LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT DATAFLOW

A Parallel Computing System
based on
The Monsoon Architecture and the Id Programming Language

(Extracts from March 1988 DARPA Proposal)

Computation Structures Group Memo 285

March 25, 1988

Arvind
Michael L. Dertouzos
Rishiyur S. Nikhil
Gregory M. Papadopoulos

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Proposal to DARPA

Executive Summary

The Laboratory for Computer Science at the Massachusetts Institute of Technology proposes to prototype a *Parallel Dataflow Computer System* for high-performance, general-purpose computing, based on the following system components and development path.

1. System Components

- Processing Elements (PEs) with a new dataflow architecture called Monsoon.
  - A 6-MIPS Monsoon accelerator board (single PE) for Sun workstations.
  - A 48-64 MIPS accelerator board (4 PEs in ASICs) for Sun workstations.
  - A 256-PE multiprocessor (with the help of an Industrial Partner).
- A high-speed multistage interconnection network, based on 4×4 network routing chips.
- A high-level, declarative, parallel language Id, with a compiler.
- Resource Managers (basic operating system) for the multiprocessor.
- A programming environment for preparing, compiling, running and debugging Id programs on the multiprocessor.

The system is designed to be scalable in performance, both by using more aggressive technology as well as by increasing its size.

2. Development Path

Following discussions with DARPA, we have chosen a development path that facilitates dissemination of this new technology to the manufacturing and user communities, and is characterized by:

- *Staged Contruction:* Single-PE Monsoon accelerator boards will be built by 12/88 and will be distributed to users. A sixteen-PE system with Monsoon in ASICs will be built by 12/89. The 256-PE system should be functioning by 1990. Throughout, the software will be ahead of the hardware development.
- *Industrial Participation and Transfer of Technology:* The first two stages (up to the 16-PE system) will be done by MIT LCS, subcontracting to industry for production of pc boards, design and production of ASICs. The hardware for the third stage (256-PE system) will be done in direct collaboration with an Industrial Partner chosen through an open "bidding" process, who will be responsible for packaging and producing the large system and subsequent copies.
- *Development of User Community:* The single PE accelerator and the 16 PE multiprocessor will be available to users with Sun workstations. Id World, the programming environment, is written in Common Lisp and the first version has already been distributed for Lisp machine use. A Unix version of Id World for Sun workstations will be available soon. The Id graph-interpreter is being re-written in C to run on other machines, including supercomputers. Preliminary manuals for Id and Id World have been produced. A book on Id programming and the dataflow approach to parallelism is under way.

Direct funds are requested for the completion of the first two stages, i.e., through the construction of the 16-PE multiprocessor. Funding options are provided for the third stage, *i.e.,*

the 256 PE system which will also be funded by our Industrial Partner. Some initial funds are already available under the existing multiprocessor contract. A meeting with prospective Industrial Partners is being organized in early 1988 to launch the selection process.

3. Background

Two major goals of the proposed project are very ambitious, but are supported by many years of careful research at our laboratory:

1. At the hardware level, to demonstrate that data-driven instruction scheduling exploits significantly more parallelism than other methods, and is essential for scalable performance.
2. At the software level, to demonstrate that declarative languages with implicit parallelism are as great a prospective step (and as unquestionably essential) in parallel computing, as was the step from assembly languages to FORTRAN in sequential computing.

For an application to run on a multiprocessor, it must be divided into multiple concurrent threads. For more processors, the division must be into more, and smaller threads. Except for a few applications with regular structure, threads must communicate with each other in complex ways. For scalable performance, therefore, *a)* the overhead of switching between threads must be low, and *b)* the synchronization cost (for matching communicated information with waiting threads) must be low. The proposed Monsoon architecture *uniquely* addresses these issues, in hardware.

Augmenting an existing language with a set of concurrency primitives is a good interim solution to the programming question, and can lead to effective results. The proposed dataflow system is certainly programmable from concurrent extensions of FORTRAN, C or Lisp. However, a future programmer of parallel machines should not be preoccupied with handling all the tedious details of concurrency and architecture. The proposed Id language offers a unique solution to this problem, because of its implicit, fine-grained parallelism.

We believe that the Monsoon architecture is a good target for, say, concurrent FORTRAN, and that Id is an appropriate language for non-dataflow parallel machines. However, we are focusing on implementing Id on Monsoon because it offers maximum parallelism, and because we have a deeper understanding of the compilation problem. The ultimate success of this approach can only be determined if there is a total system with enough high performance and adequate programmability to be used practically and effectively by a broad community. This is precisely the goal of the MIT's Project Dataflow.

Section B: Innovative Claims

The prototype Dataflow Computer System will comprise: a high-performance, MIMD machine based on a new dataflow processor architecture called Monsoon; a high-level, general-purpose parallel programming language called Id; basic resource managers for managing parallelism, and a program development environment for Id. The system will be *general-purpose*, supporting numeric and symbolic computing simultaneously (the expected norm in the future). With parallel I/O, it will also support high-performance databases.

Architecture

The proposed machine has an *asynchronous* model of parallelism, in contrast to synchronous models such as SIMD and systolic arrays, which exhibit high performance on a smaller class of applications. Compared to other asynchronous MIMD approaches, the proposed machine offers the following advantages:

- A more general, flexible and configuration-independent programming model than MIMD machines based on message-passing.
- Better *scalability* than MIMD machines based on von Neumann processors. Unlike those machines, the prototype is based on data-driven (dataflow) instruction-scheduling, with support for cheap hardware synchronization. This is necessary to tolerate increased memory and communication latencies, and for full exploitation of dynamic producer-consumer parallelism.
- A significant dataflow innovation— *directly-addressed* Wait-Match memory. Previous dataflow architectures have proposed or used a large associative memory implemented using hashing. The new solution is now feasible due to techniques we have developed recently in controlling parallelism. It also permits unifying Wait-Match memory and I-structure memory, resulting in easier resource-management and lower production cost.

Performance of Monsoon will scale with implementation technology, since processor and communication components are balanced under "constant" technology.

Language

Id is a practical language today, with a compiler largely developed and in place. While our approach requires programmers to learn and use a new language, it has the following advantages over those that add parallel constructs to an existing language:

- *Implicit* parallelism and determinate semantics, relieving the programmer from managing parallelism explicitly and of concerns about specifics of machine configuration and timing. Debugging is therefore relatively straightforward.
- Greater expressiveness, due to higher-order procedures and non-strict data structures. We believe that an experienced programmer can write significantly shorter programs in Id than is possible in current high-level languages like Common Lisp.
- Greater applicability across parallel architectures, in the same sense that FORTRAN became a general language for von Neumann machines, though it was developed for the IBM 650.

In short, we believe that Id will mitigate, not aggravate, the software crisis in the move to parallelism.

Resource Management

Very little is known today about the significant problem of resource management in large, general-purpose parallel computers. We expect preliminary solutions through our proposed work. Further, the system will be one of the first adequate testbeds for more experimental study of these pivotal issues.

Relation To Other Approaches

Though there are numerous multiprocessor projects worldwide exploring architectures, programming languages and applications, very little is known, in a scientific sense, about "best" approaches and "ultimate" limits. An increasing number of computer architects are striving for high-performance multiprocessor *hardware*, while deferring the software question. Japan is close to realizing a parallel dataflow machine, but without the architectural innovations of our proposed machine, and without a general software solution.

While we too have much to learn, we believe that our proposal stems from perhaps the longest background of careful multiprocessor studies and experiments (over ten years), resulting in a total (*i.e.*, hardware *and* software) solution based on a solid technical foundation.

Experience has shown that advertised *peak* instruction and floating-point operation rates are often misleading measures of a machine. A more realistic (and often quite different) characterization emerges by examining the *range of applications* for which it is suitable, the *ease of programming* such applications, and the *typical sustained rates* on such programs.

It is our long history of dataflow research that gives *technical support* (elaborated in Section G) to our claims that the proposed dataflow system:

(1) is as general-purpose as current sequential machines,
(2) is at least as easy to program as current sequential machines, and
(3) will consistently demonstrate sustained performance that truly attains the promise of multiprocessors.

Section C: Deliverables

The results from this project will be a significant increase in understanding fundamental principles of parallel computing, as well as a set of concrete high-performance parallel hardware and software systems. We expect much leverage by distributing the proposed system in stages to a large external user community.

Expected Results:

- We hope to demonstrate that the dataflow approach provides mechanisms:
  - for efficient synchronization of parallel activities, and
  - that tolerate increased memory and communication latencies.

  Both are necessary for any successful MIMD machine.
- We hope to show that in moving to parallel computing:
  - it is *possible* to raise the level of programming without sacrificing efficiency, and
  - it is *necessary* to use higher-level declarative languages like Id to be able to compile good parallel code.

  This is crucial in any parallel machine, since increased performance will inspire larger and more complex applications.
- We hope to show significant solutions to some of the basic resource-management problems of MIMD machines, about which almost nothing is known today.

Expected Deliverables:

A prototype of a complete Parallel Dataflow Computer System, built in three stages. By such staging, we immediately begin building a large external user community, giving them progressively improved systems to experiment with. We describe here mainly the deliverables as seen by such external users. A more detailed internal schedule is described in Section D:

Stage 1: 2/89. *Hardware:* A 6 MIPS dataflow accelerator board on VME bus for Sun workstations. The board will implement a single Monsoon PE . *Software:* Id language manuals, compiler, Id World programming environment to edit/compile/debug/study parallelism behavior.

6/89. *Hardware:* 16-PE multiprocessor for internal use.

Stage 2: 10/89. *Hardware:* A 48-64 MIPS dataflow accelerator board on VME bus for Sun workstations. The board will implement four Monsoon PEs and an I/O subsystem in ASICs. *Software:* Resource managers for small number of PEs.

2/90. *Hardware:* 16×16 Network Board (VME/Sun) based on 4×4 routing chip. With 4 PE boards, makes a 16 PE system. *Software:* same.

Stage 3: 8/90-2/91. *Hardware:* 256-PE machine, designed and built in collaboration with an Industrial Partner. Over 2000 MIPS peak, 1000 MIPS sustained. *Software:* Resource managers to run on large number of PEs.

The hardware will be easily reproducible and distributable. Both hardware and software will be distributed at cost under license.

Section D: Schedule and Milestones

The prototype system will be built in three overlapping stages:

1. Single PE (processing element) accelerator board for VME bus on Sun workstations,
2. 16 PE system (PEs in Integrated Circuits), and
3. Full 256 PE system.

where each PE is a complete dataflow computer including processor and memory.

In early 1988, we plan to hold a meeting for the purpose of identifying an Industrial Partner who will help fund and cooperate in building the third stage of the system. Options to that end are included in the budget section.

Stage 1: "Single-board Single PE dataflow accelerator"

6/88 *Hardware:* Wire-wrap Monsoon PE prototype on NuBus/Explorer. 4×4 Network Router Chip. *Software:* Monsoon Code Generator for Id compiler. Single-processor resource-managers. Id World ported to Suns.
8/88 *Hardware:* Network link chip and 4×4 network card (for VME).
11/88 *Hardware:* VME/Sun printed-circuit version of PE.
2/89: Production version of pc-board and software, *distributed to external users.* Basic I/O drivers for internal use.
6/89 *Hardware:* Internal use of 16 PE multiprocessor with four Suns, four PE cards and two 4×4 network cards per Sun. *Software:* I/O drivers, Resource managers for small multiprocessor.

Our initial wire-wrap prototype is on the NuBus because all our software currently runs on TI Explorers, and we have already begun prototyping components on the NuBus. We are targeting all subsequent development to the VME bus because we believe that the Sun workstation will be a better host for widespread distribution.

Stage 2: "5-board 16 PE (ASIC) Dataflow Small Multiprocessor"

8/88 *Hardware:* Begin development of ASIC version of Monsoon PE.
10/89 *Hardware:* ASIC Monsoon card ready (VME/Sun). 16×16 network card ready (VME/Sun).
2/90 *Hardware:* 16 PE multiprocessor, *distributed to external users.* 4 cards (4 PEs each) and one 16×16 network in a Sun. *Software:* Id compiler, Id World programming environment on Suns, small multiprocessor resource managers.

Stage 3: "256-PE Dataflow Large Multiprocessor".

11/88 *Hardware:* Begin development with external industrial partner on 256-PE dataflow system. *Software:* Study large-multiprocessor resource-management problems.
2/90: Primitive large-multiprocessor resource managers ready.
2/91 *Hardware:* 256-PE system. Improved large-multiprocessor resource managers.

Section E: Proprietary Claims

There are no proprietary claims by any external party on the ideas or products of this project— MIT and the principal researchers named in this proposal have full control over all intellectual and material rights.

Section F: Statement of Work

The MIT Laboratory for Computer Science will design and construct a prototype Parallel Dataflow Computer System in three stages, simultaneously building a substantial external user community. The following is a statement of work for all three stages, with the third stage being based on exercise of associated options.

In general, we are resisting the temptation to scale performance through faster technology (such as very high-speed integrated circuits), in favor of architectural innovation. Such scaling is always possible later. However, even though the machine is a prototype, its size demands precision and professional manufacturing standards. Thus, it will be easily reproducible and distributable to external users.

Stage 1: "Single-board Single PE dataflow accelerator"

- 6/88. *Hardware:* We will build a wire-wrap Monsoon PE prototype on a NuBus card (for TI Explorer workstation), with the following characteristics: 6 MIPS, 128KW Token Store, 128 KW Instruction Store, 128 K-entries Token Queue, floating point, no caches. Completely micro-programmable, with microcode in RAMs. We will complete the 4×4 Network Router Chip, which is already under development.

  Software: We will implement extensions to the existing Id language to include non-deterministic and imperative constructs necessary for resource-managers, making it a complete systems language. We are already experimenting along these lines.

  We will incorporate extensions to the existing Id compiler to support the above extensions. We will incorporate several known optimizations, and develop new optimizations.

  We will implement a Monsoon Code Generator for the Id compiler (which currently generates code for the MIT Tagged-Token Dataflow Architecture).

  We will implement single-processor resource-managers.

  We will port the Id World programming environment (currently running on Lisp machines) to Sun workstations. This work is already in progress.

  We will extend Id World to handle the new language extensions, the new Monsoon instruction set, and control of resource manager parameters.

- 8/88. *Hardware:* We will design and implement the network link chip and a 4×4 network card.

- 11/88. *Hardware:* We will produce the VME/Sun printed-circuit version of the Monsoon PE accelerator board. Its characteristics: 6 MIPS, 2 MW Token Store and Instruction Store, 128 K-entries Token Queue, floating point, caches, I/O subsystem, no network interface. Still completely micro-programmable. PC-board construction will be sub-contracted out. Manufacturing the cards will be set up as a turnkey line, due to the number of processors required. This involves full assembly documentation, qualified component vendor bill of materials, and in-circuit and functional test fixtures.

- 2/89. The production version of the pc-board PE and its software will be *distributed to external users.*

- 2/89. *Software:* Internally, we will complete basic I/O drivers. We do not want to spend time writing I/O services such as file systems. Thus, we will exploit the host SUN services as much as possible.

- 6/89. *Hardware:* For internal use, we will construct a 16 PE multiprocessor with four Suns, each with four PE cards and two 4×4 network cards.

  Software: We will write resource managers for the small multiprocessor in Id. The emphasis is on single programs, not multi-programming. The main issues: controlling program unfolding, load balancing, and heap storage management.

Stage 2: "5-board 16 PE (ASIC) Small Dataflow Multiprocessor"

- 8/88. *Hardware:* We will begin development of an ASIC version of the Monsoon PE. This will either be subcontracted out, or we will hire a designer.

- 10/89. *Hardware:* We will produce the ASIC version of the Monsoon PE card. Its characteristics: 12-16 MIPS per PE, with 4 PEs, an I/O system, and a network interface on each card. The design will be reproducible, for Stage 3.

  We will produce the 16×16 network card (VME/Sun). It will be implemented in impedance-controlled printed circuit technology, due to the high-speed nature of the digital and analog circuitry. Cable assemblies will be manufactured by a quality full-capability commercial outfit.

- 2/90. We will distribute a 16 PE multiprocessor system *to external users*. It will consist of *Hardware:* 4 cards to plug into a Sun workstation: 4 cards with 4 PEs each and one 16×16 network card, and *Software:* the Id compiler, Id World programming environment on Suns, small multiprocessor resource managers.

Stage 3: "256-PE Large Dataflow Multiprocessor". This will be built in cooperation with an Industrial Partner.

- 3/88. We will hold a meeting in which we will invite several potential Industrial Partners and present our plans. The objective is to attract active, competitive participation in contruction of the large dataflow multiprocessor.

- 8/88. We prepare and submit a joint funding proposal for Stage 3 with the Industrial Partner.

- 11/88. *Hardware:* Begin development with industrial partner on hardware of 256-PE dataflow system. *Software:* Study large-multiprocessor resource-management problems.

- 2/90. *Software:* Primitive large-multiprocessor resource managers ready.
- 2/91. *Hardware:* 256-PE system. Improved large-multiprocessor resource managers.

For resouce-management efforts, the emphasis is on single programs, not multi-programming. The main issues: controlling program unfolding, load balancing, and heap storage management.

The Industrial Partner will be involved only in the hardware effort. MIT will be responsible for the entire software effort.

Section G: Technical Rationale

Dataflow graphs were invented over a decade ago by Professor Jack Dennis at MIT as a radical, but exciting parallel computation model. Dataflow graphs may be regarded as a parallel *machine* language, because they can be executed directly by hardware. Many issues required exploration:

- What conditions are required for a parallel machine language to be well-behaved (determinate)?
- What are the characteristics of a *general-purpose*, high-level language for parallel computation?
- How should such high-level languages be compiled to parallel machine code?
- What are the architectural requirements for efficient execution of a parallel machine language?
- What are the architectural requirements for *scalable* performance?
- What is the dynamic behavior of realistic applications written in such high-level languages and executed on such architectures?

In the MIT Tagged-Token Dataflow Project, our research approach has been first to obtain a substantial understanding of these issues before declaring ourselves ready to build hardware. Based on extensive and careful experimentation, today we have solid answers to many of these questions. Some of our results are recent (within the last few years), because prior to that we did not have adequate computing power to simulate realistic applications.

We believe that we are unique in the extent to which we have built technical foundations to support our proposed system. For example, instead of under-estimating programming issues and predicting performance on the basis of clock speeds and number of processors, we base our claims on detailed study of a *total* system— a programming language that has undergone substantial development and has been learned easily by non-specialists, a wide range of applications, extensive experience with dataflow compilers, and simulations that incorporate accurate and realistic accounts of latency and synchronization costs.

In Section 1 we present results from this substantial research history (this is a greatly condensed version of References [1, 8, 15, 16]). Then, in Section 2, we present the proposed Dataflow System, which is based on these results.

1 MIT Tagged-Token Dataflow Project Results

We first describe our research tools, and go on to show three important results:

- Id is an expressive and compilable language for parallel programming.
- There is sufficient parallelism in typical programs for machines comparable to the proposed one to be utilized effectively.
- The Tagged-Token Dataflow Architecture (of which Monsoon is a realization) is effective in exploiting this parallelism and tolerating communication latencies.

1.1 Research Tools

For 6 to 8 years our studies of parallel computing have focused on:

- Id, a high-level language which is a superset of a full, higher-order, non-strict functional language,
- Dynamic Dataflow Graphs— a parallel machine language, and
- The MIT Tagged-Token Dataflow Architecture (TTDA) that executes dataflow graphs directly.

We have studied them extensively using these tools:

- *Compilers:* For over 5 years we have had a compiler to translate Id into dataflow graphs. It has gone through three major versions, in step with advances in the language and the architecture. The structure of the current compiler is extremely modular to facilitate experiments in language design and compilation techniques[24].

  All our simulation and emulation tools (see below) execute identical object code.

- *Simulator:* For over 2 years we have had a detailed event-driven simulator for the TTDA, allowing us to study the relative speed and capacity required for various TTDA components[12].

- *Emulators:* For about 2 years we have had a heavily instrumented emulator for the TTDA called GITA (Graph Interpreter for the Tagged-Token Architecture). It can be run in two modes: emulating multiple TTDA processors on a single physical workstation, or on the MEF (Multi-Processor Emulation Facility), an actual multi-processor[5]. The MEF consists of 32 TI Explorer Lisp Machines connected by a multistage high-bandwidth communication network. We constructed the MEF under DARPA funding specifically to facilitate dataflow and other parallel processing research.

- *Programming Environment:* For over a year we have had a complete programming environment called Id World, including an Id editor, the Id compiler, a symbolic, parallel debugger, GITA, and facilities to collect and display emulation statistics [23].

  (In response to several requests, Id World was released for external use in April 1987. It is available under license from MIT.)

Our confidence in the proposed system stems from the deeper understanding of the strengths and weaknesses of the TTDA and Id. As we shall show, our studies have both confirmed their basic viability as well as revealed remaining implementation issues.

1.2 Id is a Good Language for Parallel Programming

In any parallel programming language, the programmer should be able to express parallelism without undue burden, and the compiler should be able to compile code with sufficient

parallelism. Ideally, none of the available parallelism should be obscured in going from abstract algorithm to program to machine code.[1]

1.2.1 Id is Expressive

Id [22] is a higher-order, non-strict, functional language, augmented with dynamically-allocated, parallel data structures called I-structures. These features elevate coding to a level much higher than, say, in Common Lisp. We have written and studied both scientific and symbolic applications in Id.

The Id programmer does not have to partition a program into parallel tasks and manage their synchronization, because the parallelism is *implicit* in the operational semantics of Id. The programmer is insulated from details of the architecture such as number of processors and component speeds. Id programs are *determinate*— outputs depend only on inputs, and not on runtime scheduling. This is invaluable in debugging— the programmer has a simple, time-independent model of computation.

I-structures, used as arrays, are essential for scientific computing [11]. The codes we have written include SIMPLE, a hydrodynamics and heat-conduction code, and PIC (Particle in a Cell), an electro-dynamics code. To support these, we have written libraries of scientific utilities, such as matrix multiplication, LU decomposition, and various transcendental functions. We have been able to demonstrate that Id permits coding scientific applications at a very high level [6]. For example, SIMPLE is expressed succinctly in 550 lines of Id (including libraries), compared to 1500 lines of FORTRAN. Further, the Id code has an almost 1-to-1 correspondence with the mathematical description of the problem.

For symbolic computing, Id has symbols and flexible data structures such as lists and generic types. We have written a polynomial algebra package, DNA sequence matching, various search and sort routines, *etc.* in Id.

1.2.2 Id is Compilable

Unlike imperative languages, translating Id code into dataflow graphs is straightforward [24].[2] Functional, determinate semantics also enables several powerful optimizations, such as loop-constant detection, constant folding, inline substitutions and fast function calls. Compiling is simplified because the object code is independent of machine characteristics such as size (number of processors) and speed of components (memories, networks, *etc.*).

1.2.3 Id Does Not Obscure Parallelism

Parallelism Profiles Before we can determine whether a language obscures parallelism or not, or whether an architecture exploits parallelism or not, we need a reference point, a

[1] Of course, it is always possible to change the algorithm itself to increase parallelism. The focus here is on coding issues only.

[2] This does not mean that Id can be executed easily on any parallel machine. Executing dataflow graphs efficiently requires architectural support, such as I-structure storage[3].

way to judge how much parallelism there is to begin with in an algorithm. For this, we use a *Parallelism Profile*, a graph showing the number of instructions that *could* be executed at each time step.

To obtain this profile, we first code it in Id and compile it into dataflow graphs. Then, we execute it on GITA, our graph interpreter, under the following (ideal) assumptions:

- Each node (instruction) in the dataflow graph takes one time unit to execute,
- The results produced by a node are available at the destination nodes (successors in the graph) instantaneously, and
- Each node *fires* (is executed) as soon as it has the required input operands.

The parallelism profile, constructed by GITA, is the function $pp(t)$ which gives the number of nodes fired at each time step. The time step beyond which $pp(t)$ is uniformly zero is called the *critical path*, *i.e.,* 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 (in our dataflow model, this number does not vary with machine configuration, *e.g.,* the number of processors). Many details such as number of processors, locality, contention, and distribution of work are abstracted away entirely. Thus, these profiles capture the parallelism *inherent* in the program.

Figure 1 gives the parallelism profile for matrix multiplication of two 16×16 matrices using the traditional algorithm. The upper profile counts *all* operations, while the lower profile counts only the floating point operations. The critical path is 295 and the total number of operations is 72,513, out of which 8,192 are floating point operations. The bell-shape arises because the unfolding of loops is staggered slightly, and the summation in the innermost loop is done sequentially.

1.2.4 Id Also Exploits Parallelism Adaptively

Dataflow scheduling reveals additional parallelism in subtle ways. Consider two programs: `vsum` to compute the vector sum, and `ip` to compute the inner-product of two vectors. Then, consider composing them together, *e.g.,*

```
Def ip_vsum A B = ip (vsum A A) (vsum B B) ;
```

With most conventional languages, the best one could expect is that the critical path for `ip_vsum` would be the sum of the critical paths of `ip` and `vsum`.

In Id, however, functions and data structures are *non-strict*, so that each `vsum` can return the descriptor of its result vector as soon as it is allocated, even before it has written all its components. Thus, `ip` can begin work immediately. Because of I-structure semantics, there is no race between the writes in `vsum` and the reads in `ip`. Thus, `vsum` and `ip` are automatically *pipelined*, working in tandem as producer and consumer respectively.

Experiments on GITA confirm this, as shown in Figure 2. The parallelism profile of the `ip_vsum` is not obtained by stringing out the profiles of `vsum` and `ip`— instead, they are

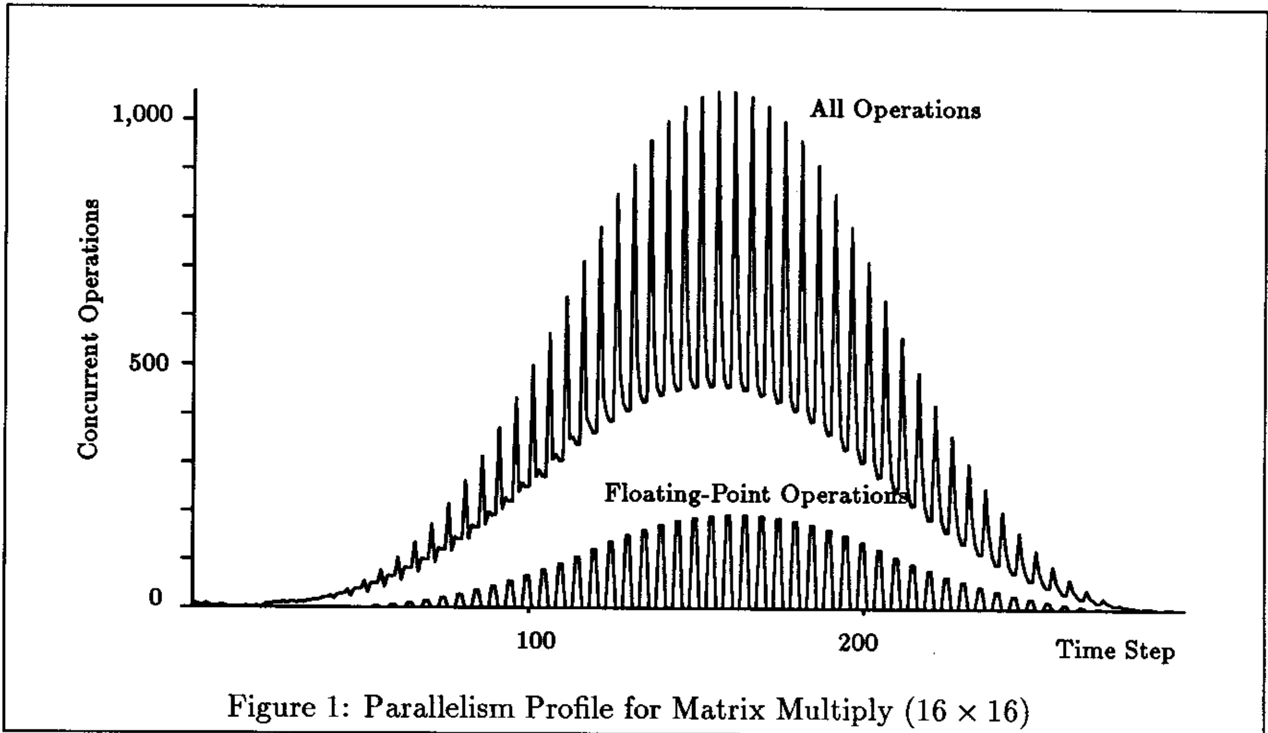

Figure 1: Parallelism Profile for Matrix Multiply (16×16)

overlapped. On vectors of size 10, the critical paths for `vsum` and `ip` alone are 63 and 59 respectively, while the critical path for `ip_vsum` is only 76 (much less than $63 + 59$).

It is important to note that this behavior is achieved *automatically*, without any optimizations (like loop-jamming). In fact, there was no change in the source or object codes of `vsum` and `ip`.

1.3 There Is Sufficient Parallelism in Existing Programs

An important question is whether or not there is sufficient parallelism in typical programs to keep (say) 256 processors busy. Interpreting the literature is difficult because of variations in the "unit" of parallelism (granularity), the language, the compiler algorithms, the number of processors, and the size of the problem itself.

We have studied this issue systematically. In the following sections, we look at one application— the SIMPLE code, a hydrodynamics and heat flow code kernel that has been studied extensively both analytically [13] and by experimentation. We first show the parallelism profile under idealized assumptions (infinite number of processors, zero communication latency), and then under realistic assumptions.

1.3.1 Inherent Parallelism in SIMPLE

Figure 3 shows the parallelism profile of 3 iterations of SIMPLE on a 20×20 mesh (an actual simulation typically performs 100,000 iterations on 100×100 mesh). The critical path is

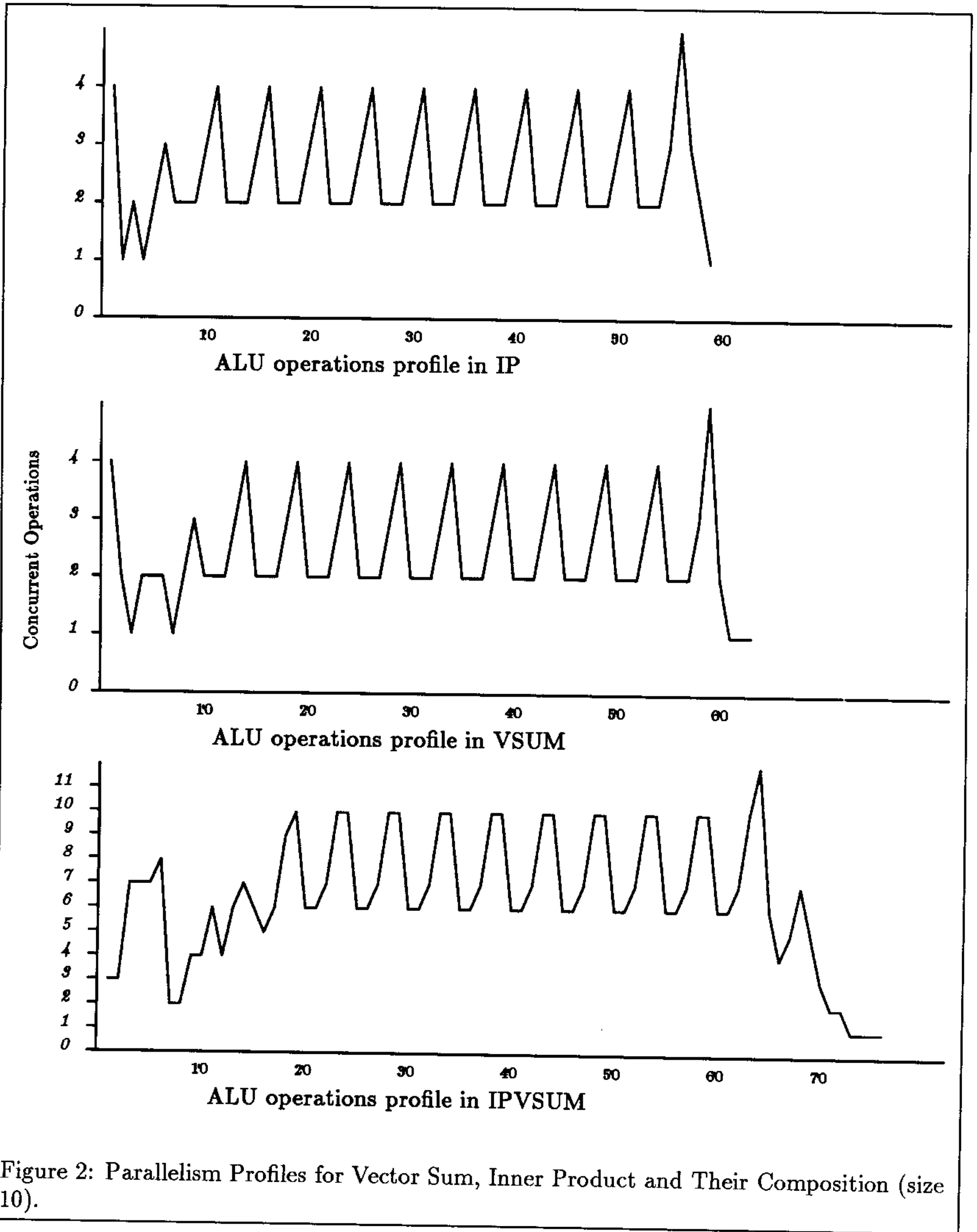

Figure 2: Parallelism Profiles for Vector Sum, Inner Product and Their Composition (size 10).

1,976 and the instruction count is 1,471,374. Note that there is no significant parallelism between outer loop iterations. The potential parallelism varies tremendously within each iteration (typical of even the most highly parallel programs, in our experience).

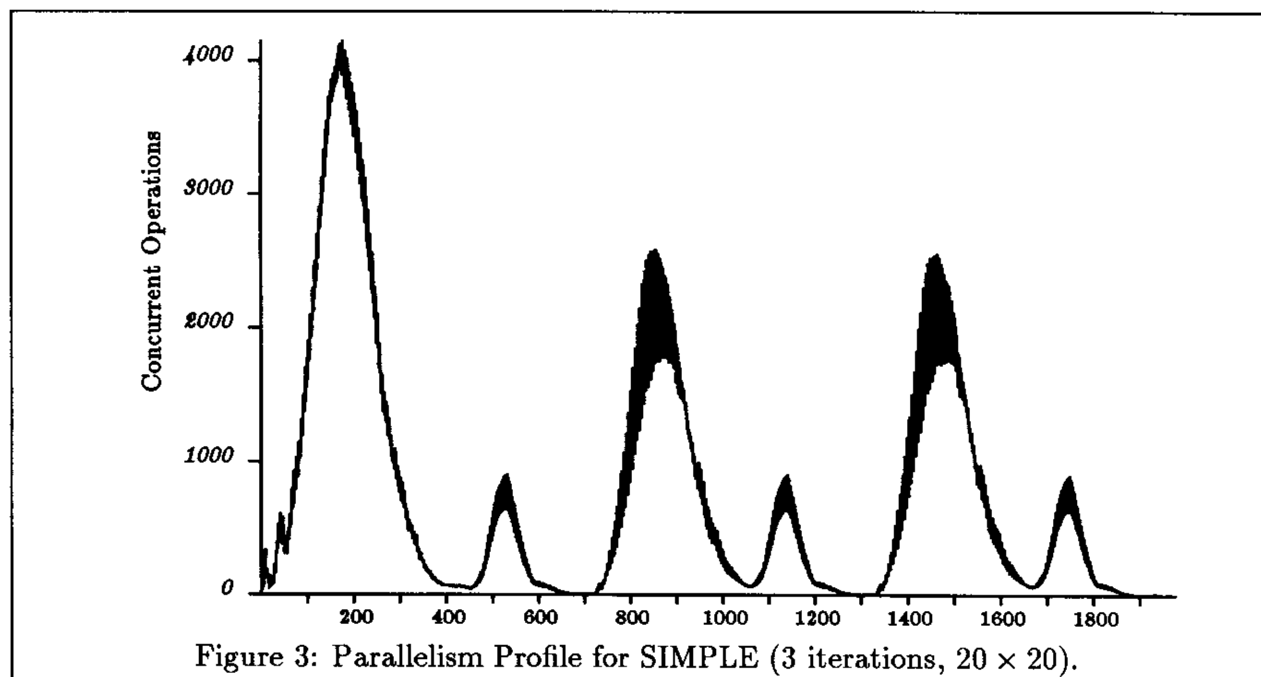

Figure 3: Parallelism Profile for SIMPLE (3 iterations, 20 × 20).

The shape of the profile for SIMPLE does not change significantly with the size of the problem.

1.3.2 Parallelism on a Finite Number of Processors

The critical path of a computation with *tot* operations on *n* processors can vary between $\lceil \frac{tot}{n} \rceil$, if there are no data dependencies and 1, if there is a completely sequential data dependency.[3] Thus, data dependencies limit parallelism.

Figure 4 shows the profile for SIMPLE on 1,000 processors, generated by constraining GITA to execute no more than 1,000 operations at each step. The critical path is 2,763 (compared to 1,976 for infinite processors). Of course, finite-processor profiles are also sensitive to the choice of the particular *n* operations at each step, but our experiments indicate that except in pathological cases, this variation is negligible.

The parallelism in a program can be summarized in terms of

$$speedup(n) = \frac{t(1)}{t(n)} \qquad \text{and} \qquad utilization(n) = \frac{t(1)}{n \times t(n)}$$

where $t(n)$ is the time to execute on n processors. $t(1)$ is simply the total number of operations executed, *i.e.*, the area under the parallelism profile.

[3] We use the term "processors" loosely here; all we mean is that no more than n ALU operations can be performed at each step.

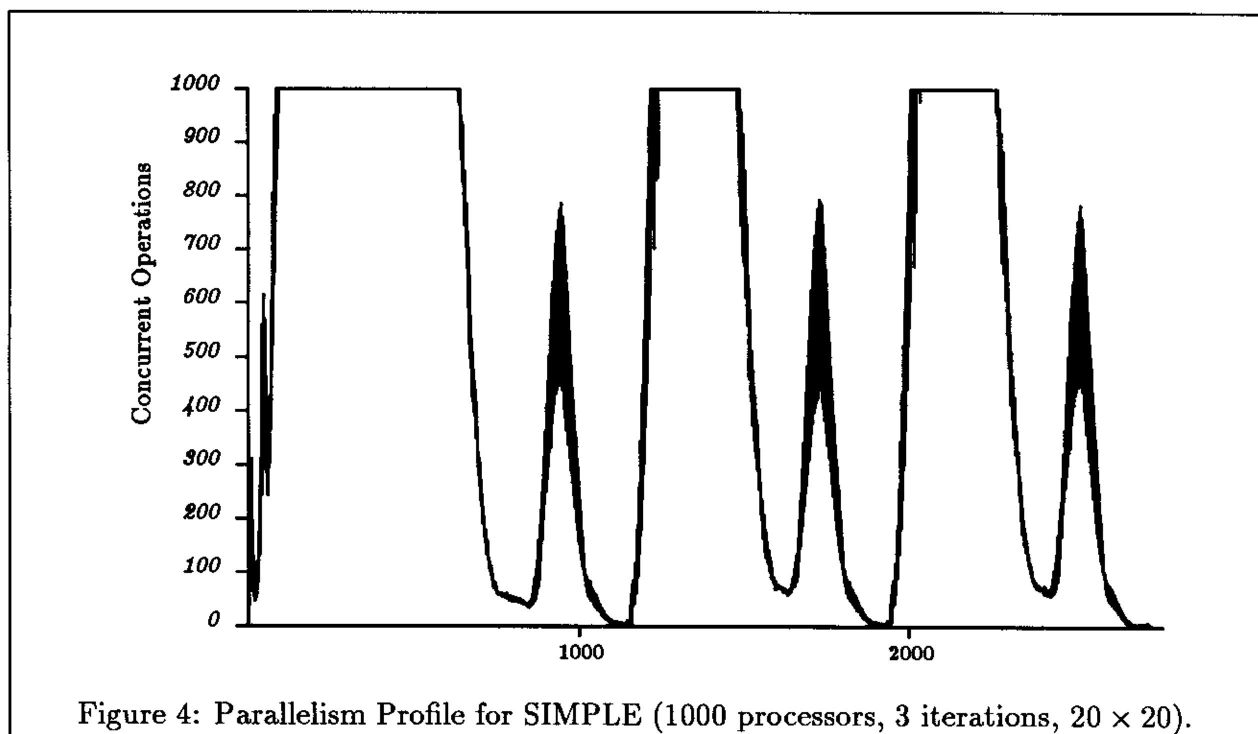

Figure 4: Parallelism Profile for SIMPLE (1000 processors, 3 iterations, 20 × 20).

Figure 5 shows the speedup and utilization curves for SIMPLE.[4] The curves show the limits to improved performance imposed by data dependencies *in the algorithm itself*. For example, even on an ideal machine, SIMPLE (20 × 20) exhibits $speedup(100) = 97$ only ($utilization(100) = 97\%$).

1.3.3 Parallelism with Non-Zero Latencies

In a realistic machine, n processors and memories would be interconnected by a network with non-zero average latency l (typically $O(log(n))$). We can model this latency by assuming that the output of every instruction takes l time steps to reach its destination (thus, the parallelism profiles so far had $l = 0$). Figure 6 shows two profiles for 3 iterations of SIMPLE (20 × 20), assuming no more than 100 processors. One profile assumes $l = 0$, and the other assumes $l = 10$.

Despite the enormous increase in latency, the critical path lengthens by hardly a factor of 2 (from about 15,000 to about 30,000). This shows that SIMPLE has enough dependency-free parallelism to absorb much of the latency (up to 10) on 100 processors.

Figure 7 shows the *speedup* curves for SIMPLE (taking latency into account).[5] The points in the figure are from GITA runs for various settings of n and l, while the curves are derived analytically from the ideal parallelism profile. One way to interpret these curves is

[4]These are estimated from the ideal parallelism profile; we have shown that our estimates come within a small factor of actual runs.

[5]The data for this figure was generated using an earlier version of the compiler (with fewer optimizations).

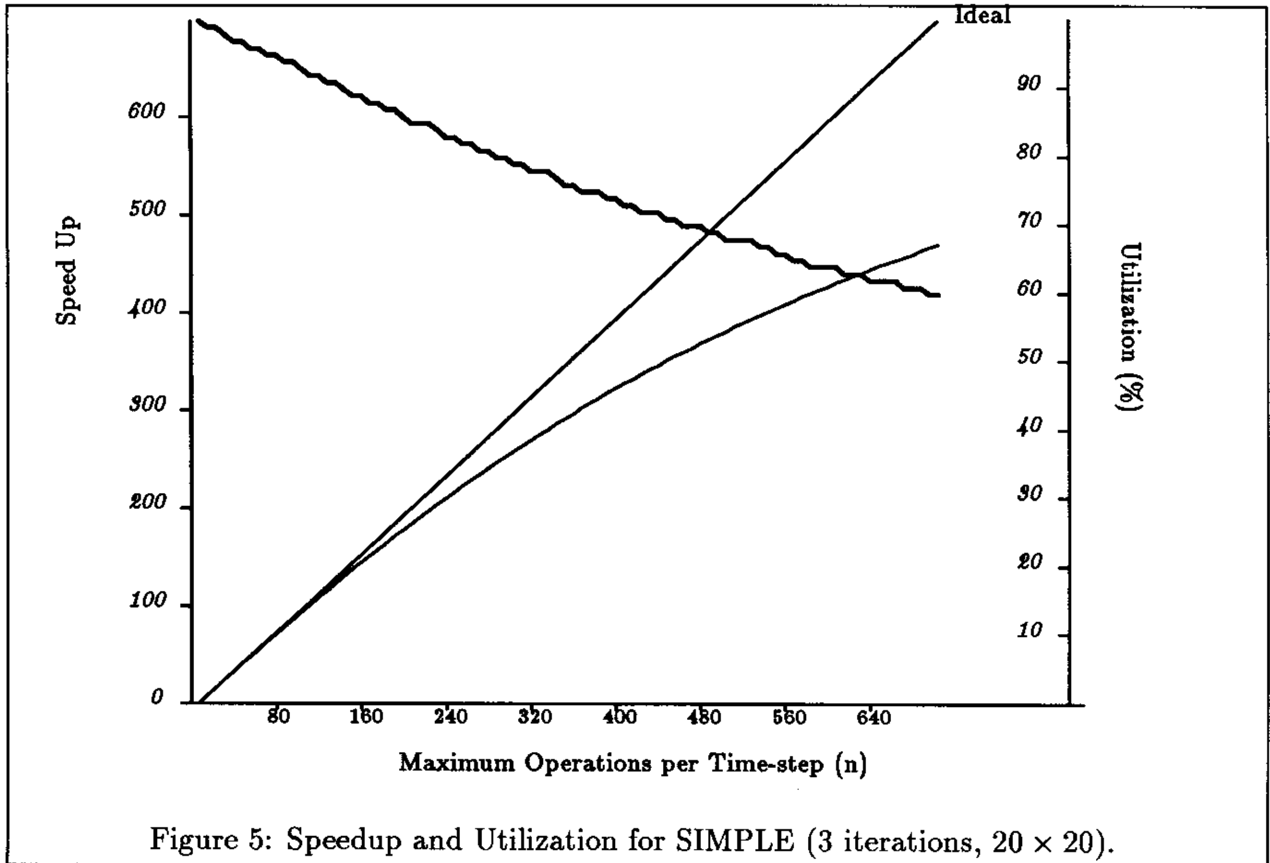

Figure 5: Speedup and Utilization for SIMPLE (3 iterations, 20 × 20).

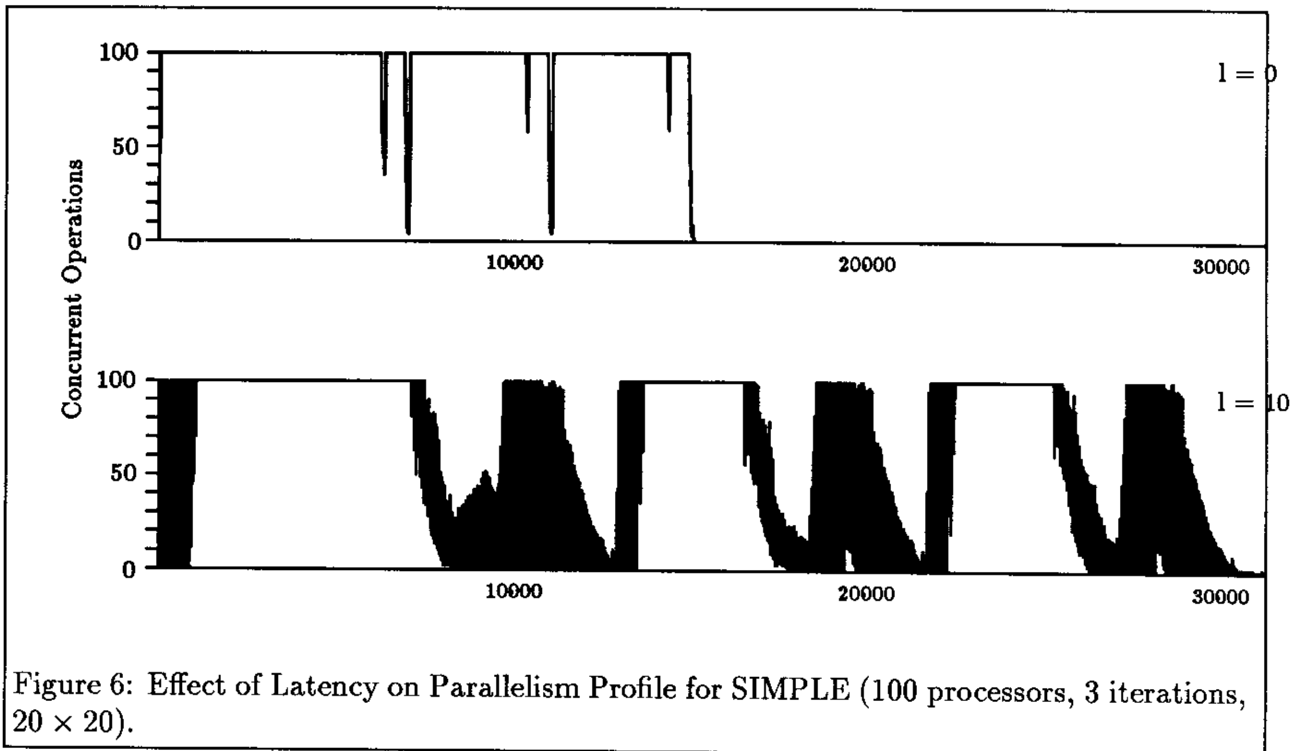

Figure 6: Effect of Latency on Parallelism Profile for SIMPLE (100 processors, 3 iterations, 20 × 20).

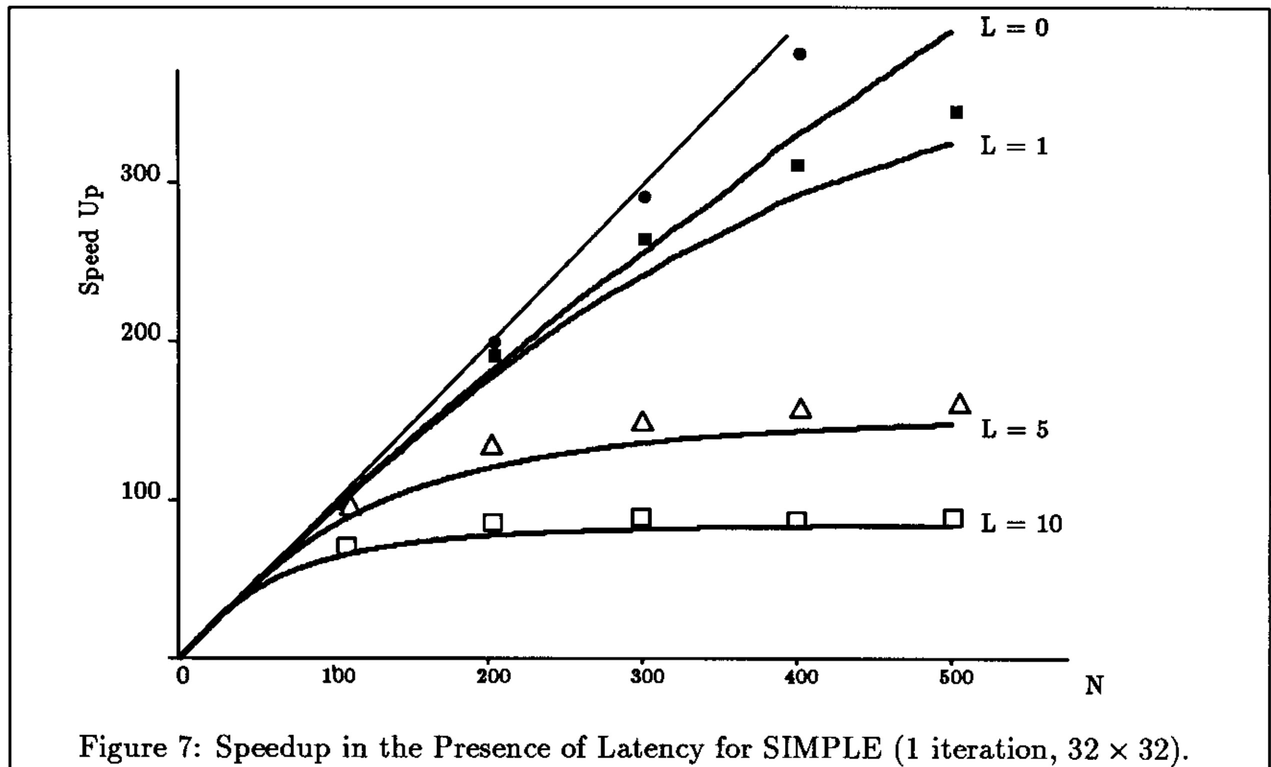

Figure 7: Speedup in the Presence of Latency for SIMPLE (1 iteration, 32 × 32).

that if we do not get a near-linear speedup on a machine with, say, 50 processors and latency 10, then the fault lies with the machine, not the program. On the other hand, the program does not have sufficient parallelism to utilize a machine with 500 processors effectively.

1.3.4 Larger Granularities May Have Inadequate Parallelism

The absence of speedup on an application can sometimes be traced to the lack of support for sufficiently fine-grained parallelism in a machine architecture.

Many proposed parallel architectures are based on von Neuman processors. Because of the high cost context-switching, researchers often advocate parallelism only at a coarser grain:

- Procedure-level: procedures may execute in parallel, but procedure bodies execute sequentially.
- Iteration-level: in addition, iterations of a loop may execute in parallel, but loop bodies execute sequentially.

Using a modified GITA, the parallelism profiles for SIMPLE (32 × 32, 1 iteration, infinite processors, zero-latency) are shown in Figure 8 for instruction, iteration and code-block parallelism.[6] Note the dramatic drop in available parallelism with grain size, both in the number of concurrent operations and in the critical paths. It is also possible to plot these with

[6] The data for this figure was generated using an earlier version of the compiler (with fewer optimizations).

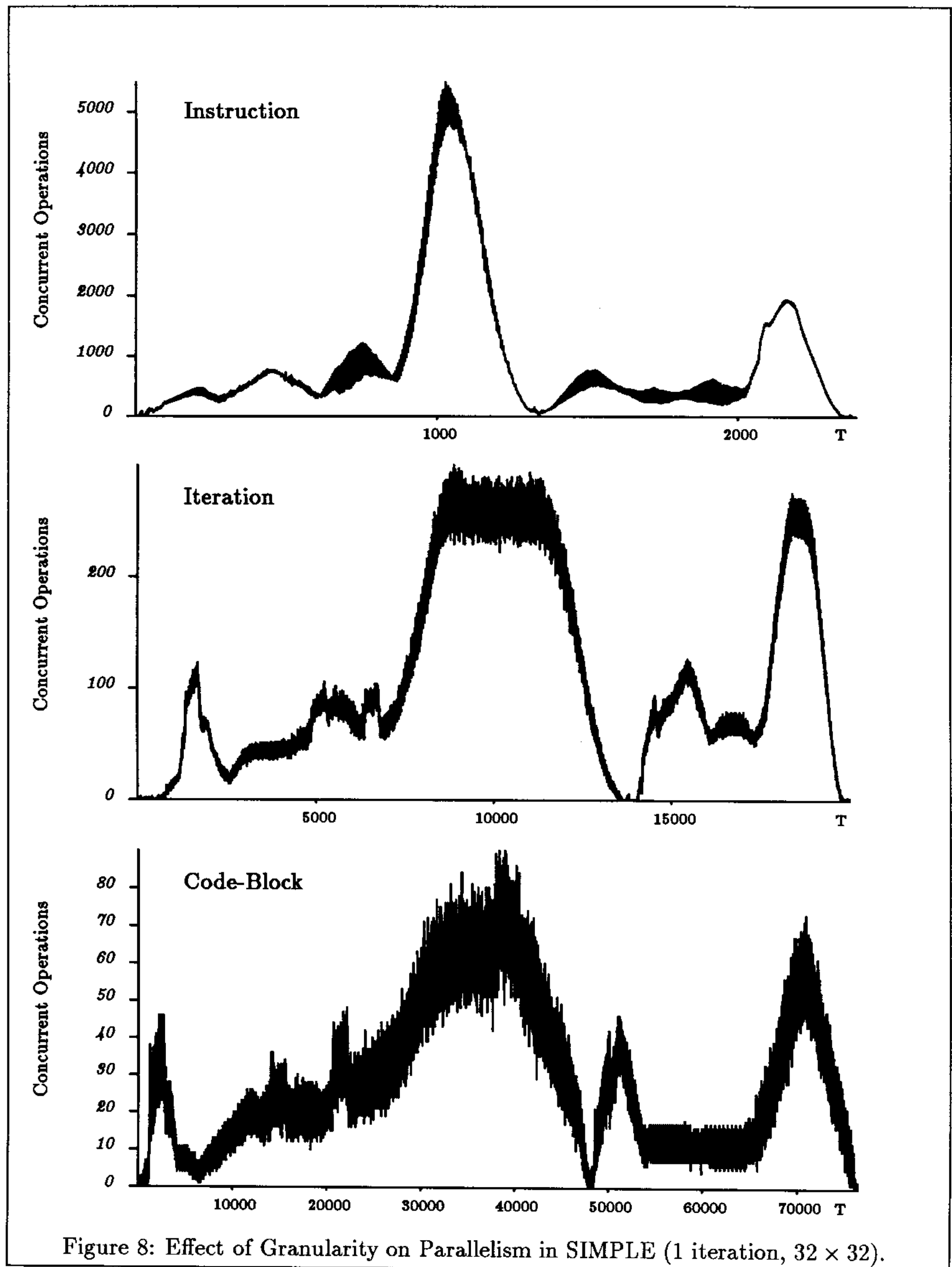

Figure 8: Effect of Granularity on Parallelism in SIMPLE (1 iteration, 32 × 32).

finite-processors and non-zero latency. The results show that instruction-level parallelism can keep many more processors busy.

1.4 The TTDA Can Exploit Parallelism in Programs

The MIT Tagged-Token Dataflow Architecture (TTDA) executes dataflow graphs directly. An overview of the TTDA may be found in [9]. Throughout this section we describe the TTDA as if it exists. However, it should be clear that only "soft implementations" exist, *i.e.,* emulators and the simulator.

1.4.1 Dataflow and von Neumann Instructions are Comparable

We compared SIMPLE written in FORTRAN and in Id. The FORTRAN version was the original one from Livermore. Dr. Ekanadham of IBM Research compiled it (for sequential execution) using the IBM 370 FORTRAN compiler, at the maximum optimization level (Level 3). Our Id version was compiled and executed on GITA. The measured dynamic instruction counts for 1 iteration of a 32×32 run (not counting I/O) are:

	FORTRAN/IBM 370		Id/TTDA	
Type	Count	%	Count	%
Floating point	349,646	26	354,848	25
Fixed point	179,888	13	61,452	4
Load,store,move	602,081	45	313,565	22
Identity			340,004	24
Branch/Switch	142,502	11	90,341	6
Subroutine linkage, *etc.*	70,084	5	132,325	9
Misc.			153,943	10
Total	1,344,201	100%	1,446,478	100%
Critical Path:			1,083	

The dataflow instruction count is within 7% of the FORTRAN version, but the following facts must be kept in mind:

- Both compilers use optimizations such as common subexpression elimination, inline substitution, constant folding, *etc.* The Id compiler does not yet do subscript analysis of array references.
- The dataflow code is already parallel. Parallelizing the FORTRAN code will increase the number of instructions due to task creation management. The precise significance of this factor is not yet known.
- Neither the Id nor the FORTRAN code counts instructions inside resource managers. Id's dynamic view of resources may cost it more.

Based on this experiment, on smaller experiments on a Cray, and on hand analysis of code, we are confident that dataflow instruction counts are comparable to von Neumann instruction counts.

1.4.2 TTDA Storage Requirements are Reasonable

What *is* Token Storage? On sequential machines, the storage for a procedure activation is a "frame". Loops cause no allocation. At each step, the storage in use equals the frames in the current call chain. Data structures may be in frames, or in a separate area called the heap. Most modern languages (including Common Lisp and Id) require heap allocation.

In parallel machines, concurrent invocations require a *tree* of frames in place of a stack. Since loop iterations can execute in parallel, even loops can require allocation of frames.

It is now clear to us that in the TTDA, token storage in the Wait-Match Unit corresponds to the traditional stack-based storage for frames, and I-structure storage corresponds to the traditional heap.

Storage Requirements Due to Uncontrolled Unfolding Consider a loop executing a 1000 iterations on 10 processors. With no constraints, we may soon execute instructions from every iteration, causing 1000 frames (and perhaps 1000 local arrays) to be allocated. But the frames and arrays for the later iterations are likely to be idle for a long time, and so the allocation is premature. Worse, the program may deadlock for lack of storage. Thus, estimates of storage requirements must be predicated on the strategy for unfolding.

Controlling Unfolding by Loop Throttling We have developed a compilation technique [14] that limits the unfolding of loops to no more than k iterations, where k for each loop can be specified as late as loop invocation time. Throttling can decrease token storage requirements dramatically *without* increasing the critical path. The upper curve in Figure 9 shows the Wait-Match store required for 3 iterations of a 20×20 SIMPLE run, assuming no throttling. The lower curve assumes $k = 1$ for the outermost loop. The storage requirements decrease dramatically, even though the critical path is almost unaffected.

1.4.3 The TTDA Tolerates Communication Latencies

We studied SIMPLE on our TTDA simulator[20]. Because of the level of simulation detail, the practical upper limit for the simulated system size is around 16 TTDA processors, and the biggest SIMPLE problem we can run is about one iteration on a 10×10 grid (just over 200,000 dataflow instructions).

The results are presented as speed-up curves in Figure 10. The curves confirm that the TTDA can indeed sustain an extraordinary amount of latency while still retaining much of its speed. In contrast, a single von Neumann processor would slow down ≈ 50 times given a similar arrangement. The reason is that in a dataflow processor, a memory-fetch does not cause the processor to idle at all— other instructions follow immediately in the pipeline, whereas a von Neumann processor must idle during an entire memory reference.

The same experiments, run on GITA modified as before for code-block level parallelism with round-robin scheduling, confirm these results.

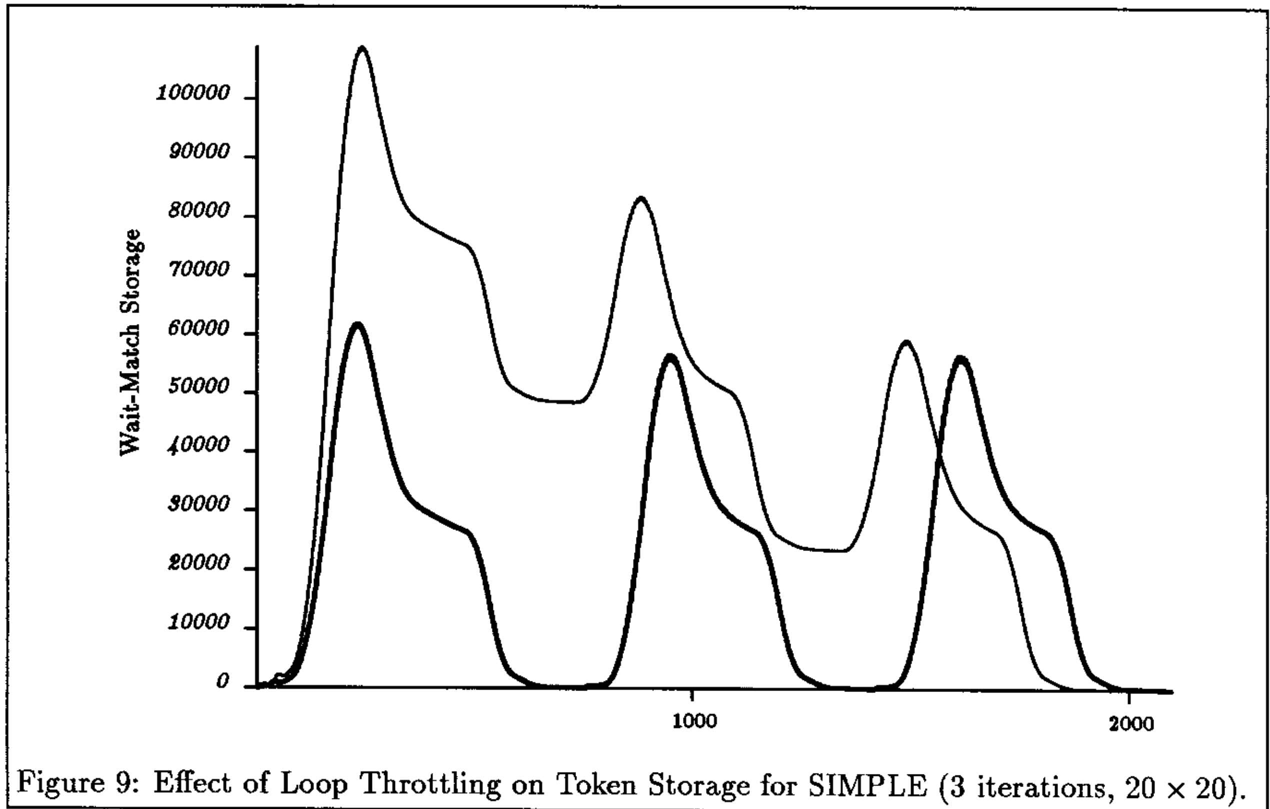

Figure 9: Effect of Loop Throttling on Token Storage for SIMPLE (3 iterations, 20 × 20).

1.4.4 Simple Code-Block Distribution is Robust

For each code block invocation, the system must decide where (on which processor) to allocate its frame, the objective being to balance load. We implemented several code-block allocation policies on the simulator, including static allocation, round-robin, hashed, load-leveling *etc.* Our results show that a simple round-robin strategy does almost as well as the more complex schemes.

1.4.5 Lessons from Experimental Results

1. *Instruction Set*: Dataflow MIPS are comparable to von Neumann MIPS.

2. *Directly-Addressed Wait-Match Storage*: Wait-Match store can be implemented as: **A)** An associative memory for tokens, keyed on token tags, or **B)** a collection of frames, one for each code block activation.

   In **A** the size of the memory required is the maximum number of concurrently active tokens. Our studies show that this is large, so the only feasible implementation is a hash memory. Because of collisions, accesses can take several cycles, thus degrading performance. This approach has been used in both the Manchester and the Sigma-1 dataflow machines [17, 19].

   The attraction of **B** is that the store is *directly addressed* using the tag. This suggests a very simple, fast implementation. The problem is that with uncontrolled program

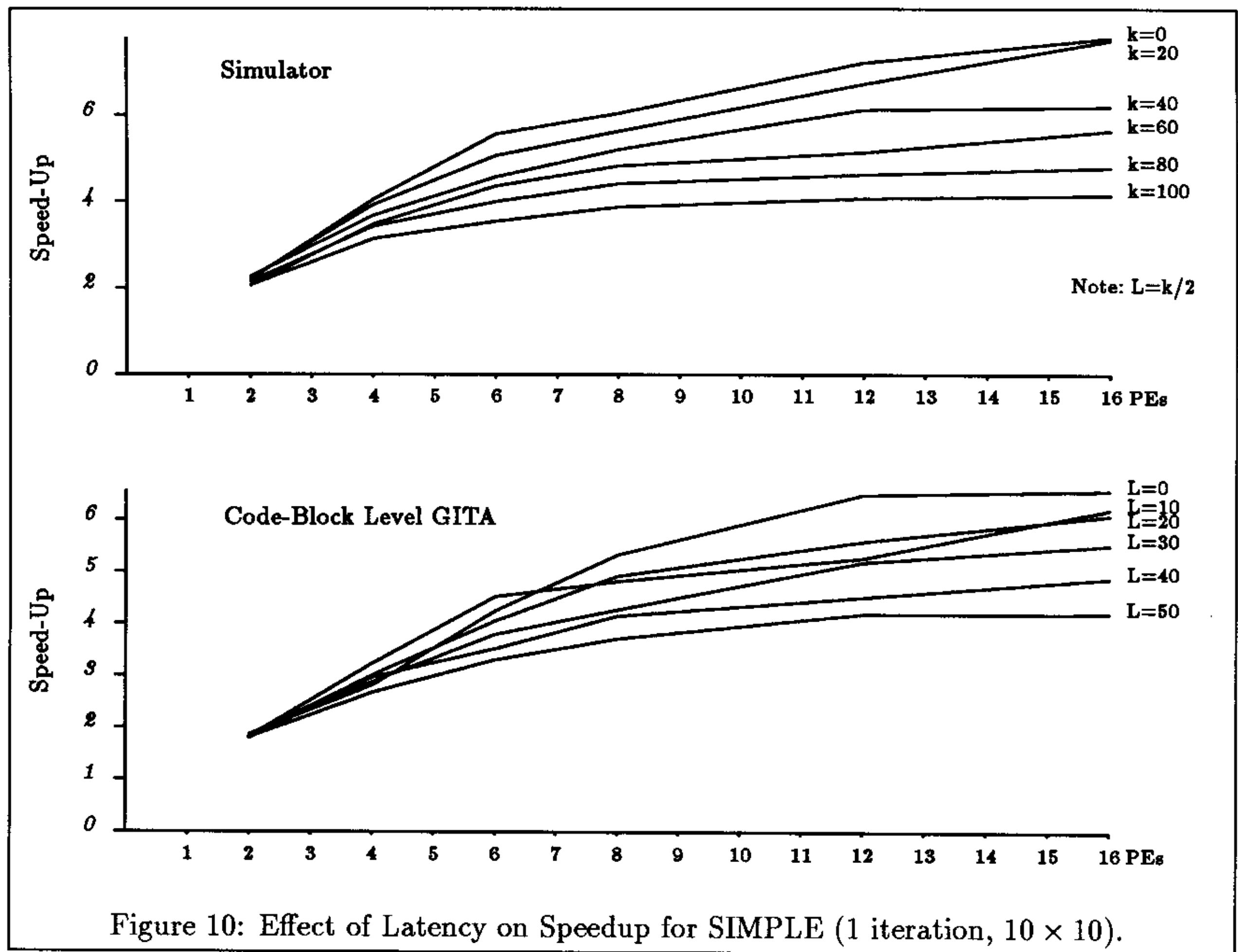

Figure 10: Effect of Latency on Speedup for SIMPLE (1 iteration, 10×10).

unfolding, few locations in each frame hold active tokens. Thus, much space is wasted due to fragmentation and the total memory required is too large.

We have shown that loop bounding makes **B** viable. It limits the number of active frames, and decreases fragmentation by concentrating activities in fewer frames. The Monsoon architecture adopts this approach.

3. *Unification of Wait-Match and I-structure Memory*: Wait-Match store and I-structure storage both implement *synchronization* in the TTDA. When Wait-Match store is implemented as a directly addressed memory, the similarity is greater, suggesting a unification of the two.

4. *Completeness*: An issue not addressed seriously is that of completeness— the ability to code and execute *all* support software, notably resource managers, on the TTDA itself. Resource management actions were performed by an unspecified "Control Section", which could even be a conventional von Neumann processor. This approach was deliberate: we wanted to understand the resource management requirements before implementing such managers.

   We have now begun addressing this issue. We are looking at clean extensions to Id for coding managers, including non-deterministic constructs. We are exploring architectural support for managers, such as instructions for atomic queue management and sequential entry into managers.

2 The Proposed Dataflow System

We are greatly encouraged by the results of our research— we are confident that Id is suitable as a high-level parallel programming language, that dataflow graphs constitute an ideal parallel machine language, that Id can be compiled easily into dataflow graphs, and that a multiprocessor that can execute dataflow graphs directly will be efficient and scalable. We can show that it is not enough to *interpret* dataflow graphs on, say, von Neumann processors— direct hardware support for data-driven scheduling is essential to realize the benefits of the dataflow model. Our deeper understanding has also led to new and significant ideas on how practically (and economically) to implement dataflow processing elements. We are now no longer hesitant to build an actual dataflow computer.

A major question remains— effective resource management. This is still poorly understood for *all* parallel architectures, not just dataflow. Unfortunately, simulations of parallel architectures are just not able to run large enough problems to shed light on the resource-management problems— real machines must be built to run such large programs.

We propose to build a machine that balances this need for adequate performance with conservative engineering discipline.

2.1 System Overview

A block diagram of the proposed Dataflow Computing System is shown in Figure 11. It will comprise 256 dataflow processing elements/structure controllers, a high performance two-stage packet switched interprocessor network, and an off-the-shelf Input/Output subsystem.

The dataflow Processing Elements (PEs) have a new architecture which we call "Monsoon". It will be capable of over 10 million instructions per second, any fraction of which may be 64-bit floating point operations. A Monsoon PE can also behave as an I-Structure Controller, yielding over 10 million structure operations per second. Each PE will have two million words of primary store. The network will provide 100 megabytes per second interprocessor bandwidth per PE.

The raw performance of the entire dataflow system will be in excess of 2000 million instructions and/or structure operations per second. The total primary memory will be 512 million words. The primary memory of all PEs are uniformly addressed using a single, global addressing mechanism. On a wide variety of scientific codes we expect a sustained performance of about 200 MFLOPS (about 8 times the average throughput of a Cray-1), making the proposed system competitive with contemporary von Neumann supercomputers.

2.2 The Monsoon Processing Element/Structure Controller

Monsoon is a new architecture for dataflow processing elements. The Monsoon PE may be regarded as a concrete hardware realization of the TTDA. Thus, it addresses the two fundamental issues of multiprocessing [7]— latency and synchronization— by (1)providing non-blocking, split-transaction, remote memory references, (2) providing efficient hardware synchronization on a per instruction basis, and (3) interleaving threads on a per instruction basis.

The major innovation in a Monsoon PE is the implementation of Wait-Match store as a *directly addressed* memory. Previous dataflow architectures (Manchester, ETL Sigma-1) used a large associative memory implemented as a hash table. Not only are large, fast hardware hash tables difficult to design, they are also slow because resolution of collisions can take several cycles. In contrast, Monsoon's Wait-Match operations involve simple RAM reads and writes.

A second innovation is that the identical Monsoon PE can serve as an I-structure controller. This unification leads to considerably simplified engineering of the hardware. It also simplifies resource management; since the same memory is now used for program code, token store and I-structures, the partitioning can be done dynamically.

Monsoon also supports the execution of imperative and non-deterministic code. This permits coding resource managers, and supports other programming languages.

The block diagram of a Monsoon PE is shown in Figure 12. The primary stores (DRAMs) of all the PEs in the system are collectively addressed in a uniform, global address space. Thus, a global memory address can be viewed as the concatenation of a PE number and an address within that PE. Unlike the TTDA, which had separate memories for program code, waiting tokens, and data structures, Monsoon's DRAM serves all purposes.

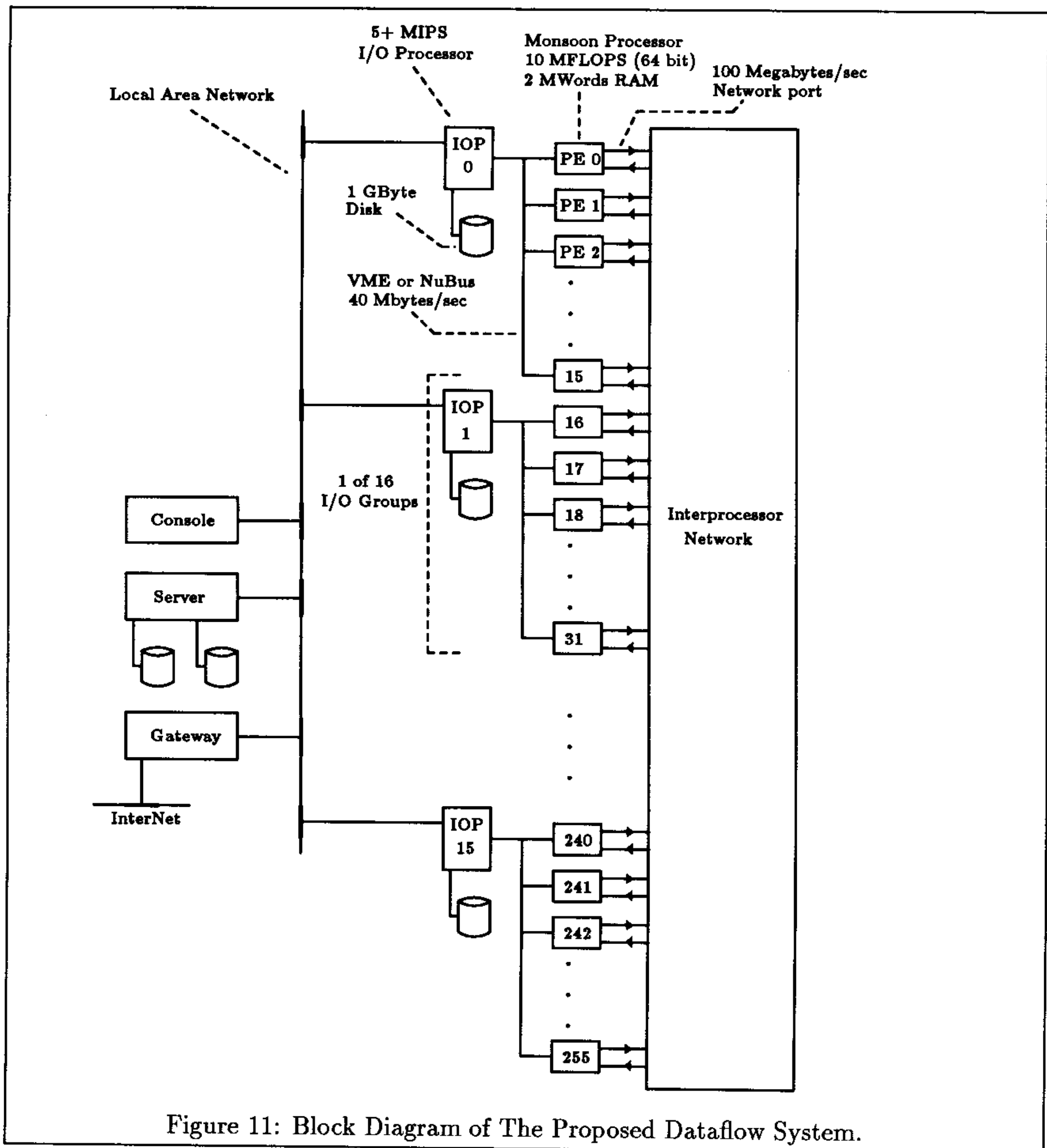

Figure 11: Block Diagram of The Proposed Dataflow System.

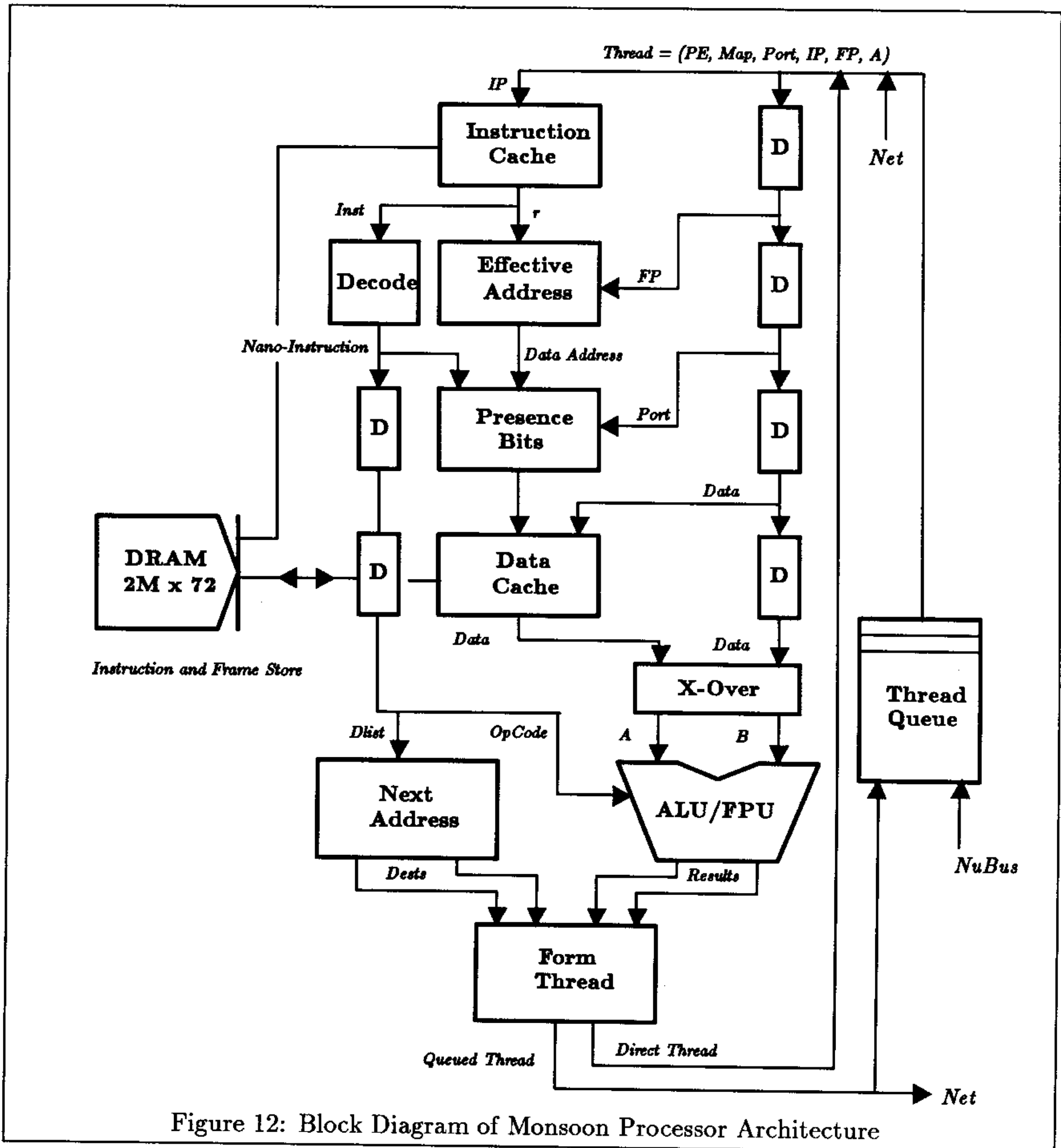

Figure 12: Block Diagram of Monsoon Processor Architecture

2.3 Executing Dataflow Graphs on Monsoon

The compiler plans the layout of a *frame* for each code block. Each time the code block is invoked, such a frame is allocated for that activation. A frame contains a *rendezous* location for every diadic operator in the code block. Thus, a frame pointer is the analog of the *context* identifier in TTDA tags.

A token has the form $(\langle PE, IP, FP\rangle, A)$ (see Figure 13). The tag of a token consists of a Processing Element number PE, an instruction pointer IP, and a frame pointer FP.[7] IP and FP are addresses in processor PE's memory. It is also possible to view a token as a *thread* descriptor, where IP plays the role of program counter, FP the role of stack frame pointer, and A the role of the classical "accumulator".

Each processor may typically support thousands of tokens (simultaneous threads) in the thread queue, which feeds the main processor pipe.

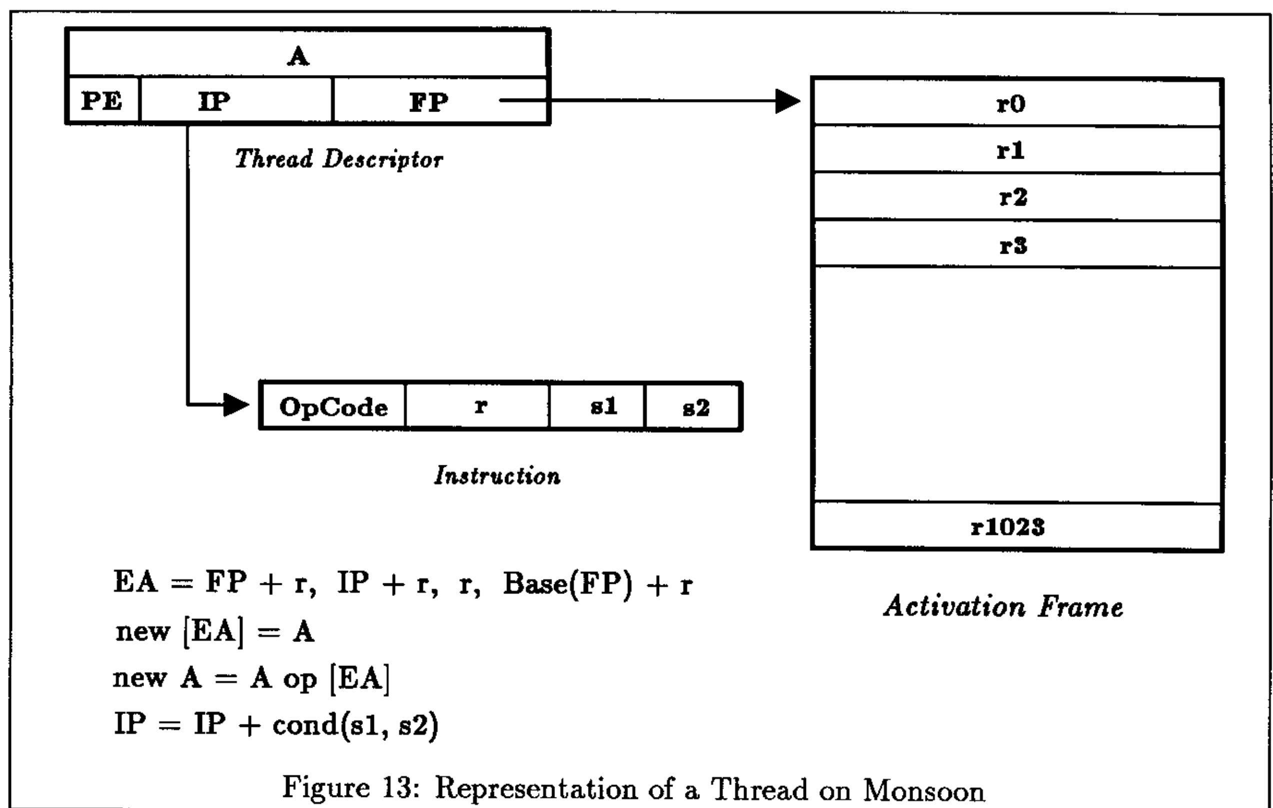

Figure 13: Representation of a Thread on Monsoon

The following actions take place as a token passes through the pipe:

1. *Instruction Fetch*: An instruction of the form $(Opcode, r, s)$ is fetched from location IP. *Opcode* specifies a "frame operation" and an ALU operation, r specifies an offset in the frame, and s specifies an increment to IP that locates the destination instruction (there may be two destinations s_1 and s_2).

[7] In more detail, it is actually $\langle Map, PE, IP, port, FP\rangle$. *Map* defines the subdomain size for structure interleaving, and *port* indicates the left or right port to a diadic instruction.

2. *Effective Address Computation*: A local memory effective address EA is computed depending on the addressing mode (part of the instruction). Examples: $FP+r$ (frame relative), $IP+r$ (instruction relative), r (absolute), or $FP_{base}+r$ (loop constant).

3. *Frame Operation*: The normal frame operation is a "wait-match" operation (see Figure 14). Every memory word has extra bits called "presence bits". If data in location EA is absent, A is stored there. If some data B is present, it is extracted. The wait-match operation can also be viewed as the *join* or rendezvous of two threads.

   Other frame operations include simple reads, writes (with A), exchanges (read B, write A), *etc.*

4. *ALU operation*: A and B are sent to the ALU, which applies the specified ALU operation, yielding a new A'. The ALU has parallel function units for floating point, bit-manipulation, and pointer arithmetic, permitting generic (type independent) opcodes.

5. *Next Address Computation*: A new destination (PE', IP') is computed using (PE, IP) and s. Finally, an output token emerges with $(\langle PE', IP', FP\rangle, A')$. The old frame pointer is retained. If there are two destinations s_1 and s_2, two output tokens are produced. This corresponds to a *fork* yielding two threads within the same frame.

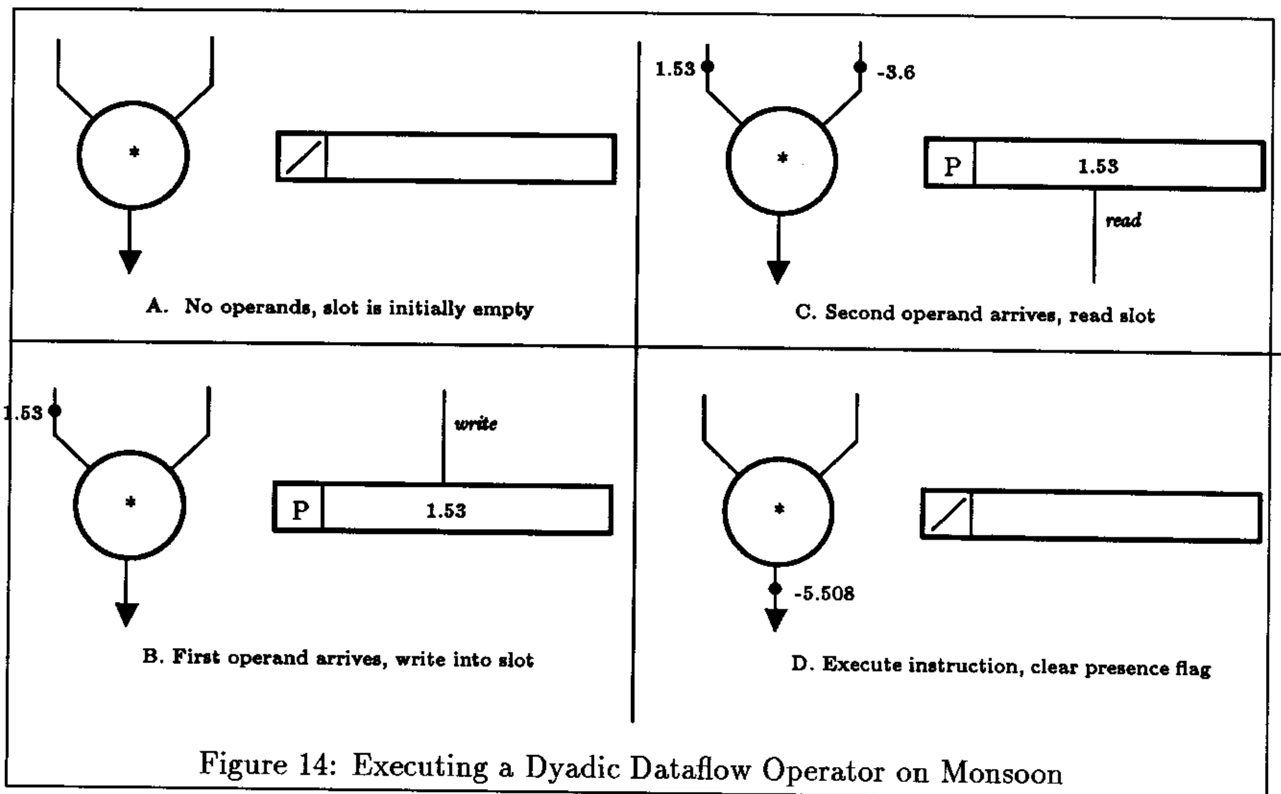

Figure 14: Executing a Dyadic Dataflow Operator on Monsoon

A very important optimization in the TTDA is the *loop constant*, which permits sharing of values across iterations without repeatedly copying (circulating) them. A special addressing

mode (loop constant) in Monsoon provides most of the efficiency of TTDA loops without introducing a special i field on tags.

The tag $\langle PE, IP, FP\rangle$ and the data A on a token are of the same size. Thus, tags can be carried and manipulated as data. This is used to implement resource managers that allocate/deallocate tags, and by the diadic SEND instruction which takes a tag (as data) and some data, and produces a token with that tag and data. This is the TTDA CHANGE-TAG instruction, and, as we shall see, subsumes the TTDA's I-FETCH and I-STORE instructions as well.

2.4 I-Structure Operations on the Monsoon Processor

The Monsoon processor can also behave as an I-structure controller.

To write value A into location l of processor PE, we construct the following token using the SEND instruction: $(\langle PE, IP_{write}, l\rangle, A)$. The token arrives at processor PE, where it is processed by the "I-structure Write" instruction at reserved location IP_{write}, whose effect is simply to write A into location l.

To read a location l from PE_1 such that the result A should go to instruction IP in PE_2 with context (frame) FP, we construct the following token using the SEND instruction: $(\langle PE_1, IP_{read}, l\rangle, \langle PE_2, IP, FP\rangle)$. The token arrives at PE_1, where it is processed by the "I-Structure Read" instruction at reserved location IP_{read}, whose effect is simply to read the value A there, and construct the token: $(\langle PE_2, IP, FP\rangle, A)$

Note that memory reads are split into two transactions— one that sends a read request, and another that replies with the value. In the interim, the pipeline is not blocked— other instructions are free to execute.

Implementing a single deferred read for I-structures is easy. When a read request arrives at an empty I-structure location, it simply stores its destination there. When the write arrives, it reads out the waiting destination, writes the value, and sends the value to the waiting destination. Implementing multiple deferred reads is a little trickier, but possible.

In general, tags $\langle PE, IP, FP\rangle)$ can be viewed as pointers. Unlike conventional machines however, every pointer carries with it an instruction address that specifies what to do with the pointer.

One significant departure from the TTDA is that atomic values are of a single fixed size. This means that compound objects that used to be carried on tokens, like array descriptors and closures, must instead be references to regions in memory where the information is maintained. For example, an I-structure descriptor will simply become a pointer to an array header that contains its bounds and type information.

2.5 Support for Other Programming Languages

Monsoon is more general than a TTDA graph interpreter. In fact, one can view it as a processor analogous to the Denelcor HEP, supporting multiple concurrent threads efficiently.

Each traversal through the pipe is potentially a join of two threads, an operation, and a fork into two threads. Threads are interleaved on every cycle from a hardware-managed task queue, without any context-switching overhead.

Thus, Monsoon can also support existing languages like FORTRAN, C and Lisp. Of course, Monsoon is unlikely to compete with a von Neumann processor on a single thread— whereas the latter may be able to pipeline a sequential thread, successive instructions in a thread must pay the full latency of the pipe in Monsoon. However, if we generalize the problem to deal with multiple threads, Monsoon's efficient interleaving of threads begins to pay off. We conjecture that on multi-threaded applications, even a *single* Monsoon processor will be competitive with a single von Neumann machine.

One language that we are examining closely for the Monsoon processor is Multilisp [18], which subsumes conventional languages like FORTRAN, C and Lisp. This has also influenced the architecture, leading to some changes in the repertoire of Monsoon's Frame Operations for efficient support of Multilisp's `FUTURE` construct.

2.6 Technology Considerations for the Monsoon Processing Element

Major considerations in designing the Monsoon Processing Element are:

- *Technological realizability.* Every component should have a feasible implementation using current technology. For example, it is not feasible to implement a very large, fast associative memory.
- *Technological scalability.* Every component should be able to benefit automatically from normal improvements in technology. For example, improved circuit speeds should result in higher performance— there should be no critical "hot component" that limits speed.

We believe that the processor pipeline is well-balanced and thus technologically scalable: an individual processor can be sped up by employing a faster implementation technology.

We can obtain competitive performance with simple technologies. We expect an initial implementation of PEs in field-programmable CMOS gate arrays and TTL, and later in ASICs with a cycle time of 100 nanoseconds or less. The datapaths will be 64-bit values with 8 bits of type and GC information. Double precision (64 bit) IEEE arithmetic is employed yielding over 10 Megaflops/10 MIPS peak rate per processor. Each processor will have 2 MWords (18 Mbytes) of local storage implemented with 1Mbit DRAMS (error correcting) with a 128 KWord static RAM cache. We will use a single double-height (9U) VME card for the discrete PE. Later, four ASIC PEs will fit on the same card. We envision four VME cards per VME host (Sun workstation).

2.7 Interprocessor Communication

The memories on each PE are collectively organized into a global address space. Experimental results indicate the need to spread out loops and data structures across the machine,

rather than trying to exploit locality within a processor. Hardware mapping functions permit the *interleaving* of structures on a word-by-word basis across processors. This randomizes structure references, and has the positive effect of statistically averaging processor and network loading on a fine-grain basis. The disadvantage is that it can cause extra network traffic.

The network load is relatively easy to estimate. Structure references account for about 30 percent of the instructions executed in the SIMPLE code. If all references require a network transaction then each processor will, on average, send and receive a message once every three instructions. For a 10 MIPS processor this is three million messages per second. All messages are the same size, 18 bytes, yielding an average message rate of about 60 megabytes per second per processor.

To meet these requirements, interprocessor communication is accomplished through a network with a per-port bandwidth of 100 megabytes per second, organized as a two-stage exchange network with 16×16 routers. The network is not multipath, but it is pipelined and packet-switched. A packet switch of this size can deliver about twice the useful bandwidth as a circuit switch with an equal raw per-link bandwidth.

We believe that the network is readily realizable with compositions of CMOS 4×4 routers (about 30,000 gates each) and nibble-wide ECL links over low-loss coaxial cable.

2.8 Input/Output System Architecture

Dataflow machines offer a truly unique possibility for well integrated I/O systems, by attaching I/O devices as nodes in the interprocessor communication network. For example, to read a disk block, we send it a read-request token containing a pointer to an I-structure. The device simply sends the data as tokens to be written into that I-structure. A consumer can begin reading contents of the I-structure even as it is being filled. The device could send a "completion token" to a waiting manager (analogous to a I/O completion interrupt). This overlapped processing, synchronization, *etc.* fits naturally into the dataflow model.

However, the engineering effort required for even a simple disk controller is non-trivial. We have tried to minimize our development effort for Monsoon I/O in two ways. First, a large main store (512 Megawords) mitigates the need for high-speed temporary or swapping store. Second, we are employing, wherever possible, off-the-shelf I/O components, including commercially available processors, device controllers and LANs.

An *I/O Group* is a collection of four Monsoon PE cards and an off-the-shelf host processor for I/O (workstation). In the discrete implementation, a processor occupies an entire card, so there are four PEs per I/O host. In the ASIC version, there will be four PEs per card, or sixteen per host. The bus is used for power, I/O transfers between devices and Monsoon PEs, and for maintenance and monitoring. The memory in each Monsoon PE will be dual-ported, with one port used for DMA transfers to and from the bus, initiated by special I/O instructions. We will rely on the file system and LAN interface of the host processor.

Each I/O processor will have local disk capacity of at least one gigabyte, and all I/O processors will be networked under a single file system. Ideally, disks should be of supercomputer-class performance: 2-3 millisecond seek times and transfer rates in the range of 10

megabytes per second. Alternatively, high performance and less expensive mainframe drives could be employed. These have competitive seek times but support only about one-quarter of the transfer bandwidth.

The I/O processor should be robust, have rapid I/O responsiveness, an open and well-understood operating system interface, an integrated network file system, and host an industry standard, high-performance multi-master bus. The candidates under consideration include: the TI Explorer II Lisp Machine, the Sun 4, Apollo DN4000, and MIPS. At this time a Sun 4 running UNIX (or, preferably, a robust, real-time derivative) appears to meet most of these goals.

2.9 Software Plans

2.9.1 Id as a Layered Language

The Id language is unique in its structured approach to parallelism.

The core of Id is a purely functional programming language with higher-order procedures and non-strict operations, the benefits of which include:

- A level of programming much higher than current programming languages such as Common Lisp.
- Abundant implicit parallelism, freeing the programmer from the tedious details of decomposing an algorithm into concurrent parts.
- Guaranteed determinacy, freeing the programmer from the difficult issues of synchronization, and making debugging straightforward.

But functional languages do not support data structures such as arrays efficiently, nor are they appropriate for expressing concurrent programs with shared resources (such as Input/Output). Thus, we need to go beyond functional languages.

Conventional wisdom equates "non-functional languages" with "languages with side-effects". Side-effects in a parallel language immediately destroy determinacy and therefore complicate the language and the compiler significantly.

In contrast, Id is a *layered* language, with many stages between the purely functional core and full side-effects, so that the programmer does not have to concede indeterminacy immediately if he has to go outside the functional core:

(1) Functional Language.
(2) Layer (1) + I-structures.
(3) Layer (2) + other determinate constructs, such as "accumulators".
(4) Layer (3) + non-deterministic constructs, such as "managers" (which includes some forms of Input/Output).
(5) Layer (4) + arbitrary side-effects (including other forms of Input/Output).

We already have much experience with I-structures (layer 2), which solve most of the efficiency problems of data structures in functional languages. We have implemented and are experimenting with a construct called "accumulators" in layer 3, which are a further generalization of parallel, determinate data structures. We have a proposed design for resource

managers in layer 4, which introduce non-determinacy without serious synchronization issues.

Each layer adds some "power" to Id, but at the expense of some desirable property such as referential transparency, determinacy, *etc.* One should always use the least power that will suffice for an application. This requires a programming methodology that allows constructs from level j to be packaged into abstractions at level $j-1$. We have already developed such a methodology for layer 2. We need to extend this to higher layers.

2.9.2 Types and Type-checking in Id

Current programming languages have a wide variety of data-type disciplines, ranging from statically type-checked, non-polymorphic Pascal to dynamically type-checked (and therefore polymorphic) Lisp. A rich type system, along with static type-checking can result in significantly better (and early) error-checking, and significantly more efficient code. However, a lack of polymorphism (generics) and/or a requirement for detailed type declarations can seriously hinder ease of programming. A type system due to Milner [21] strikes an excellent balance— it permits the convenience of Lisp in its polymorphism and lack of type-declarations, while checking types statically, thus enabling the compiler to catch errors early and to generate very efficient code.

We have already incorporated a first version of such a Milner-style type system in Id. Our future plans include:

- Extensions for incremental type-checking,
- Extensions for a limited form of sub-typing,
- A methodology to turn off type-checking selectively when the expressive power of the typed language is inadequate, and
- Use of type information for generating better code.

2.9.3 Id Compiler

The back end of our Id compiler currently generates dataflow graphs in terms of TTDA machine instructions. Conversion to the Monsoon instruction set is very easy because of the close relationship between the two machines. We have already begun this work.

The Id compiler already incorporates several major optimizations. Almost all these optimizations precede the machine-level dataflow graphs, and thus carry over directly to the Monsoon version. The work on optimization will continue to be a very important component of our software effort.

The compiler will continue to evolve in step with the language extensions.

2.9.4 Resource Management

Several issues in resource management (described below) are likely to dominate our software efforts. Common to all of them is an extension to Id to express non-deterministic managers.

We have already designed such an experimental extension and are currently incorporating it in the Id compiler.

Controlling Program Unfolding. We have seen that loop bounding has a dramatic effect on storage requirements. We need to extend these techniques to control the general recursive unfolding of programs. The problem is to achieve a balance— rapid unfolding can swamp the machine; slow unfolding may underutilize the machine. Further, the rate of unfolding can affect the deadlock behavior of a program.

Storage Management. Id has dynamic storage allocation both for expressiveness[8] as well as for parallelism. But this requires automatic storage reclamation, which is a difficult problem in parallel machines. Our general approach is to use *reference counting.* We are investigating several optimizations based on type-checking and *abstract interpretation* to minimize the overhead of reference counting operations. Abstract interpretation can also sometimes predict that the lifetime of a structure coincides with the context in which it was allocated, in which case we can compile code so that it is frame allocated instead of heap allocated. Deallocation is then automatic along with the frame. Frame allocation, when combined with loop bounding, can further reduce this overhead by pre-allocating all the arrays for k iterations, reusing them during the loop, and deallocating them all at once at the end.

Load Balancing. Our studies so far have shown that simple round-robin allocation strategies, together with interleaving, are quite effective in balancing load. We will study this problem in more detail when we can run large programs on the proposed system.

Input/Output. We will use the host processors' low-level I/O drivers as far as possible. Our activities will, instead, concentrate on the Id interface for I/O. We have already begun experimenting with parallel I/O constructs for Id, together with their compilation strategies.

[8]No modern programming language, even a sequential one, can afford not to have dynamic allocation.

Contents

1 MIT Tagged-Token Dataflow Project Results 1
1.1 Research Tools . . . 2
1.2 Id is a Good Language for Parallel Programming . . . 2
1.2.1 Id is Expressive . . . 3
1.2.2 Id is Compilable . . . 3
1.2.3 Id Does Not Obscure Parallelism . . . 3
1.2.4 Id Also Exploits Parallelism Adaptively . . . 4
1.3 There Is Sufficient Parallelism in Existing Programs . . . 5
1.3.1 Inherent Parallelism in SIMPLE . . . 5
1.3.2 Parallelism on a Finite Number of Processors . . . 7
1.3.3 Parallelism with Non-Zero Latencies . . . 8
1.3.4 Larger Granularities May Have Inadequate Parallelism . . . 10
1.4 The TTDA Can Exploit Parallelism in Programs . . . 12
1.4.1 Dataflow and von Neumann Instructions are Comparable . . . 12
1.4.2 TTDA Storage Requirements are Reasonable . . . 13
1.4.3 The TTDA Tolerates Communication Latencies . . . 13
1.4.4 Simple Code-Block Distribution is Robust . . . 14
1.4.5 Lessons from Experimental Results . . . 14

2 The Proposed Dataflow System 16
2.1 System Overview . . . 17
2.2 The Monsoon Processing Element/Structure Controller . . . 17
2.3 Executing Dataflow Graphs on Monsoon . . . 20
2.4 I-Structure Operations on the Monsoon Processor . . . 22
2.5 Support for Other Programming Languages . . . 22
2.6 Technology Considerations for the Monsoon Processing Element . . . 23
2.7 Interprocessor Communication . . . 23
2.8 Input/Output System Architecture . . . 24
2.9 Software Plans . . . 25
2.9.1 Id as a Layered Language . . . 25
2.9.2 Types and Type-checking in Id . . . 26
2.9.3 Id Compiler . . . 26
2.9.4 Resource Management . . . 26

Section H: Products and Transferable Technology

There are two kinds of prospective users: applications programmers who would like to use our machine, and manufacturers who would like to produce a machine based on our prototype.

Applications Programmers

Applications programmers may already begin experimenting with Id and dataflow by acquiring the existing Id World software which is currently available on Lisp machines and is soon to be available on Suns and other workstations such as MicroVaxes. The system is available at no cost under license from MIT, and has already been installed at several sites in the U.S. The system includes an editor, the Id compiler, an Id debugger, and a dataflow machine emulator on which one can run Id programs and study their parallel behavior.

We expect that users will be able to learn Id and begin to use it on the proposed system with minimum training (we have been quite successful in training even FORTRAN programmers to use Id World in just a few days).

Subsequently, users can acquire a succession of progressively more powerful dataflow machines that permit larger applications and/or increased performance. Simultaneously, users can expect software upgrades that will keep pace with the hardware. These machines are:

- From 2/89: the 6 MIPS single-PE dataflow accelerator board for Sun workstations.
- From 10/89: a 48-64 MIPS four-PE dataflow accelerator board for Suns, including basic I/O capabilities.
- From 2/90: a four-board, sixteen-PE dataflow subsystem for Suns, including small-scale resource-managers.
- From 2/91: a complete, 256-PE dataflow system based on Sun chassis, including large-scale resource managers (hardware built by Industrial Partner).

We expect the final system to have a peak performance of over 2000 MIPS, and to be capable of sustaining 1000 MIPS over a wide range of applications, making it competitive with the fastest contemporary general-purpose von Neumann machines. Any proportion of the instructions may be floating-point operations.

Manufacture

The single-board accelerators and sixteen-PE system will be designed by MIT with detailed design documents, and the construction will be subcontracted to industry. Thus, replication by interested manufacturers will be straightforward.

The 256-PE machine will be designed and constructed by an Industrial Partner, thereby resulting in straightforward replication.

References

[1] Arvind, S. A. Brobst, and G. K. Maa. Evaluation of the MIT Tagged-Token Dataflow Project. In *Submitted to the Fifteenth Annual International Symposium on Computer Architecture, Honolulu, Hawaii*, May 30-June 2 1987.

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

[3] Arvind and D. E. Culler. *Dataflow Architectures*, pages 225–253. Volume 1, Annual Reviews Inc., Palo Alto, CA, 1986.

[4] Arvind and D. 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.

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

[6] Arvind and K. Ekanadham. Future Scientific Programming on Parallel Machines. In *Proceedings of the International Conference on Supercomputing (ICS), Athens, Greece*, June 8-12 1987.

[7] Arvind and R. A. Iannucci. Two Fundamental Issues in Multiprocessing. In *Proceedings of DFVLR - Conference 1987 on Parallel Processing in Science and Engineering, Bonn-Bad Godesberg, W. Germany*, June 25-29 1987.

[8] Arvind, G. K. Maa, and D. E. Culler. Parallelism in Dataflow Programs. In *Submitted to the Fifteenth Annual International Symposium on Computer Architecture, Honolulu, Hawaii*, May 30-June 2 1987.

[9] Arvind and R. 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.

[10] Arvind, R. S. Nikhil, and K. K. Pingali. *Id Nouveau Reference Manual, Part II: Semantics.* Technical Report, Computation Structures Group, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, April 1987.

[11] Arvind, R. S. Nikhil, and K. K. Pingali. I-Structures: Data Structures for Parallel Computing. In *Proceedings of the Workshop on Graph Reduction, Santa Fe, New Mexico, USA, (Springer-Verlag LNCS 279).*, September/October 1986. (also Computation Structures Group Memo 269, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139).

[12] S. Brobst. *Instruction Scheduling and Token Storage Requirements in a Dataflow Supercomputer.* Technical Report, MIT Laboratory for Computer Science, 545 Technology

Square, Cambridge, MA 02139, May 1986. (Master's Thesis, Dept. of Electrical Engineering and Computer Science, MIT).

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

[14] D. E. Culler. *Effective Dataflow Execution of Scientific Programs.* PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, MA 02139, June 1988 (expected).

[15] D. E. Culler and Arvind. Resource Requirements of Dataflow Programs. In *Submitted to the Fifteenth Annual International Symposium on Computer Architecture, Honolulu, Hawaii,* May 30-June 2 1987.

[16] K. Ekanadham, Arvind, and D. E. Culler. The Price of Parallelism. In *Submitted to the Fifteenth Annual International Symposium on Computer Architecture, Honolulu, Hawaii,* May 30-June 2 1987.

[17] J. R. Gurd, C. Kirkham, and I. Watson. The Manchester Prototype Dataflow Computer. *Communications of the ACM,* 28(1):34–52, January 1985.

[18] R. H. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems,* 7(4):501–539, October 1985.

[19] K. Hiraki, S. Sekiguchi, and T. Shimada. *System Architecture of a Dataflow Supercomputer.* Technical Report, Computer Systems Division, Electrotechnical Laboratory, 1-1-4 Umezono, Sakura-mura, Niihari-gun, Ibaraki, 305, Japan, 1987.

[20] G. Maa. *Code-Mapping Policies for the TTDA.* Technical Report, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, December 1987 (expected). (Master's Thesis, Dept. of Electrical Engineering and Computer Science, MIT).

[21] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences,* 17:348–375, 1978.

[22] R. S. Nikhil. *Id Nouveau Reference Manual, Part I: Syntax.* Technical Report, Computation Structures Group, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, April 1987.

[23] R. S. Nikhil. *Id World Reference Manual.* Technical Report, Computation Structures Group, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, April 1987.

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