the impact of task granularity on the runtime behavior of the TTDA are, again, from runs of the SIMPLE code. But before the results of these varying task granularity choices from a more realistic execution with allocation policy and latency accounted for can be analyzed meaningfully, we should first determine the *intrinsic* parallelism of the program under each chosen task granularity. This can be obtained by allowing an unbounded number of concurrent tasks and isolating tasks from competing for processing resources. Within each task operations are scheduled fairly, as in the "finite-processor" model, based on availability of data; instructions are not statically ordered. Scheduling of operations within different tasks is entirely independent, except as dictated by data dependencies. Any number of tasks can be active concurrently, and in a single step an operation is processed from each task that has any enabled operations. In this way, the effects of intra-task scheduling constraints are isolated so that additional effects due to mapping tasks

onto a fixed set of processors can be completely negated. The parallelism available under this model can therefore be viewed as the upper bound on that available under the chosen partition policy and granularity. This ideal scenario is achieved by assigning each task to a unique queue on the multiple-queue GITA described in Section 3.4.3 while not charging for latency. The data are program intrinsic because they are independent of particular machine configurations or allocation policies, and affected only by the program graph topology and its logical progression dictated by the dataflow execution rules. Figure 4-4 shows the ideal parallelism profiles of one iteration of SIMPLE 32×32 under the three task granularities. Note these profiles all show operations per step, so the areas under the curves are the same. The leading and trailing edges of the peaks generated by the main nested loops of the program are noticeably steeper in the iteration case than in the code-block case, indicating that the successive iterations of the loops are more skewed in the latter case, as we have predicted earlier. This skew reduces the parallelism of the loop; it becomes worse with relatively short loops. While the peak parallelism is reduced to 1/21.5 and 1/74 from instruction to iteration and to code-block grain size, the more telltale critical paths are lengthened to 8.4 and 32 times, respectively, so the average parallelism is 1/8 and 1/32 that under the instruction level model. These ratios represent the loss in *potential parallelism* arising from the scheduling constraints imposed by the combination of the larger task granularity and the single-instruction-per-time-step-per-task scheduling discipline. The same observation is also apparent in Figure 4-5, which shows the estimated speedup, using the speedup function described in Chapter 2, with $l = 0$ from the ideal parallelism profiles of SIMPLE 32×32, 50×50, and 64×64. Note that for size $n \times n$ of this program, the plateaus in the curves of code-block-level partitioning indicate that it lacks more than approximately n-folds parallelism.

Since the size of computations for a real dataflow supercomputer would be *much* larger than what could be attempted here with the emulation tools, it is perhaps more useful to look instead at the asymptotic ratios of the critical paths as the problem size is scaled up, so that we may predict the relative merits in choosing from one of these task granularities for a more realistic size workload run on an appropriate size system. Because there is very limited overlap between successive iterations, it would be easy to extrapolate the behavior of computing more iterations; therefore, variations in the size of the mesh are far more interesting. The following

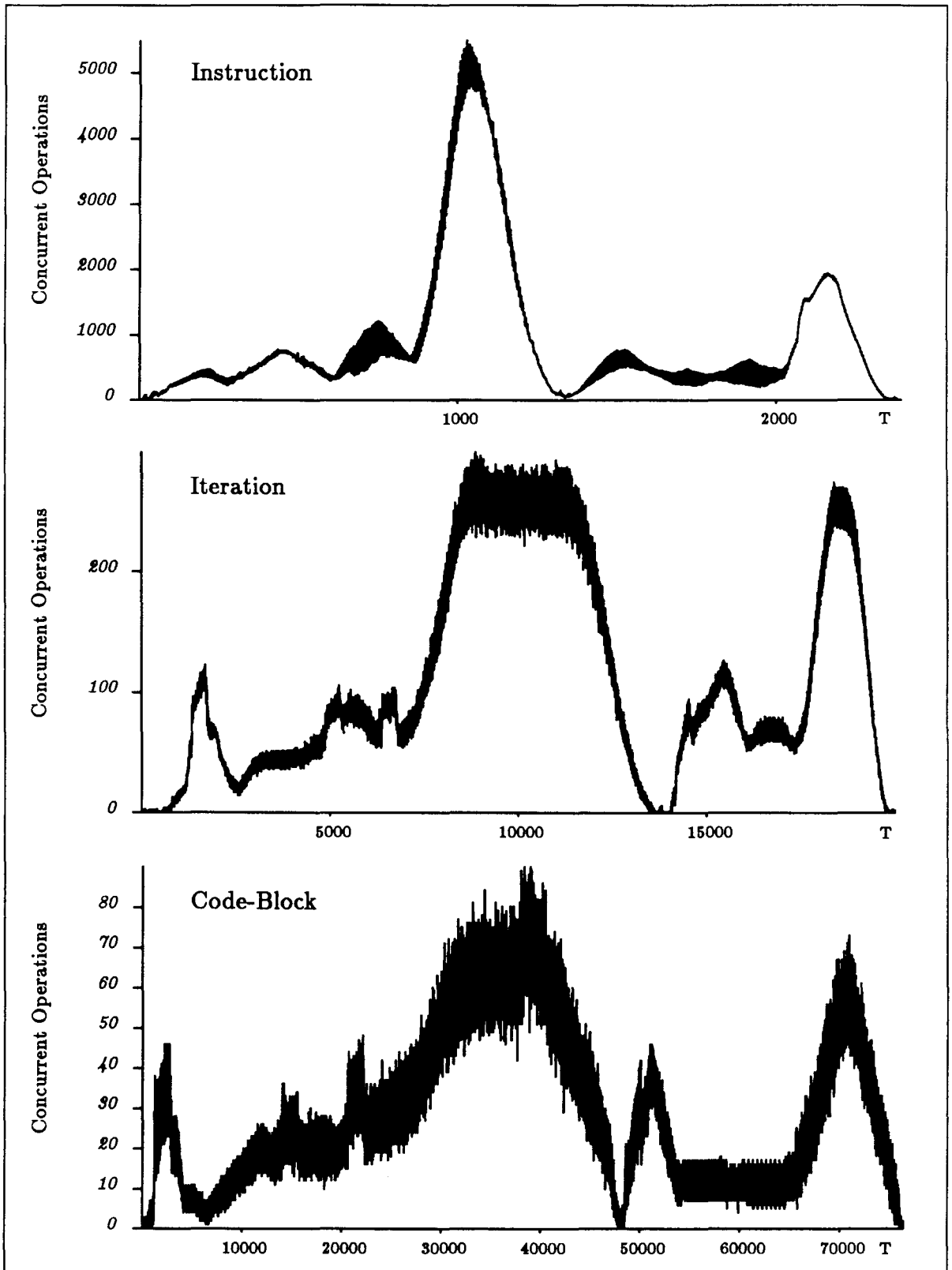

Figure 4-4: Ideal Parallelism Profiles of SIMPLE(32) under Various Task Granularities.

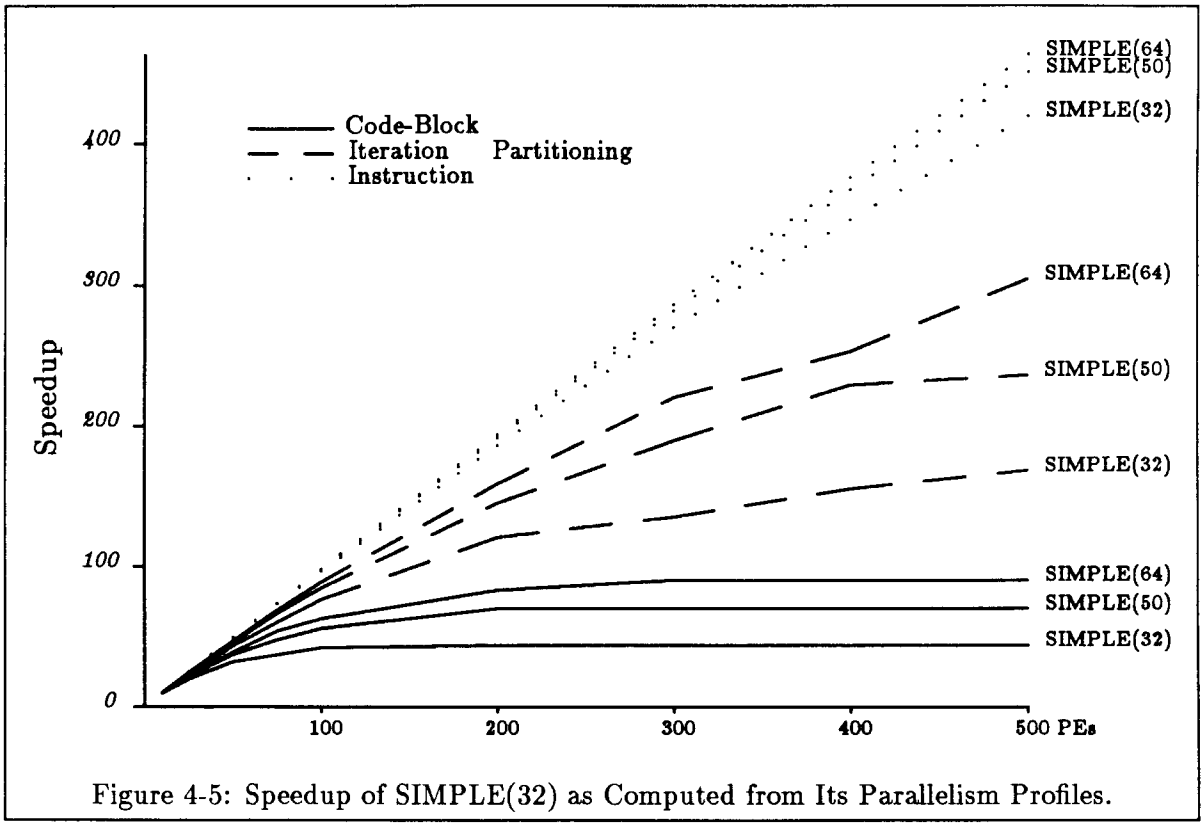

Figure 4-5: Speedup of SIMPLE(32) as Computed from Its Parallelism Profiles.

table shows the average parallelism[2] for one iteration of SIMPLE with various mesh sizes under the three granularities. Figure 4-4 represents the middle column. Across the columns the average parallelism changes with problem size.

Mode	Mesh Size	10 x 10	16 x 16	32 x 32	50 x 50	64 x 64
	Ops	201,806	532,148	2,207,156	5,472,974	9,023,418
Instruction		178	361	922	1584	2108
Iteration		19	40	110	199	270
Code-block		8	14	29	46	59

To observe the trend more clearly, consider the ratio of average parallelism, which effectively normalizes the problem size, given in the following table.

[2]This is identical to the reciprocal of the critical path.

Ratio	Mesh Size	10 x 10	16 x 16	32 x 32	50 x 50	64 x 64
	Ops	201,806	532,148	2,207,156	5,472,974	9,023,418
Instruction / Code-block		22.3	27.8	32.9	35.2	35.7
Instruction / Iteration		9.9	9.0	8.4	8.0	7.8
Iteration / Code-block		2.25	3.1	3.9	4.4	4.6

The ratios indeed seem to be stabilizing for larger size runs, reassuring that the scenario noted would not change much even with an appreciable increase in the size of computation. The phenomenon also can be intuitively justified, since the critical-path ratio of instruction to iteration granularity, for example, just equals to the amount of parallelism within an iteration, which only the instruction-level allocation scheme can exploit. Another noteworthy observation here is that, while the critical-path ratios for the instruction to code-block and iteration to code-block granularity rise with larger runs, that for the instruction to iteration granularity actually falls. The rising ratio curves, which indicate that the code-block-level task allocation performs even worse relative to the other two for increasing size of SIMPLE runs, are due to the two shortcomings of code-block-level task allocation previously enumerated: slow loop unfolding due to contention within the loop code-block, and the task granularity's increasing with an increase in the number of loop iterations. The falling ratio curve is due to the fact that, as increasing size SIMPLE runs are used, most tasks are either inner loops or those spawned by inner loops[3]. Since the inner-loop iterations are often tight and compact, especially in comparison to the top-level initialization routines and the more bulky outer loops, their proliferation in larger runs would be expected to reduce the total effective granularity of the iteration-level allocation scheme.

4.5 The Impact of Task-Allocation Policies on System Speedup

The previous discussion assumed that the execution vehicle has an infinite number of processors, so that every task invoked can be allocated on a separate processor, thus, there would never be any contention for computation resources between activities of different tasks. Under this and the zero communications latency assumptions, then, it is not surprising that the

[3]The number of inner loop instances is proportional to the size of SIMPLE squared, whereas the number of top-level functions and outer loops remain fairly constant.

finer-grain partition would undoubtedly be favored. When limits are imposed on the number of available processors, however, a policy must be selected to map the potentially unbounded set of tasks onto a finite set of processors. An effective allocation policy tries to minimize the possible contention among tasks allocated to the same processor, which would further constrain the useful parallelism originally present in the partitioned program graph.

The simulation study presented in section 4.3 seems to indicate that, in the absence of complete information about the topology of the runtime program graph and instruction timing, the simple, oblivious dynamic allocation policies actually perform up to par against either a more sophisticated static allocation policy, which imposes a specific processor assignment based on knowledge of the structure and behavior of the program, or a load-sensing allocation policy, which detects the "load-level" of the processors in the system in determining the processor assignment. The following discussion, therefore, will focus on the performance of some instances of the simple, oblivious dynamic allocation policies under the bounded-queue GITA described in Section 3.4.3 *sans* communications latency, because they are easy to arrange[4] and incur the least runtime overhead.

A useful metric to establish first is the speedup performance of an "unbiased" allocation policy, one we approximate by the "finite-processor" emulation using the multiple-queue GITA. The n-processor "finite-processor" version assumes there is a processor for each task as before, but it only allows n of them to execute during a time step. When there are more than n enabled processors, the scheduling order is determined by a round-robin. This model differs from one with a real allocation policy in that, when there are $n + 1$ instead of n enabled tasks, every task is slowed by $1/n$, whereas a real allocation policy would force two tasks onto the same processor, thus causing a load imbalance. Note that the only reason this does not produce the shortest-critical-path execution is because the scheduling order, and *not* the allocation policy, is nonoptimal.

The speedup curves of SIMPLE 32 x 32 under various combinations of grain sizes and allocation policies are plotted in Figure 4-6. The allocation policies, real and idealized, sampled here are as follows:

Estimated Speedup computed from the ideal parallelism profile using the estimation function given by Equation 3.1 with $l = 0$.

[4]No programmer input or complex compile-time analysis is needed.

Finite-processor Idealized allocation policy. This is expected to show better speedup over the *estimated* mode due to a pessimistic assumption the speedup function makes about instruction scheduling[5].

Round-robin Successive procedure invocations are dispatched to processors based on a global round-robin processor pointer. Global synchronization is implicit.

Own Round-robin Successive procedure invocations are dispatched to processors based on the local round-robin processor pointer residing on each processor. No global synchronization is required.

Random Procedure invocations are assigned to processors chosen by a pseudo-random number stream. No global synchronization is required.

Evidently the performance of the system, under the current set of idealized conditions, is much more strongly influenced by its task granularity than by its allocation policy, realistic or otherwise. The results also show that these oblivious allocation policies really serve adequately in view of their simplicity, which is very important because the time it takes to decide which processor to use often directly contributes to the overall execution time. In the 75 to 400 processor range, the code-block-level partitioning has long exhausted its usable parallelism to saturate the system with work, as its two flattened ideal-mode curves hint. Any speedup indicated by the curves from the three real allocation policies is directly attributable to the fact that the probability of two or more active code-blocks being allocated on the same processor decreases as the system size increases. We can expect a more intelligent allocation policy to improve this scenario somewhat, but clearly there is much more to gain by reducing task granularity instead. The instruction-level partitioning has more than enough parallelism to keep every processor populated with tasks, so that the impact of incrementally allocating a task to any processor has negligible effect on the system's load balance, as long as the policy remains statistically unbiased towards all the processors. In this situation, even a more sophisticated allocation policy cannot be expected to improve performance. Lastly, the iteration-level partitioning has characteristics in between the two extremes. The two round-robin policies are consistently 80% effective relative to the "finite-processor" mode throughout the sampled range, with even higher ratio (around 90%) approaching the low-end (25-processor

[5]Implicit in the equation's taking the floor function of the fraction.

system.) The random allocation policy behaves noticeably more poorly than either of the round-robins, conceivably because it causes more "collisions" in assigning successive tasks than the more disciplined round-robin scheme.

Although these speedup curves of different task granularities seem qualitatively different, they are really manifestations of the same phenomenon on different scales. That is, for SIMPLE 32×32, if we execute iteration-size tasks on a smaller system, less than 50 processors for instance, its behavior would be much like that of the instruction-level partitioning observed above, whereas if it is executed on a much larger system, 500 to 1500 processors for instance, then it would experience the same symptoms plaguing the code-block-level tasks above. There is one element clearly not in common among the three task granularities, however: uniformity of task size. Instructions are all uniform (*i.e.*, take one time step to execute) in our timing model, but code-blocks can vary widely in size. Furthermore, an insufficient number of task partitions to keep processors loaded with enough tasks to statistically smooth out the load variations among them exacerbates the effects of the disparity in task size.

4.6 The Impact of Latency on System Speedup

All experimental data presented thus far have unreservedly favored fine-grain tasks. We shall now introduce one facet of all real multiprocessor systems which biases against fine-grain partition: communications latency. Its related issue of communications bandwidth is also extremely important, but it will not be addressed because it depends too heavily on the nature of the program, such as the percentage of instruction mix that accesses I-structure, or, in general, the ratio of computation to communications bandwidths, which would involve specifying network designs and their particular idiosyncrasies. Instead, we shall assume that the network has been designed such that it can support the communications bandwidth requirement of the system, or, rather, we can imagine that the processors' speed is scaled down such that the network will not become the bottleneck, so that all the results concluded here may still apply. The other major concern left out is the work required of the managers to allocate and distribute tasks throughout the system. This overhead, as was pointed out earlier, can contribute significantly to the execution time of a program, especially if the managers, by necessity of some global aspects to the policy algorithm, must be centralized. The issue of implementing managers, nonetheless, is a whole different topic onto itself and cannot be adequately dealt with here.

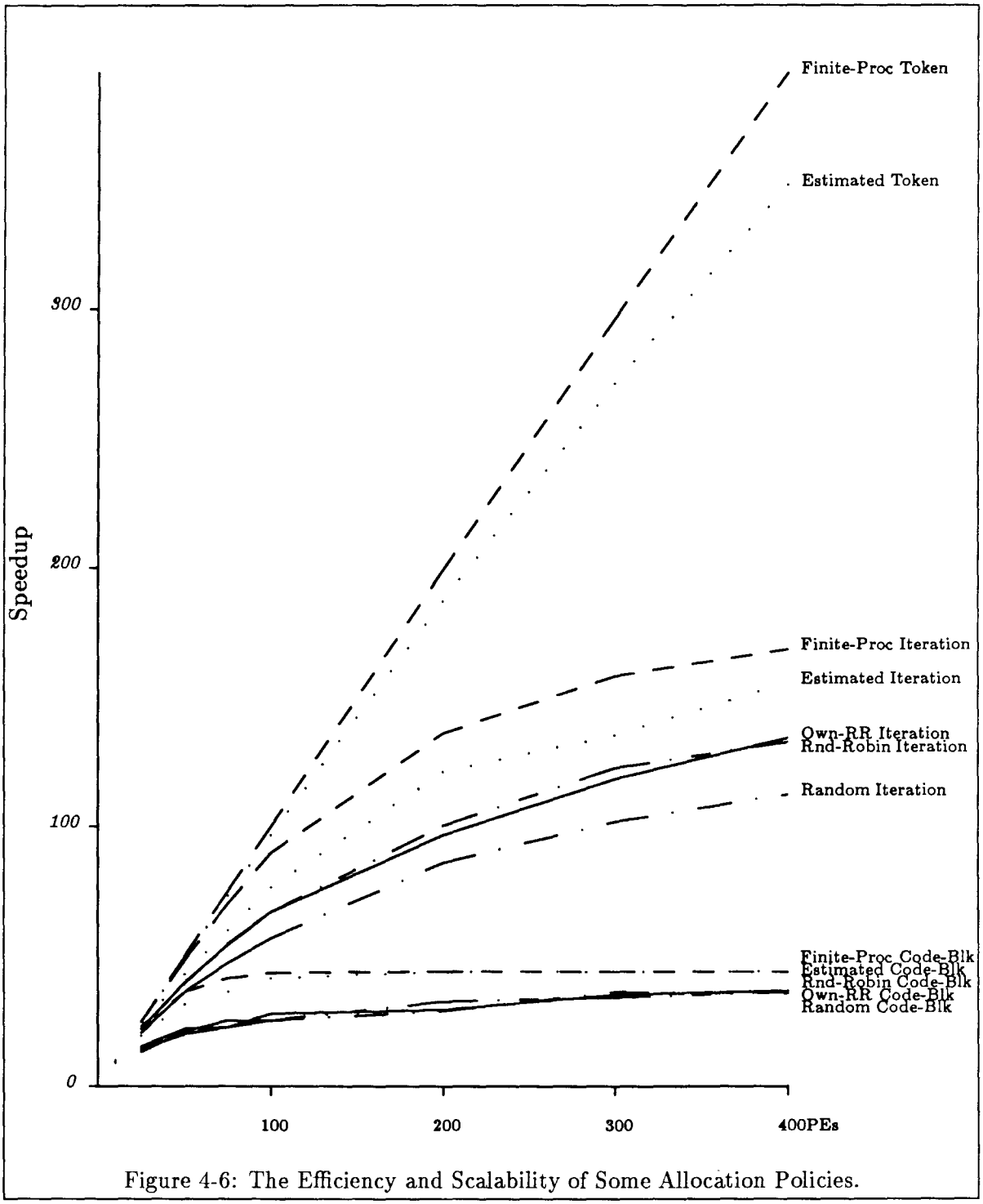

Figure 4-6: The Efficiency and Scalability of Some Allocation Policies.

With the introduction of communications latency we come to realize the utility of both program locality and intra-processor parallelism, which is automatically exploited by the instruction scheduling mechanism of a dataflow computer, but not by a von Neumann computer. To begin, according to the speedup estimation function of Equation 3.1, any latency would be completely masked by execution whenever there is enough work for each processor to do until the network message completes. Otherwise, the extra latency would increase the total running time by forcing processors to idle until the network delivers the packets and more work for the next time step is generated. For a properly mapped program, a large amount of the processing should be internal to a processor, so that only the network-bound tokens would incur this latency. The *interprocessor latency*, IL, is the amount of time charged to each token whose destination is a different processor and to each I-structure request and acknowledgment token. $\bar{l} = 1 \times (1 - X) + (1 + IL) \times X$, where X is the fraction of the network-bound tokens, then gives the *average* latency incurred by each token in the program. This is the value of latency which should be substituted into the above speedup equation.

The fraction of network-bound tokens is certainly a function both of the program itself and of its partitioning strategy. The decomposition of token types for two kernel programs and three partitioning granularities are summarized below:

Sample		*Token type*			
Kernel	*Granularity*	*Local*	*Inter-PE*	*I-structure Access*	*% Non-Local*
SIMPLE 32	Instruction	0	2,760,796	1,393,631	100.0
	Iteration	2,662,938	97,858	1,393,631	35.9
	Code-Block	2,760,796	0	1,393,631	33.5
MatrixMul 20	Instruction	0	237,079	79,225	100.0
	Iteration	182,714	54,366	79,224	42.2
	Code-Block	237,079	0	79,225	25.0

Note that these are the locality figures from the partitioning granularity. During actual execution, the processor allocation policy may assign two tasks to the same processor, thus causing their communication to become local and improving upon these values. This effect is minimal, however, when there is a large number of processors; with n processors, only $1/n$ of such global traffic would be internalized when the allocation policy is oblivious to minimizing network traffic (*e.g.*, the round-robin case). There is no inter-PE communication in code-block

partitioning because all inter-code-block (procedure linkage) information exchange currently takes place through I-structure accesses to allow dynamic procedure-linking.

The empirical results are based on the "finite-processor" mode of the multiple-queue GITA, which, introduced in Section 4.5, reflects an optimal allocation policy. It is further modified by the following latency charges:

- Zero communications delay for tokens between operations within a task, as was assumed for the ideal GITA model.
- Fixed communications delay for tokens crossing between tasks or to or from I-structures.

Figure 4-7 shows the effect of communications latency on SIMPLE 32×32 under various task granularities. IL varies from 0 to 80, while the task sizes used are instruction, iteration, and code-block. The instruction-level version experiences this latency on every token; the code-block-level version experiences it only in transfer of arguments and results, and in I-structure requests. The iteration-level version experiences additional latency on values circulated across iterations. Therefore, we expect that for sufficiently high latency, the reduction in speedup due to latency may be greater than that due to intra-task scheduling constraints. Indeed, at $l = 10$ instruction- and iteration-level versions cross at roughly 50 processors. At $l = 40$ performance of the instruction-level tasks falls below that of the code-block level. However, we should note that the crossovers observed here occur at high latency relative to the number of processors, at a large number of processors given the problem size, and without accounting for the internal latency due to processor pipelining. Figure 4-8 shows a similar trend for matrix multiply. The smaller average code-block size of the matrix multiply leads to much better speedup under code-block partitioning than that for SIMPLE, although it also has become more sensitive to latency increases.

Caveats

It should be duly noted that there are a few designs in the organization of the system which predisposes the outcome of the experimental results towards distribution rather than locality:

- The butterfly multistage interconnection network keeps all other nodes in the system equidistant from a particular node, so that there is no "neighbor" locality to exploit;

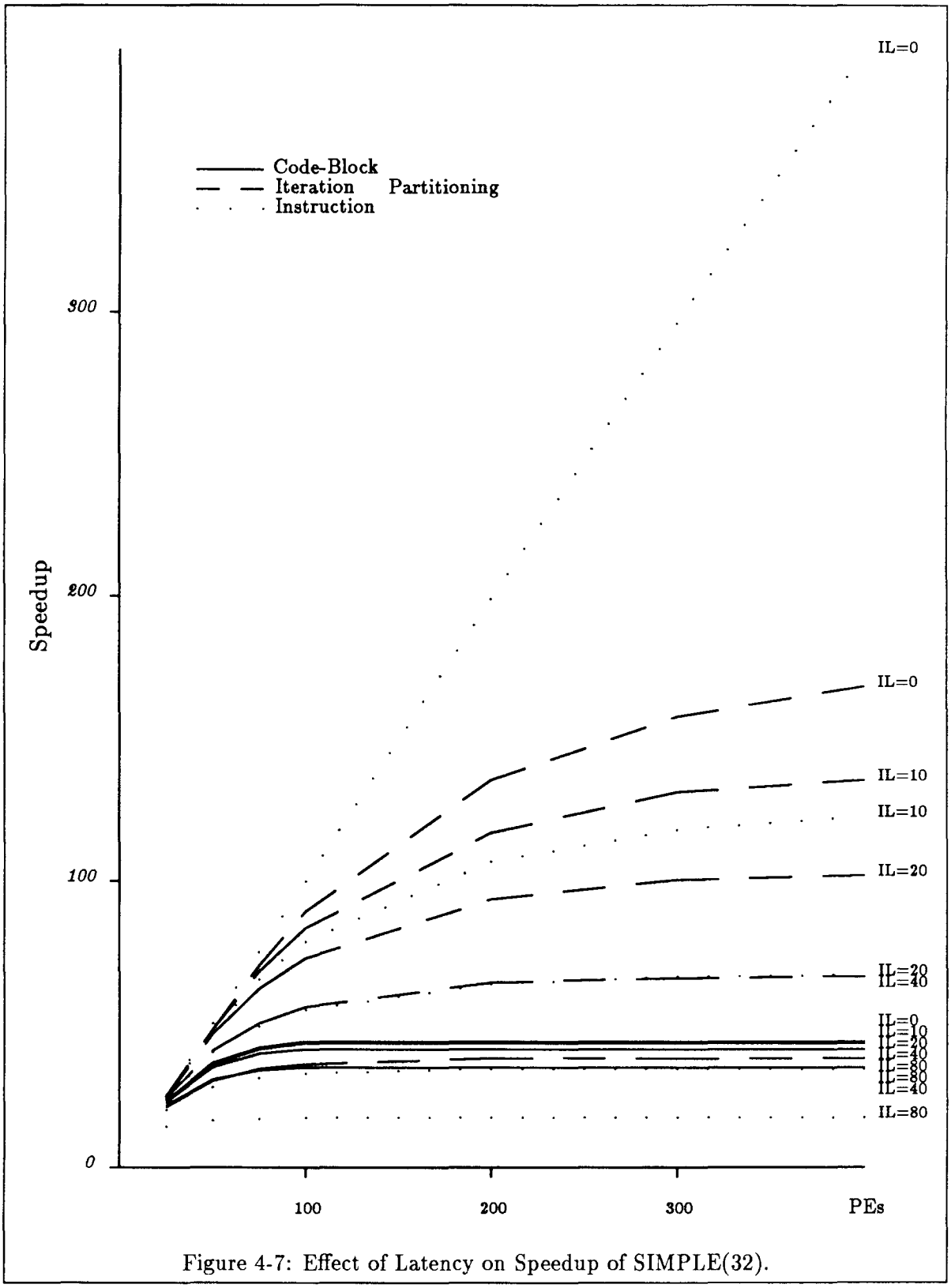

Figure 4-7: Effect of Latency on Speedup of SIMPLE(32).

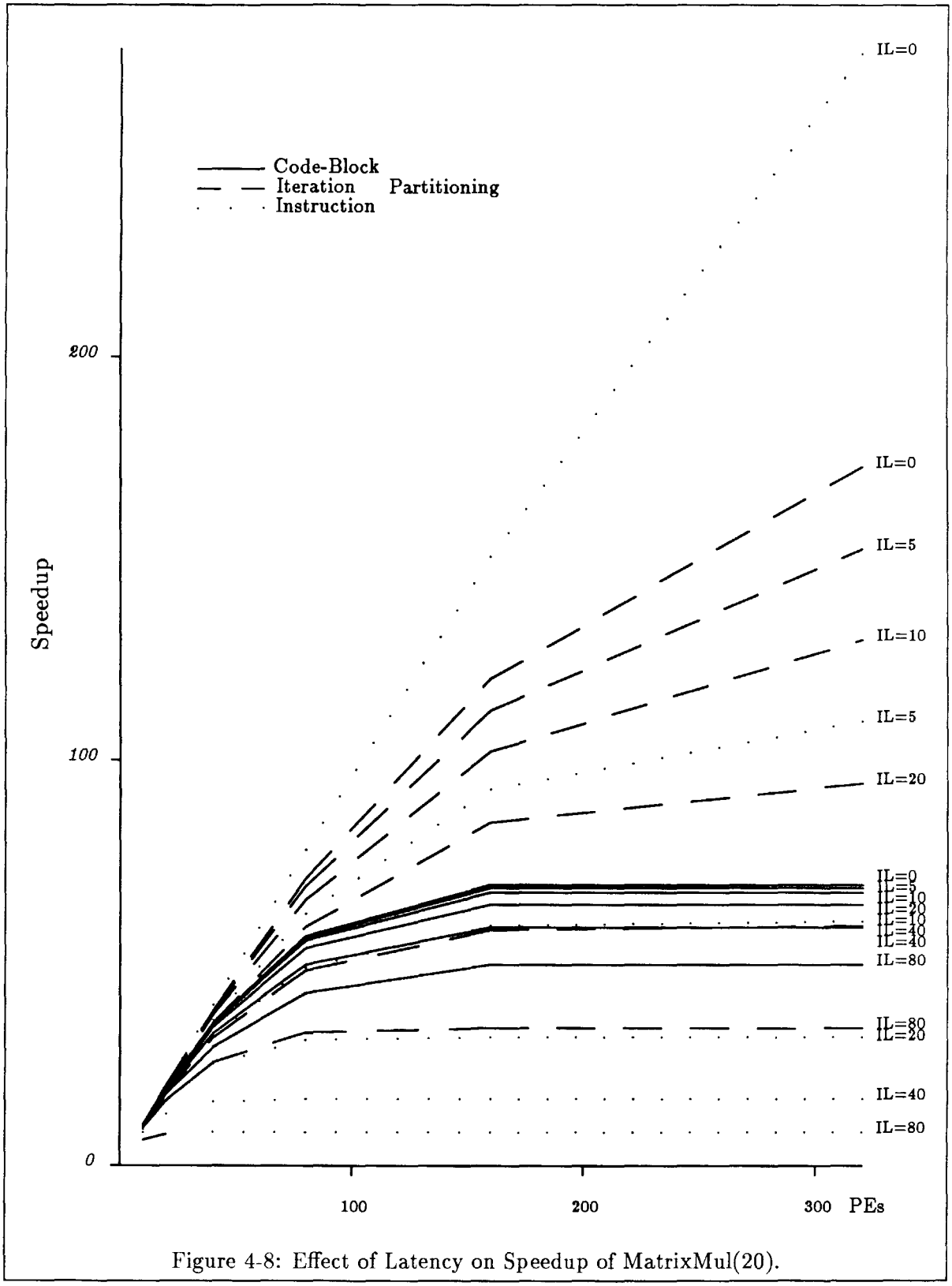

Figure 4-8: Effect of Latency on Speedup of MatrixMul(20).

every access or token is either local or global.

- Data structures stored in I-structure are interleaved throughout every memory module in the system. This strategy favors distribution of work, but reduces the gain achievable in mapping function calls to the same processor executing the caller because the data references that are often shared between function calls become global.

- I-structures are used to pass arguments and results between function calls. This reduces the gain achievable in mapping function calls to the same processor executing the caller, because the I-structure references needed to access arguments and results are always global[6].

Compared to a von Neumann program, the dataflow program already includes *all* of the synchronization costs down to individual machine instructions, hence, there are no more instructions to execute when using finer granularity (and no instruction count savings, either, when using coarser granularity.) Finally, the innate tolerance of dataflow processor to communications latencies also greatly contributes to this system's remarkable indifference towards network structure and locality issues, thereby allowing us to concentrate more effort on balancing processor load and maximizing processor utilization.

[6]The most recent versions of the Id-Nouveau compiler can generate direct argument/result-passing code.

Chapter 5

Conclusion

Supercomputers that push to the limits of hardware technology and incur great costs doing so have not begin to satisfy the demands of computationally intensive problems. Currently multiprocessing is touted as the only means of increasing computation speed significantly beyond that afforded by advances in semiconductor device technology. Since it has become relatively easy to provide an abundance of hardware and the communications medium needed for such pursuit, its ultimate success then rests on our ability to partition, allocate, distribute, and schedule work, with minimum overhead costs, amongst a large collection of processors, while maintaining high utilization of them.

To keep many processors busy with work, it is necessary that the program and therefore the algorithm that it expresses have sufficient concurrent operations that can be mapped to those processors. But given that there is much concurrency in many large programs, the onus lies on the code-mapping process to maintain as much concurrency as is feasible without compromising its efficiency. This goal maximizes the scalability of the system. Our study is based on a dataflow computer, the MIT Tagged-Token Dataflow Architecture, because dataflow program graphs explicitly express parallelism down to the machine instruction level and explicitly account the cost of the parallel operation. These properties are invaluable since they permit the mapping strategies to be changed or the number of processors scaled through a very wide range without having to modify the program, which would have made analysis of the empirical results difficult or impossible.

This study mostly concentrates on the "effectiveness" part of the whole cost-effective anal-

ysis needed to determine the exact code-mapping policy for the TTDA. As such it develops metrics and a valuable method for evaluating the effectiveness of mapping policies on a given benchmark program and proceeds to apply it to a variety of readily implementable schemes. The "operational cost" part of the analysis is mostly neglected here since many operating-system related implementation details have never been specified. For example, the structure of the hierarchy, and thus the number, of resource managers a system would have have not been explored, so we cannot determine the throughput at which system services requests are processed. Also, no resource managers have been written yet in Id, so it is uncertain just how long it would take to allocate or terminate a task. These issues await future research efforts.

5.1 Summary of Significant Results

The initial step towards understanding the complex code-mapping process of the TTDA is to identify some relatively mutually independent phases of it and to examine the effects of each phase and its input variables on the behavior of the system independent of all other factors. These phases are task-partitioning, task-allocation, and instruction-scheduling. The ideal interpreters we have defined enable the effectiveness of the various realistic code-mapping strategies to be evaluated in absolute terms for a given benchmark program.

Observations of results from this study suggest the following:

- Dataflow instruction scheduling choices affect the performance of the machine by at most a factor of two. In general, when scheduling a large number of small tasks, as in this case, the variance in completion time due to scheduling anomalies is not significant (less than 10% in our experiments.)

- Task-partitioning[1] divides the available parallelism of the program into two classes: The interprocessor parallelism, provided by concurrent activities assigned to different tasks, allows the system to be scaled by adding more processors, while the intraprocessor parallelism, provided by those within a task, allows each processor to pipeline its operation and to mask the effects of communications latency. Instruction-level granularity generates heavy network traffic and scatters context information across the entire system, thus

[1] Partitioning for the sake of allocating work to processors; it is not for scheduling of work as on von Neumann processors. Scheduling is still done at the instruction level, by the dataflow instruction scheduling mechanism.

complicating resource management tasks. Iteration- and code-block-level granularity internalize much of the token traffic, but iteration-level produces more even-sized tasks and exposes much more program parallelism at the cost of creating more network traffic relative to code-block level, which also permits context information such as each loop constant to be consolidated on a single processor.

- The goal of task allocations is *not* to insure that all processors perform an equal amount of work; it is not even to minimize the critical path. Rather, since the dataflow multiprocessor never busy-waits or otherwise executes any more operations than the program graph requires, the real goal is to reduce processor idle time, or to maximize processor utilization rate. When there is a sufficient number of tasks (*i.e.*, there are several times more active tasks than the number of processors) throughout most of the program execution, many simple task allocation policies, such as a round-robin scheme, work remarkably well with respect to the theoretically optimal task allocation. But when there is insufficient number of tasks, the statistical variations may cause some processors to be assigned several tasks and others to have no work to do at all. On the other hand, programmer- or compiler-specified task allocation mostly fails because it is very difficult to predict before runtime, due to the asynchronous nature of dataflow computing, when a procedure is actually invoked and when it terminates.

- With the introduction of communications latency, we observe the crossover points between latency and granularity from our model, which demonstrates that an increase in latency should have greater effect on a finer-grained partition, although at lower latency the finer-grained partition would be more scalable by exposing more program parallelism. These crossovers occur at fine granularity and high latency settings, indicating that, given the operating conditions and assumptions of our model and the particular benchmark program used, the optimal granularity should be on a fairly small scale — around the size of an iteration.

5.2 Future Research

As we articulated previously, certain implementation-specific parameters are necessary to complete the cost analysis to determine the particular code-mapping policy best suited for the

architecture. These refer to the design of resource managers mentioned earlier. In addition, the cost of distribution of loop constants and context information when successive iterations are mapped onto distinct processors and the overhead for managing the distributed iterations also depend on the implementation strategy and cannot be usefully estimated. The domain/subdomain approach described in [3] should perhaps be considered more carefully, since our work points to the merits of distributing iterations across the system.

This research uses one kernel throughout for consistency and conciseness of presentation and also because large kernels are difficult to produce and characterize. This method of analysis should be applied to other benchmark programs to get a broad perspective of how adequate a candidate code-mapping policy might be. Similarly, other policies can be devised and analyzed.

Finally, we would like to be able to verify the results of these experiments against a real dataflow machine, although it would not be a substitute for the abstract interpreters introduced here since the abstract interpreters provide us with ideal performance with which we can evaluate the effectiveness of the code-mapping policy implemented on the real machine.

Bibliography

[1] Arvind and J. Dean Brock. Resource Managers in Functional Programming. *Journal of Parallel and Distributed Computing*, 1(1), June 1984.

[2] Arvind and David E. Culler. Managing Resources in a Parallel Machine. In *Proceedings of IFIP TC-10 Working Conference on Fifth Generation Computer Architecture, Manchester, England*, North-Holland Publishing Company, July 15-18 1985.

[3] Arvind, David E. Culler, Robert A. Iannucci, Vinod Kathail, Keshav Pingali, and Robert E. Thomas. *The Tagged Token Dataflow Architecture.* Technical Report, Computation Structures Group, MIT Lab. for Computer Science, 545 Technology Square, Cambridge, MA 02139, August 1983. Revised October, 1984.

[4] Arvind, M. L. Dertouzos, and R. A. Iannucci. *A Multiprocessor Emulation Facility.* Technical Report TR 302, Massachusetts Institute for Technology, Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, October 1983.

[5] Arvind and Kattamuri Ekanadham. Future Scientific Programming on Parallel Machines. In *Proceedings of the International Conference on Supercomputing (ICS), Athens, Greece*, June 1987.

[6] Arvind and K. P. Gostelow. The U-Interpreter. *COMPUTER*, 15(2), Feburary 1982.

[7] Arvind and Robert A. Iannucci. *Instruction Set Definition for a Tagged Token Dataflow Architecture.* Technical Report 212-3, Computation Structures Group, MIT Lab. for Computer Science, 545 Technology Square, Cambridge, MA 02139, February 1983.

[8] Arvind and Rishiyur S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. In *Proceedings of the PARLE Conference, Eindhoven, The Netherlands. (LNCS Volume 259)*, Springer-Verlag, June 15-19 1987.

[9] Arvind and R. E. Thomas. *I-Structures: An Efficient Data Type for Functional Languages.* Technical Report TM 178, Massachusetts Institute for Technology, Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, September 1980.

[10] John Backus. Can Programming Be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs. *Communications of the Association for Computing Machinery*, 21(8):613–641, August 1978.

[11] Stephen A. Brobst. *Instruction Scheduling and Token Storage Requirements in a Dataflow Supercomputer.* Master's thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, MA 02139, May 1986.

[12] M. Campbell. Static Allocation for a Dataflow Multiprocessor. In *Proceedings of the International Conference on Parallel Processing*, pages 511–517, 1985.

[13] E. G. Coffman, Jr., editor. *Computer and Job Shop Scheduling Theory.* John Wiley and Sons, New York, 1976.

[14] W. P. Crowley, C. P. Hendrickson, and T. E. Rudy. *The SIMPLE Code.* Technical Report UCID 17715, Lawrence Livermore Laboratory, February 1978.

[15] Jack B. Dennis. Data Flow Supercomputers. *IEEE Computer*, 48–56, November 1980.

[16] Kattamuri Ekanadham, Arvind, and David E. Culler. *The Price of Parallelism.* Technical Report 278, Computation Structures Group, MIT Lab. for Computer Science, 545 Technology Square, Cambridge, MA 02139, December 1987. Submitted to the Fifteenth Annual International Symposium on Computer Architecture, Honolulu, Hawaii, May 1988.

[17] C. Gao, J. Liu, and M. Railey. Load Balancing in Homogeneous Distributed Systems. In *Proceedings of the International Conference on Parallel Processing*, pages 302–306, 1984.

[18] Michael R. Garey and David S. Johnson. *Computers and Intractability*, pages 236–244. W. H. Freeman and Company, San Fransisco, 1979.

[19] R. L. Graham. Bounds for Certain Multiprocessing Anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[20] P. Henderson. *Functional Programming: Application and Implementation.* Prentice/Hall International, Englewood Cliffs, New Jersey, 1980.

[21] Kei Hiraki, Satoshi Sekiguchi, and Toshio Shimada. *Load Scheduling Schemes Using Inter-PE Network.* Technical Report, Computer Systems Division, Electrotechnical Laboratory, 1-1-4 Umezono, Sakura-mura, Niihari-gun, Ibaraki, 305, Japan, 1987.

[22] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and Wolfe M. Dependence Graphs and Compiler Optimizations. In *Proceeding of ACM Symposium on Principles of Programming Languages*, January 1981.

[23] Robin Milner. A Proposal for Standard ML. In *Proceedings of the ACM Symposium on LISP and Functional Programming*, pages 184–197, The Association for Computing Machinery, Inc., New York, August 1984.

[24] Rishiyur Sivaswami Nikhil. *Id Nouveau Reference Manual, Part I: Syntax.* Technical Report, Computation Structures Group, MIT Lab. for Computer Science, 545 Technology Square, Cambridge, MA 02139, April 1987.

[25] Rishiyur Sivaswami Nikhil. *Id World Reference Manual.* Technical Report, Computation Structures Group, MIT Lab. for Computer Science, 545 Technology Square, Cambridge, MA 02139, April 1987.

[26] G. F. Pfister and V. A. Norton. Hot Spot Contention and Combining in Multistage Interconnection Networks. *IEEE Transactions on Computers*, C-34(10):943–948, October 1985.

[27] Kenneth R. Traub. *A Compiler for the MIT Tagged-Token Dataflow Architecture*. Technical Report LCS TR-370, Massachusetts Institute for Technology, Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, August 1986. (Master's Thesis, Dept. of Electrical Engineering and Computer Science, MIT).

This blank page was inserted to preserve pagination.

CS-TR Scanning Project
Document Control Form

Date : 3/17/95

Report # LCS-TR-425

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 87 (75-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☒ Single-sided or
☐ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☐ Unknown ☐ Other:_____

Check each if included with document:

☒ DOD Form (2) ☐ Funding Agent Form ☒ Cover Page
☒ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages (by page number): PAGE AFTER i, ii, PAGES 4, 46

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description :	Page Number:
IMAGE MAP (1-6) UN#, TITLE, ACK, PAGES #'ED i, ii, iii, BLANK	
(7-87) PAGES #'ED 1-81	
(88-92) SCANCONTROL, COVER, SPINE, DOD(2)	
(93-95) TRGTS	

Scanning Agent Signoff:
Date Received: 3/17/95 Date Scanned: 3/22/95 Date Returned: 3/23/95

Scanning Agent Signature: Michael W. Cook

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) MIT/LCS/TR-425	5. MONITORING ORGANIZATION REPORT NUMBER(S) N00014-84-K-0099
6a. NAME OF PERFORMING ORGANIZATION MIT Laboratory for Computer Science 6b. OFFICE SYMBOL (*If applicable*)	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research/Department of Navy
6c. ADDRESS (*City, State, and ZIP Code*) 545 Technology Square Cambridge, MA 02139	7b. ADDRESS (*City, State, and ZIP Code*) Information Systems Program Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA/DOD 8b. OFFICE SYMBOL (*If applicable*)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (*City, State, and ZIP Code*) 1400 Wilson Blvd. Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (*Include Security Classification*)
Code-Mapping Policies for the Tagged-Token Dataflow Architecture

12. PERSONAL AUTHOR(S)
Maa, Gino K.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (*Year, Month, Day*)	15. PAGE COUNT
Technical	FROM _____ TO _____	1988 June	81

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*)
FIELD	GROUP	SUB-GROUP	Dataflow, Multiprocessors, Code Mapping, Program Partitioning, Processor Allocation, Scheduling, Granularity, Multiprocessor Performance Scalability, Parallel Computing

19. ABSTRACT (*Continue on reverse if necessary and identify by block number*)

Multiprocessing seems to be the only viable way to gain significant speedup beyond that afforded by performance advances in semiconductor devices and hardware construction, which are beginning to face the limitations of physics. Although it is relatively easy to improve the "raw" computational performance of a system simply by adding more processors to it, the far more difficult task is to insure that the additional resources actually reduce a program's computing time. Thus, the ultimate success of multiprocessing as a means of increasing computation speed rests on the ability to parallelize computation: to partition, allocate, distribute, and schedule work efficiently amongst the large collection of available system resources, while maintaining a high rate of utilization of these resources.

To keep many processors busy with work, it is necessary that the program has enough concurrent operations to map to those processors. But given that there is much concurrency in many large programs, the onus lies on the code-mapping process to maintain as much (cont.)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Judy Little, Publications Coordinator	22b. TELEPHONE (*Include Area Code*) (617) 253-5894 22c. OFFICE SYMBOL

☆U.S. Government Printing Office: 1985—507-047

19. concurrency as is feasible without compromising its efficiency. This goal maximizes the scalability of the system. Our study is based on a dataflow computer, the MIT Tagged-Token Dataflow Architecture (TTDA).

This study focuses on analyzing the effectiveness of the code-mapping policies for the TTDA. It develops metrics and practical method for evaluating the effectiveness of mapping policies on a given benchmark program and proceeds to apply it to a variety of readily implementable schemes. This approach provides answers to the following important questions:

- What is the maximum gain achievable by any code-mapping strategy and, therefore, whether it is worthwhile to seek a more sophisticated strategy?
- If some of the processors are idle, does it mean that the program lacks sufficient parallelism, and therefore either should be rewritten or perhaps is entirely unsuitable for a multiprocessor, or that the code-mapping process is too inefficient?
- What is the effect of communications latency, an inherent part of all multiprocessor systems, on the performance of the system and the code-mapping strategy?