MIT/LCS/TR-365

ID WORLD: AN ENVIRONMENT FOR THE DEVELOPMENT OF DATAFLOW PROGRAMS WRITTEN IN ID

Dinarte R. Morais

May 1986

This blank page was inserted to preserve pagination.

ID WORLD:
An Environment for the Development of Dataflow Programs Written in ID

by

Dinarte R. Morais

Submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

May 1986

Signature Of Author Dinarte Morais

Department of Electrical Engineering and Computer Science
18 May 1986

Certified by

Arvind
Thesis Supervisor

Accepted By

David Adler
Chairman, Department Committee

ID WORLD:
An Environment for the Development of Dataflow Programs Written in ID

by Dinarte R. Morais

Arvind, Associate Professor of Computer Science and Engineering
Thesis Supervisor

Abstract

The ID WORLD project involves the interfacing of a compiler, interpreter, debugger and editor mode to create an environment for the development of dataflow programs written in ID. It replaces the Tagged-Token Dataflow Architecture (TTDA) Emulator as the foundation for the Multiprocessor Emulation Facility at the Laboratory for Computer Science, M.I.T.

This thesis presents the design of ID WORLD, noting the need for such a system, and includes a detailed examination of each subsystem. Special attention is paid to the problems inherent in the old TTDA Emulator, and how they were solved in ID WORLD.

Acknowledgments

I would like to thank Professor Arvind for the support and guidance he has given me while writing this thesis. I would also like to thank all the members of the Computation Structures Group, especially Ken Traub, Richard Mark Soley, David Culler, Greg Papadopoulos, Steve Heller, Bhaskar Guharoy, and Andrew Chien, for their invaluable assistance in the design and implementation of ID WORLD, and tremendous support they have given me over the past two years as an undergraduate researcher.

Table of Contents

Table of Figures

Chapter 1

Introduction

The Computation Structures Group (CSG) at the Laboratory for Computer Science (LCS) is currently building a Multiprocessor Emulation Facility (MEF [2]) to facilitate research and development in parallel architectures and languages. The first use of the MEF involves the emulation of the Tagged-Token Dataflow Architecture (TTDA [1]) to demonstrate the feasibility of general purpose dataflow machines.

To support these efforts, several programs have been written over the past several years to support dataflow software development. Unfortunately, these programs were not designed to be directly interfaced to one another. As a result, the software development cycle for dataflow programs was very long. A typical cycle would begin with the use of an editor to edit a file containing a piece of code written in a dataflow language. The file would then be written to disk, and a compiler would be invoked and told to read the source file, outputting a file containing the compiled code. Finally, an interpreter would be told to load the compiled file just produced, and only then would the original dataflow program be executed. When the interpreter would detect an error in the program, the user would have no choice but to go back to editing the source file and start the whole cycle over again.

To someone who develops programs in, say, C or CLU, this may seem like a normal cycle. A Lisp Machine programmer, however, has become accustomed to a very interactive environment where programs may be incrementally edited, compiled, executed, and debugged, without ever leaving the editor.

The goal of ID WORLD is to provide such an interactive environment for the development of programs written in ID [4], the dataflow language used by the Computation Structures Group. With ID WORLD the typical software development cycle is as follows. A user enters the editor and types in an ID procedure. With a single keystroke, the

procedure is sent to the ID compiler, and the output of the compiler is automatically loaded into the interpreter. Then, without leaving the editor, the user instructs the interpreter to execute the procedure, and when the interpreter detects an error the user simply continues editing the procedure until it is correct. Thus, with only a few keystrokes and without writing a single file, ID WORLD interfaces the editor, compiler, and interpreter, making it much quicker to develop programs.

The remainder of this thesis is organized as follows. Chapter two presents the design of GITA, the interpreter used in ID WORLD to execute dataflow programs. Chapter three presents the GITA debugger, noting how it differs from normal debuggers for sequential languages. In chapter four I describe ID Mode, a new editor mode for the development of ID programs. Chapter five discusses how each of the subsystems in ID WORLD generalizes when multiple machines are used. Chapter six presents the three abstract machines supported by ID WORLD. In chapter seven I presents conclusions and offer suggestions for future improvements to ID WORLD. Finally, the Appendix contains the ID WORLD users manual.

GITA, the Graph Interpreter for the Tagged-Token Architecture, was designed by Ken Traub and Richard Soley and was originally implemented by the author and Richard Soley. Improvements to GITA have been made by by David Culler, Greg Papadopoulos, Andrew Chien, Steve Heller, and Bhaskar Guharoy.

2.1 Background

The first method of executing ID programs involved using a program called IDSys which took ID programs, compiled them into MacLisp, and executed them. This ID to LISP compiler was followed by an ID to Graph compiler, which compiled ID procedures directly into dataflow graphs. A TTDA Simulator was then created to execute these dataflow graphs. At the time it was envisioned that ID programs would be debugged on IDSys, and only working programs would be run on the TTDA Simulator.

The TTDA Simulator was followed by a TTDA Emulator, written in LISP for the Multiprocessor Emulation Facility (MEF [5]), which was supposed to be a much faster interpreter of dataflow graphs. The TTDA Emulator was eventually abandoned, however, partially because it failed to perform as expected, and partially because the MEF was in the process of changing over to using T.I. Lisp Machines instead of Symbolics Lisp Machines.

The ID to Graph compiler was eventually ported to the Lisp Machine, but IDSys was not. The only way left to execute ID procedures was the TTDA Simulator. Debugging programs on the TTDA Simulator was very difficult because both the ID to Graph compiler and the TTDA Simulator were still relatively untested and full of bugs. Gino Maa eventually succeeded in debugging a 1200 line ID program, but only after many months of effort. It was suggested that perhaps a version of the simulator which had only one fast

processing element would simplify the debugging of programs. It was then suggested that the TTDA Emulator could be resurrected and simplified. Finally, Ken Traub suggested that it would be easier to create a new interpreter, called GITA.

2.2 GITA vs. The TTDA Emulator

GITA is an object-oriented dataflow interpreter, which replaces the Tagged-token dataflow emulator on the MEF. In this section I will point out some of the poor design decisions in the TTDA Emulator, and show how they are solved in GITA.

2.2.1 Data Structures

One of the biggest differences between GITA and the TTDA Emulator involves how each implements the fundamental data structures necessary for executing dataflow graphs.

In the TTDA Emulator, tokens were represented as 32 contiguous bytes of data at some offset into a large array called the *token store*. Accordingly, whenever a part of the Emulator had a reference to a token, all it really had was an integer specifying the offset into the array. This made it very difficult to debug the TTDA Emulator because it was impossible to tell if a variable was referencing a token by just looking at its value. The user would have to somehow know that if the value was an integer, then it might be referencing a token. In addition, because the data inside a token was represented as a series of bytes, it was necessary to write encoding and decoding functions for each of the slots in the token data structure. Furthermore, there needed to be debugging functions which would take an offset into the token store and print a human-readable representation of the contents of the token.

These problems were avoided in GITA by simply representing a token as a structure of type **token**. By making a token a distinct, recognizable object in GITA, it is much easier to debug code since there is no longer any confusion about what a variable actually references. Also, since a token is made up of several slots which can each contain a reference to any LISP object, there is no need to perform any encoding or decoding of data. GITA, in fact,

maps the data types in ID directly into the primitive data types of LISP, where possible, and where not possible a new LISP data type is created to represent the ID data type. For example, the ID data types boolean, integer, and real number, are directly represented in LISP by booleans (`t` and `nil`), integers, and floating-point numbers. Since I-Structures don't map directly to any primitive LISP object, however, a new data type was defined to represent them.

One disadvantage of representing the data structures in GITA as objects is that the object representation takes up considerably more storage than does the scheme used by the TTDA Emulator. For example, in the TTDA Emulator the instructions for a dataflow graph were represented as an array of 8-bit bytes, and a single instruction was represented as an offset into this array. On average a single instruction was 8 bytes in the TTDA Emulator. In GITA, however, an instruction is an object of type `instruction`, and takes up about 48 bytes of storage, a 600% increase. The reason for this large increase in size is that the 8 bytes in the TTDA Emulator actually encode about 12 different pieces of information. In GITA each piece of information is given its own slot in an instruction object, and each slot is large enough to reference any other LISP object. On the LISP machine this translates into 4 bytes per slot, for a total of 48 bytes.

In GITA, this inefficient use of storage is justified for two reasons. First, GITA is running on machines with large virtual address spaces. There is no need to waste a lot of effort saving space if you have more than you can possibly use already. And second, as a result of the inefficient use of storage, GITA can execute dataflow graphs considerably faster than the TTDA Emulator. At most, the TTDA Emulator could execute approximately 250 dataflow instructions per second. GITA, on the other hand, currently executes dataflow instructions at the rate of 2,500 per second.[1] This increase in speed is not surprising since GITA has to do almost no encoding or decoding of data before using it. The TTDA Emulator needed to decode the bit patterns of each instruction each time it interpreted it. In GITA, the bit patterns are decoded once at load time and are stored in a

[1] These rates are from each system running on a Symbolics 3600 Lisp Machine.

more easily accessible, albeit more verbose, format. In sum, a better time/space tradeoff was made in GITA than in the TTDA Emulator.

2.2.2 Error Handling

Another difference between GITA and the TTDA Emulator involves how each approached the problem of error handling. When either interpreter is executing a dataflow graph, it must make sure to catch any run-time errors, such as overflow, underflow, division-by-zero, etc., and report them to the user. There are essentially two ways to approach this problem: *eager error handling* or *lazy error handling.*

The TTDA Emulator took the eager error handling approach. What this means is that before it would execute any instruction, it would make sure that no errors were going to happen. For example, before performing a division, the TTDA Emulator would check that the denominator was non-zero. It would then check for the possibility of positive and negative overflow, and positive and negative underflow. Only if it was sure that the division would not cause any of these errors would it finally go ahead and perform the division. This approach is analogous to the strategy of touching all the pages that an instruction in a virtual memory system is going to access, and only after all the pages are guaranteed to be in physical memory is the instruction actually executed.

In GITA, the lazy error handling approach is taken. What this means is that GITA does not bother checking the arguments to an instruction before it interprets it. It just assumes that the arguments are of the correct types and that no error is going to happen. Most of the time, this turns out to be a correct assumption. There are times, of course, when the assumption is incorrect, such as when a divide-by-zero error happens. But for this case GITA just lets the Lisp Machine error handler catch the error, and intercepts it before the LISP debugger is invoked. It then figures out which part of the machine got the error (the ALU in this case), and records any relevant information (such as the arguments to the instruction) so that it can explain and analyze the error at a later time. Finally, it causes the instruction to abort and goes on to the next one. This approach is analogous to being able to back out of the execution of any instruction when a page-fault occurs in a virtual memory

system. In GITA this is always possible because the commit point in the execution of dataflow instructions occurs when a token carrying the answer is actually transmitted to the next instruction, and this doesn't happen until the answer is computed. If an error doesn't happen while the answer is being computed, then it isn't going to happen.[2]

The advantages of lazy error handling over eager error handling are clearly demonstrated by examining how the TTDA Emulator and GITA each execute a floating-point multiplication instruction. In the TTDA Emulator, execution of a single floating-point multiplication instruction required 3 function calls, 4 floating-point relational operations, 2 floating-point divisions, and 1 floating-point multiplication. The 4 relational operators and 2 floating-point divisions were done to make sure that the multiplication would not overflow. GITA, however, executes the same instruction with just 1 function call and 1 floating-point multiplication. Should the multiplication overflow the LISP system would raise an exception, which would be caught by GITA.

2.2.3 The Basic TTDA Emulator Flaw

The previous two design flaws in the TTDA Emulator were actually a result of a more fundamental problem in its design: it was designed to behave too much like a real Tagged-Token dataflow machine. This is ironic since the TTDA Emulator was supposed to be a "soft" implementation of the machine. It was hoped that by emulating the machine in software, the flaws in the design of a real machine could be ironed out before it was actually built. The problem, however, was that the Emulator became so unmanageable that even the smallest changes to it were extremely hard to make. In effect, the Emulator became almost as unmodifiable as hardware would have been.

---

[2] Assuming, of course, that the transmission of the answer does not cause an error. Unless GITA itself has a bug in it, the transmission will always succeed.

2.3 The Structure of GITA

This section presents the internal organization of GITA. It assumes the reader is familiar with the Tagged-Token Dataflow Architecture, and the dataflow language ID.

The internal organization of GITA is shown in figure 2-1. GITA can receive three different types of input, each of which is placed in its own queue. During each cycle, GITA removes and processes one entry from each of these three queues. We examine each of these queues individually.

2.3.1 The Token Queue

The token queue contains tokens which have left the instruction which created them, but have not yet arrived at their destinations. That is, it contains all the tokens still riding on the arcs of the dataflow graph (but not including those which are sitting in the waiting-matching section).

GITA processes a token from the token queue as follows. It first looks to see if the instruction is unary. If so, then it sends it directly on to the ALU section, bypassing the waiting-matching section. Otherwise, the token is binary. In this case the waiting-matching section is searched for the partner of the token. If a partner is not found, the token stays in the waiting-matching section until his partner arrives. If a partner is found, then it is removed from the waiting-matching section and both tokens are forwarded to the ALU section.

The ALU section looks at the tags on the tokens to find out what operation must be performed. This operation will either be an arithmetic operation, an I-Structure operation, or a Manager operation. In the case of an arithmetic operation, the ALU simply takes the two data values riding on the tokens, computes a result, packages it into a token, and sends the token to the destinations of the instruction it just executed. In the case of an I-Structure operation, the ALU takes the necessary information from the two tokens and outputs an I-Structure request. A Manager operation is similar, except that the ALU outputs a Manager request.

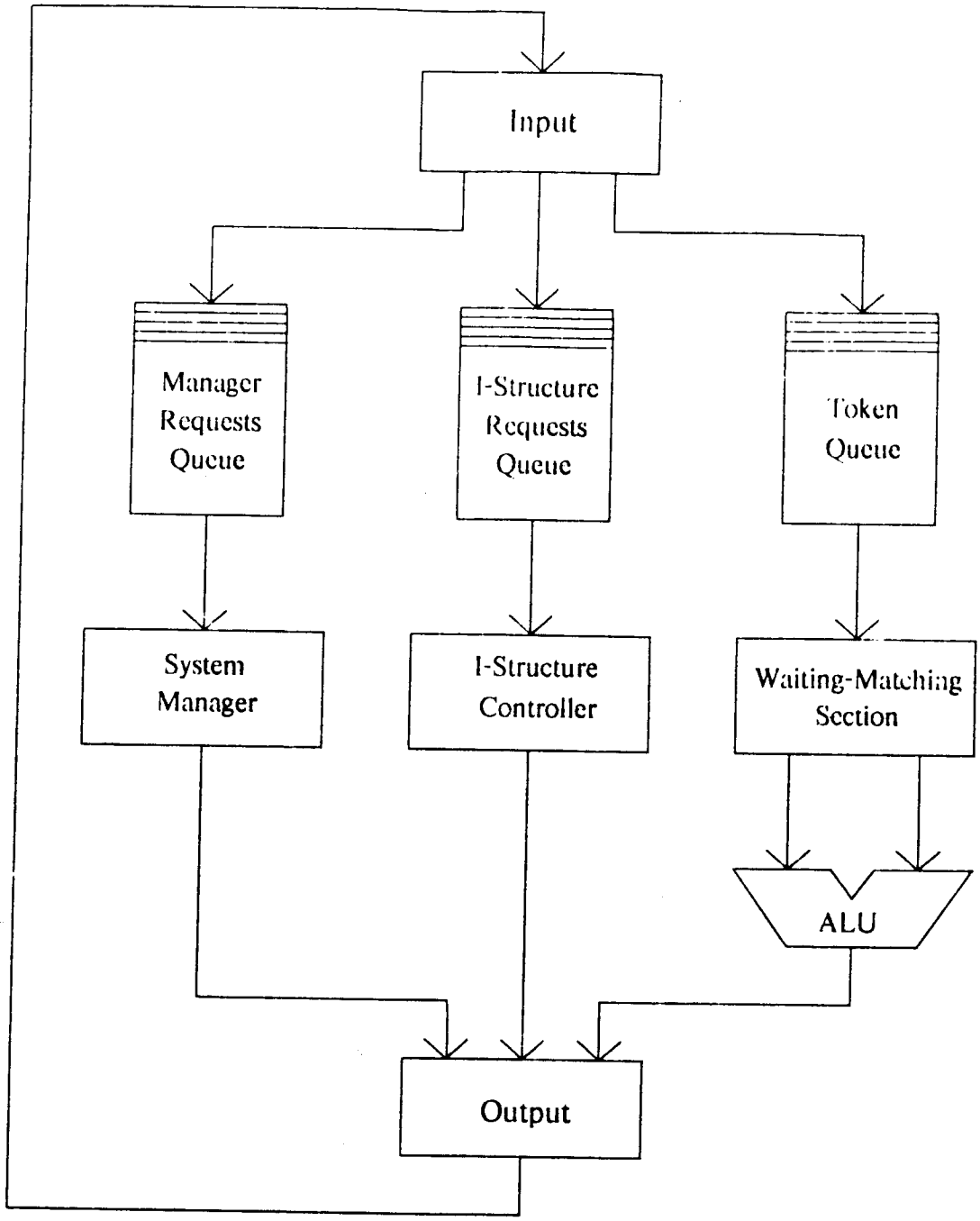

Figure 2-1: Organization of GITA

2.3.2 The I-Structure Requests Queue

The I-Structure Requests Queue contains unprocessed requests for the I-Structure controller. GITA processes a request from this queue as follows. It looks at the request-type field of the I-Structure request object to figure out what operation needs to be performed. This can be either I-Fetch, I-Store, Increment Reference Count, or Decrement Reference Count.

In the case of an I-Fetch, the rest of the request will contain a reference to an I-Structure, an index, and a destination. The destination specifies where to send the token containing the value to be read. The I-Structure controller first looks to see if the given slot in the I-Structure has already been written. If so, it takes the value stored there, packages it into a token, and sends it to the destination. If the slot is empty, then the I-Fetch is said to be *deferred*, and the request is suspended until a write of the slot occurs, at which time the request will be satisfied.

In the case of an I-Store, the rest of the request will contain a reference to an I-Structure, an index, a value to store, and a destination. The destination specifies where to send an acknowledgment token, which declares that the I-Store has taken place. The I-Structure controller looks to see if the given slot in the I-Structure has already been written. If so, an error is signalled since an I-Structure slot may be written only once. Otherwise, the controller sees if there are any I-Fetches waiting for the value in the slot about to be written. If so, it packages the data value into a token and sends it to the destinations given by each deferred I-Fetch. Finally, the value is stored in the slot, and an acknowledgment token is sent to the destination specified by the current I-Store request.

In the case of Increment Reference Count, the rest of the request will contain a reference to an I-Structure, a number specifying the amount by which to increment the reference count, and a destination. The destination specifies where to send an acknowledgment token after the reference count is adjusted. The I-Structure controller simply increments the reference count by the given amount and sends the acknowledgment token to the destination. Decrement Reference Count is done in exactly the same way, except that the

reference count is decremented.

2.3.3 The Manager Requests Queue

The Manager Requests Queue contains unprocessed requests for the System Manager. GITA processes a request from this queue as follows. It looks at the request-type field of the Manager request object to figure out what operation needs to be performed. This can be either Invoke or Terminate.

In the case of Invoke, the rest of the request will contain a reference to a procedure object, an I-Structure Descriptor (ISD) for the arguments, an ISD for the Results, and a reference to the context which is invoking this procedure. The manager creates a new context for the invocation of the procedure, fills it with the ISD's for the arguments and results, and stores the reference to the caller context in the new context. It then creates a token and sends it to the first instruction in the new procedure, causing the procedure to begin execution.

In the case of terminate, the rest of the request will contain a reference to the context to be terminated. The manager then releases any resources used by the context and sends a signal to the caller telling it that one of its sub-procedures has terminated.

2.3.4 Beginning and Ending

The previous three sections describe what GITA is doing when it is running, but how does the execution of a dataflow graph start and end?

To begin the execution of a procedure, GITA simply creates a manager request which asks that the procedure be invoked with a certain set of arguments. This manager request is simply placed into the manager request queue and the main cycle of GITA is started. The system manager will then process the request, placing the initial token to be dropped into the dataflow graph of the procedure into the token queue. From then on GITA continues processing each queue until there is nothing left to do.

The interpretation of the dataflow graph is said to have ended when there are no more entries on any of the Manager Request, I-Structure Request, or Token queues. Normally this means that there are no more tokens in the waiting-matching section, and no suspended (i.e. deferred) requests left in the I-Structure controller. However, if a run-time error was detected during execution, or if the dataflow graph was not well formed (because the compiler generated bad code, for example), then there may still be tokens or suspended I-Structure requests left around. When GITA finishes executing it will warn if the execution did not end properly.

The GITA debugger is in many ways like the normal LISP debugger. The main difference is that the LISP debugger allows you to look up and down a stack of frames, while the GITA debugger allows you to look around a tree of contexts.

3.1 The Error Handler

When GITA detects an error during execution of a procedure a message is printed saying in which part of the machine the error occurred. Execution continues, however, until there are no more activities ready to fire. At the end of execution GITA reports the total number of errors it encountered.

The GITA Debugger has commands which let you view the errors which occurred during the last GITA run. To debug a particular error you would type a command to the debugger telling it that you are interested in that error. The debugger will then set both the current and anchor contexts to the context in which the error occurred.

3.2 Invocation Tree

In GITA, a *context* corresponds to a *stack frame* in sequential languages. Whenever a procedure is invoked a context is created to hold its arguments and results, and any other information particular to the procedure's invocation. Because a procedure can execute sub-procedures in parallel, however, GITA must maintain a tree rather than a stack of contexts.

An example of an invocation tree is show in figure 3-1. The *root context* never changes and is the one context which has no father. It corresponds to the top-level procedure invocation made which started GITA running, and is shown in the figure as context 1. The *current context* corresponds to the context which is currently being examined. As you move

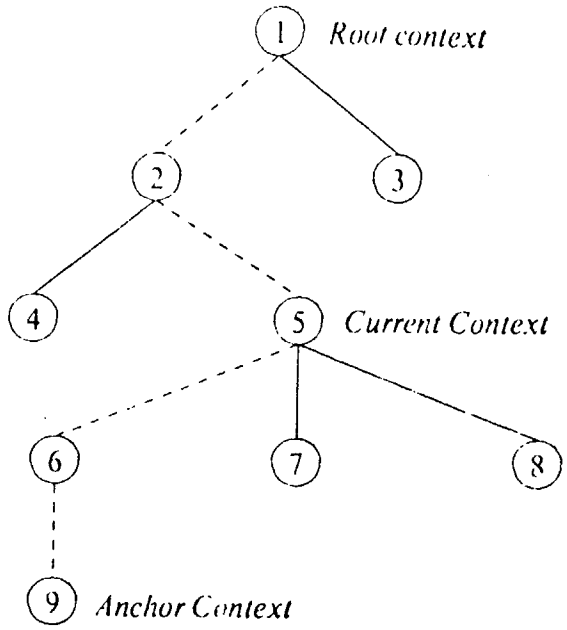

Figure 3-1: An Invocation Tree

around the tree of contexts, the *current context* is changed to reflect your position in the tree. In the figure, context 5 is shown as the current context. Finally, the *anchor context* is a context which usually corresponds to a context at which an error occurred, although is can be changed to any context at all through a debugger command. This anchor context is used in order to facilitate moving up and down the tree of contexts, as explained below.

3.3 Moving around the Invocation Tree

Moving around the tree of invocations in the GITA debugger is not quite as easy as moving up and down the stack in the LISP debugger. A context will have at most one father (the caller), but may have several sons (each corresponding to a procedure invocation which has not yet terminated).

Moving "up" the tree of invocations is straightforward, the current context simply gets set to its father. Moving "down", however, requires that a branch be selected from among its sons. For example, in figure 3-1 the current context is 5. Moving up the context tree

from context 5 would make the current context be 2. To move down, however, a choice must be made between the three contexts which are the sons of context 5 -- contexts 6, 7, and 8. Of the three possible choices, context 6 seems the "obvious" choice. This is because it is closest to the *anchor context*. The user will probably wish to return to the anchor context often since it is the one at which the error being debugged occurred. Furthermore, it is unlikely that the user will be interested in any contexts not on the path from the anchor context to the root context (shown as dashed lines in figure 3-1) since they probably had little to do with the cause of the error. In the GITA debugger, moving down the tree of invocations implies moving down the "obvious" choice, if there is one. This makes moving around a tree of invocations in the GITA debugger just as easy as moving up and down a stack in the LISP debugger.

3.4 Debugger Commands

This section briefly describes some of the more important GITA debugger commands. See the Appendix for a more complete description along with a listing of all the commands.

3.4.1 Examining the Current Context

There are many commands in the GITA debugger designed to return information from the current context. With a single keystroke you can get at any argument or result value, the ISD used to hold the arguments, the ISD used to hold the results, a local value (token), the current procedure object, or even the context itself.

As it happens, some of the information is not very helpful. To someone who does not understand the inner workings of GITA, looking inside the context or procedure object will shed little light on what went wrong. In addition, all you see when you look at the local values in a context is a bunch of tokens carrying data headed for different instructions. The problem is that the current ID compiler does not provide a mapping from arcs in the dataflow graph to names in the source code. If it did, then it would be possible for the GITA debugger to provide information such as "local variable X in procedure P has the value Y". Instead, all that can be provided now is "some local variable in procedure P has

the value Y." A second version of the ID compiler is currently being designed which will output the mapping from arcs to names. As soon as it is complete, the GTA debugger will be much more informative.

Probably the most important piece of information provided by the GTA debugger is the values of the arguments given to a procedure. If an error occurred in procedure P which was called from procedure Q, you can usually debug procedure P independently by editing it, compiling it, and then calling it directly (without going through procedure Q) with the arguments which caused it to fail, until the procedure works as expected. This "bottom-up" style of programming is facilitated by the rapid edit, compile, debug loop provided by ID WORLD.

3.4.2 Backtrace Commands

A *backtrace* is a listing of the contexts in reverse order starting from the current context and ending at the root context. There are two backtrace commands which differ only in how much detail they provide about each context. One shows only the name of the procedure for each context, while the other shows the name of the procedure along with its arguments.

These commands are useful for figuring out where in the execution of a large program the error occurred. For example, suppose you were trying to debug a recursive procedure P, and an error occurred in some call to the procedure. If the backtrace shows that only one call to procedure P has been made, then the procedure probably failed in the part which was to do the recursion. On the other hand, if the backtrace shows many calls to procedure P, then the procedure probably failed in the part which terminates the recursion (the base case).

Even in the case of non-recursive procedure invocations, the backtrace shows you the particular sequence of procedure invocations which led to the current error. This information is sometimes enough to tell you what went wrong.

3.4.3 Searching

There is a command which will search for a context with a procedure whose name contains a given substring. The contexts are searched starting from the father of the current context toward the root context. This is useful for quickly jumping to some context shown in a backtrace. For example, if the backtrace is:

```
FOO [1] <- BAR [2] <- BAZ [3] <- QUUX [4]
```

Then searching for "BA" will make the current context be `BAZ [2]`, and searching for "F" will make the current context be `FOO [1]`.

Both the Symbolics and Texas Instruments Lisp Machines have a built-in editor called ZMACS, based on the EMACS editor. One of the features of both editors is that they have the concept of a *major mode*. A major mode tells the editor what kind of document is being edited. Every buffer has a major mode.

One of the most often used major modes on the Lisp Machine is, obviously, LISP mode. When LISP code is being edited, LISP mode tells the editor how to recognize LISP procedures, how to move around LISP structure quickly, how to compile LISP procedures, etc.

In ID WORLD there is a new major mode called ID Mode, which defines several commands that recognize the structure of ID procedures, and interfaces the ID Compiler and GITA in order to simplify program development. As of this writing, ID Mode currently knows how to do the following:

- It is able to move the cursor to the beginning and end of ID procedures.
- It understands about comments in ID, and can insert and remove them at the end of lines of code.
- It can send one, two, or up to an entire buffer full of ID procedures to the ID compiler, and will automatically load the output of the compiler into GITA.

In the future, the following commands will be added to ID Mode:

- Indent for ID. By hitting the TAB key, the current line of code will automatically be indented to the correct column.
- The ability to send a batch job to the TTDA Simulator on an IBM mainframe directly from the editor.

The major advantage of ID Mode is that it provides a sort of "control panel" to the rest of ID WORLD. Without ever leaving the editor, it is possible to write, compile, execute, and debug ID procedures. In addition, even when the ID language is changed (to ID/83s), the commands in ID Mode will remain the same. See the appendix for a complete description of the ID Mode commands.

When writing code for a parallel dataflow machine, one should not have to be concerned with the number of physical processors which will ultimately be executing his program. That is, if one doubles the number of processors on his dataflow machine, all programs should run without change to the source code. Doubling the number of processors should only have the effect of speeding up the execution of those programs.

Similarly, one of the design goals of ID WORLD is that the user should not have to be concerned with the actual number of machines cooperating in the execution of his programs. Increasing or decreasing the number of physical processors should be transparent to the user of ID WORLD in that whenever ID WORLD is being used in multiple machine mode, each of its subsystems should automatically generalize to multiple machines.

5.1 The GITA Server

In order for ID WORLD to be used on multiple machines, the user requests that some number of additional machines be allocated to him.[3] The user's machine is known as the *master machine*, and the additional machines are known as *server machines*.

In order to allow for communication between the master machine and the server machines, a GITA Server connection is made over the EtherNet to each of the server machines. The rest of this chapter describes how each of the subsystems of ID WORLD use the GITA Server when running on multiple machines.

[3]Currently there is no good way to do this. In the future we envision an ID WORLD server running on some machine which will be responsible for allocating and deallocating machines in the facility.

5.2 GITA

GITA can be extended to run on multiple machines in many different ways. One way is to map the dataflow graphs of procedures across several machines so that different parts of the graph may be evaluated in parallel. This is the approach taken by the TTDA Emulator. While it is believed that something like this will have to be done eventually, it would have required substantial changes to the single-machine version of GITA, and so a less desirable but more easily implemented approach was taken.

It was decided that the procedure invocation would be the smallest unit of work shipped to other machines. Thus, whenever procedure P called procedure Q, GITA would decide where the invocation of Q should take place. Currently each machine performs round-robin scheduling of procedure invocations among all the server machines. The first procedure invocation is done on the local machine, the next on the first server machine, the next on the second server machine, etc. With each machine doing its own round-robin scheduling, the division of work tends to spread out over all of the machines in a fairly uniform way.

One of the problems with having more than one machine sharing in the execution of a dataflow program involves the access to I-Structures. In the single-machine version, all I-Structures were created on the local machine and access to them was easy -- just read the n^{th} slot. In the multiple machine version, however, it could be that an I-Structure passed in as an argument to a procedure was actually created on some other machine. In this case a request must be sent to the machine which created the structure and a reply must be sent back. This information could be sent over the EtherNet, but the EtherNet would quickly become the bottleneck in the system. Instead, a high-speed circuit switch network is utilized by GITA whenever communication of information between machines cooperating in the execution of a program is required.

The multiple-machine version of GITA uses the GITA Server only to start and stop each of the server machines. That is, the user tells the local machine to execute a procedure. The local machine then instructs each of the server machines to get ready to cooperate in the

execution of the procedure. Each of the server machines initializes internal data structures and then listens to the circuit switch network for requests to invoke procedures or read I-Structure slots. The local machine then executes the procedure, which will cause other machines to receive requests to invoke procedures, which will in turn cause more machines to receive requests to invoke procedures, etc. Eventually, the local machine detects that the initial procedure invocation has terminated, and uses the GITA Server to tell each of the server machines to stop listening to the circuit switch and idle until they are needed next.

5.3 The GITA Debugger

When GITA is run on multiple machines, errors are trapped by each machine involved in the execution of the program. Whenever a machine detects an error, it records enough information to explain what went wrong, reports to the master machine that something went wrong, and aborts the operation which caused the error. Because the operation was aborted before it could send its result to the next operation, the execution of the entire program will eventually come to a premature halt. At that point, each machine will be left with a certain number of procedures which are only partially executed.

When the user decides that enough errors have occurred, he enters the GITA debugger. At that point, the local machine must do two things: collect all errors, and build an invocation tree. To collect the errors the local machine simply uses the GITA Server to request from each machine the errors that were trapped during the last program execution. Later when the user picks one of these errors to debug, the current context will automatically be changed to the context in which the chosen error occurred.

Building an invocation tree is a bit more tricky because the contexts still active are spread out over the server machines. To simplify matters, the GITA debugger uses the GITA Server to collect each of the active contexts on the remote machines and builds one large invocation tree on the local machine. After doing this, it knows the context which called each context (the "up" links in the tree), as well as the active sub-contexts for each context (the "down" links in the tree). Furthermore, it knows on which machine each context was invoked.

Even when GITA is run in single machine mode, an invocation tree is built when the GITA debugger is entered. In this case all contexts in the tree will have been executing on the local machine. The reason for doing this is to provide the GITA debugger commands a uniform view of the world. Whenever a command requires information from a context in the invocation tree, a function is called requesting the information. The job of the function is to get the requested information anyway it can. If the context is local, then it can simply get the information from the data structures in the local machine. If the context is non-local (i.e. it was running on some other machine), then the GITA Server is used to request the information from the server machine. When the information is returned, it is cached locally so that it will not have to requested again, and is returned to the command which requested the information.

Thus, not even the GITA debugger commands need be concerned with the number of machines involved in the execution of a program. By separating the requesting of information from the obtaining of the information, the GITA debugger provides an identical interface no matter how many machines are in use.

5.4 ID Mode

The only part of ID Mode which needs to be concerned with the number of server machines in use is the part which sends the object code from the ID Compiler to GITA for loading. When running of multiple machines, each machine must be given the object code. This is because, unlike the TTDA Emulator, GITA does not transmit programs to server machines which do not have them loaded at run-time. Instead, ID Mode makes sure that all machines always have the latest compiled version of any procedure.

The main reason for insisting that each machine always have the same procedures loaded is that it minimized the changes to GITA necessary to make it work on multiple machines. In the future GITA may be changed to do dynamic loading of procedures.

Finally, a change was made to ID Mode not to *support* multiple machines but rather to *exploit* them. Whenever several machines are being used, and more than one procedure is

to be compiled (as, for example, when an entire buffer is compiled), the procedures are farmed out to each of the server machines for compiling. Thus, the time required to compile a group of procedures is greatly reduced.

ID WORLD currently supports three different abstract machines on which to execute dataflow programs. Each of the machines is similar in that they accept the same output generated by the ID Compiler, and, when told to execute a procedure with a given set of arguments, will each produce the same answer. The main difference involves how closely each attempts to simulate the actual Tagged-Token Dataflow Machine.

The first two abstract machines, GITA-I and GITA-E, are actually the same program, GITA, being run in two different modes. GITA-I is GITA running in *idealized* mode, and GITA-E is GITA running in *emulation* mode. We treat them as different abstract machines because the statistics they generate differ radically and are based on two different dataflow execution models. The third abstract machine, called SITA (the Simulator for the Tagged-Token Architecture) is implemented in PASCAL and runs on an IBM 4381. Each of these machines is examined individually in the remainder of this chapter.

6.1 GITA-I

GITA-I is short for GITA-Idealized, and is realized in software by running GITA in idealized mode. Of the three abstract machines, it does the poorest job of simulating the Tagged-Token Dataflow Machine. In fact, GITA-I makes no attempt to simulate *any* realizable machine. It is best thought of as a machine which captures the abstract behavior of the U-Interpreter [3].

The statistics collected for a program executed under GITA-I are based on the following assumptions:

- All activities ready to fire are fired in the same timestep
- All activities take one time unit to execute

- There is zero communications delay
- Unbounded resources are available

In GITA-I a *timestep* is defined to be the amount of time it takes to process all activities which are ready to fire at the beginning of the timestep. Any activities which become ready to fire as a result of the firing of an instruction during timestep n will fire during timestep $n + 1$.

The statistics collected by GITA-I should be interpreted as measuring the *inherent* parallelism in a program. The idealized statistics for a given program will not change when more machines are used to help execute the program, nor will they be affected by real-world problems such as poor load-balancing, or high communications overhead.

When a program has been debugged and is being used for experimentation it should be run once on GITA-I in order to determine the ideal performance of the program. Should this ideal performance show that little parallelism exists in the program, then one should not be surprised to find that a more realistic machine faired poorly in executing the program. On the other hand, if the ideal performance shows a great deal of inherent parallelism and yet the realistic machine failed to exploit enough of it to keep it busy, it may indicate a problem with the implementation of the realistic machine.

In sum, then, statistics collected by GITA-I should be used as a control element when experimenting with various parameters of the more realistic machines.

6.2 GITA-E

GITA-E is short for GITA-Emulation, and is realized in software by running GITA in emulation mode. Statistics collected by GITA-E come closer to describing how a real dataflow machine would perform on a program. GITA-E differs from a real dataflow machine in the following ways:

- The minimum amount or work which can be sent to another processor is an entire procedure. This sometimes causes a few processors to be busy executing

large, complex procedures, while other processors sit idle. In a real dataflow machine the minimum amount of work could be as little as a single instruction.

- I-Structures in GITA-E are always mapped onto one processor. This sometimes causes a single processor which just happened to create a structure which is referenced very often (such as a table of values used by a table-lookup routine) to be swamped with I-Structure requests. In a real dataflow machine I-Structure could be mapped across several processors, such that the first processors contains element 1, the second element 2, etc.

- GITA-E is not pipelined, whereas a real dataflow machine would be heavily pipelined.

- I-Structure requests and ALU operations are processed sequentially, whereas a real dataflow machine would process them in parallel.

Even with these differences, the statistics given by GITA-E can be used to get a feel for how the setting of various parameters in a real machine might affect its performance on certain types of programs. For example, by changing the number of machines participating in the execution of a program, one can see how well GITA-E was able to divide the work among the machines. In addition, it is possible to get a feel for how large certain parts of a real dataflow machine might have to be in order to be able to run large programs. One of these parts is the waiting-matching store. If the waiting-matching store is too small, certain programs will not be able to run. By running many different programs on GITA-E it is hoped that one will be able to get a better feel for the waiting-matching storage requirements of a real machine.

6.3 SITA

SITA, the Simulator for the Tagged-Token Architecture, is realized in software by running a PASCAL program on an IBM 4381. SITA was designed to simulate to a very low level the execution of a real dataflow machine. The statistics collected by SITA are very detailed and can provide precise information on the dynamics of a real dataflow machine.

There are two main drawbacks with using SITA, however. The first is that, unlike all other subsystems of ID WORLD, SITA does not run of a Lisp Machine. In order to use

SITA, a batch job must be submitted to the IBM 4381, and only some time later will the job be finished and the results be ready to examine. This mode of operation does not fit in too well with the highly interactive environment of ID WORLD.

The second drawback is that SITA is very slow in executing programs. It currently executes approximately 80 dataflow operations per second, compared to GITA-E which executes approximately 800 dataflow operations per second *per processor.*

The user of ID WORLD will have to decide for himself which is more important: faster execution and less true-to-life statistics (GITA-E), or slower execution and very realistic statistics (SITA).

6.4 Collecting Statistics in ID WORLD

Currently it is very easy to collect and view statistics generated by GITA-I or GITA-E. ID WORLD has yet to be interfaced to SITA, however.

See the appendix for a complete description of the statistics currently available, as well as a listing of the commands used to view them.

Chapter 7

Conclusion

ID WORLD can currently be used on any Symbolics or T.I. Lisp Machine (standalone configuration), or on the group of 32 T.I. Lisp Machines connected by a high-speed circuit switching network which comprise the Multiprocessor Emulation Facility at the Laboratory for Computer Science, M.I.T.

So far ID WORLD has proven valuable to the members of the Computation Structures Group who find that the highly interactive nature of ID WORLD greatly simplifies the task of developing dataflow programs written in ID.

7.1 Future Directions

ID WORLD is still a relatively new environment which is likely to change and evolve as more people use it. There are two places where I feel that improvements should be made soon.

The first involves the handling of statistics collected by each of the three abstract machines. Currently there is only minor support for the viewing of statistics collected from GITA-I and GITA-E, and there is none for viewing statistics from SITA (at least not in ID WORLD). A mechanism needs to be designed which will allow statistics from all three abstract machines to be collected, stored, retrieved, and compared, in a consistent manner.

The second improvement which must take place soon involves the way ID WORLD currently interfaces to the circuit switching network on the MEF. The problem with the current interface is that *there is none*. When one subsystem of ID WORLD wishes to send a message over the circuit switch it must use very low-level primitives to instruct the hardware to send a number of words from the current machine to some other machine. The problem with this approach, of course, is that there is no one uniform interface to the circuit switch

which everyone can use. I envision a substrate, somewhat like the old MIT generic software substrate, being implemented to provide a uniform communication abstraction on top of the circuit switch.

This appendix is a reference manual for ID WORLD, currently being used by the members of the Computation Structures Group in the Laboratory for Computer Science at M.I.T.

ID WORLD interfaces the ID Compiler Version 1, GITA, the GITA Debugger, and ID Mode, in order to simplify the development of dataflow programs written in ID. The ID Compiler Version 1 was written by Ken Traub and Steve Heller, and is based in large part on the ID Compiler Version 0 by Vinod Kathail. GITA, the Graph Interpreter for the Tagged-Token Architecture, was designed by Ken Traub and Richard Soley and was originally implemented by the author and Richard Soley. Improvements to GITA have been made by David Culler, Greg Papadopoulos, and Steve Heller. The GITA Debugger and ID Mode were designed and implemented by the author.

In order to begin using ID WORLD, you must make sure that the machine you are using has loaded all of the programs which make up ID WORLD. Currently these are:

- The Graph Interpreter for the Tagged-Token Architecture (GITA)
- The ID Compiler Version 1
- ID Mode

If you are using one of the Lisp Machines belonging to the CSG group at the M.I.T. Laboratory for Computer Science, whether it be a Symbolics 3600 series or T.I. Explorer, then ID WORLD is already loaded into your environment.

If you are on a non-CSG Lisp Machine, you can load ID WORLD by loading the file **OAK:>ID-WORLD>ID-WORLD.LISP**. This file contains all the commands necessary to load all the parts of ID WORLD.

ID Mode is a new major mode for ZMACS, the editor available on T.I. and Symbolics Lisp Machines. It defines several commands which understand about the structure of ID procedures, and interfaces the ID Compiler and GITA in order to simplify program development.

In the commands which follow, abbreviations are used for the control and shift keys. They have the following meanings: '`c`' means the *control* key, '`m`' means the *meta* key, '`s`' means the *super* key, '`h`' means the *hyper* key, and '`sh`' means the *shift* key. A command such as `c-sh-C` is typed by holding down the *control* and *shift* keys and typing the '`C`' key. The command `c-m-A` is typed by holding down the *control* and *meta* keys any typing '`A`'.

Commands such as `m-X Compile File` are typed in as follows. First hold down the *meta* key and type '`X`'. In the small window at the bottom of the editor window you will be prompted with `Extended Command:`. Type in `Compile File` and hit `<RETURN>`.

2.1 The File Attribute List

It is a good idea to put the following line (called the *attribute list*) at the top of every file containing ID procedures that you will executing in ID WORLD:

```
! -*- Mode: ID; Package: GITA -*- !
```

Whenever a file which has this line at the top is loaded into the editor, ZMACS will automatically set the current major mode to be ID Mode, and the current package to be the GITA package. If you load in a file containing ID procedures which does not have this line at the top, then you should add it and then type `m-X Reparse Attribute List`. This will tell the editor to look at the attribute list and set the major mode and package to ID and GITA, respectively.

Alternatively, the following two editor commands can be used to construct the attribute list at the top of a file, or correct it if it is wrong.

m-X ID Mode *ZMACS Command*

Sets the major mode for the current buffer to ID Mode, a mode for editing ID programs.

m-X Set Package *ZMACS Command*

Prompts for the name of a package to become the default for the current buffer. When editing ID programs you should answer the prompt with **GITA**.

After each of these commands you will be asked if you want the attribute list at the top of the buffer to be updated. You should answer **Yes** to the question. The attribute list will either be created if one doesn't exist, or the appropriate field will be updated with the correct information. These two commands should be invoked whenever a new file or a new buffer is created in order to correctly set up ID Mode.

You can tell if you are in ID mode because the editor mode line (the top most line in the small window at the bottom of the screen) will say something like:
ZMACS (ID) SIMPLE.ID >ARVIND> OAK:

The symbol in parenthesis after the word **ZMACS** always specifies which major mode is currently active. If you switch to another buffer containing LISP code, for example, the major mode becomes LISP. However, it will revert back to ID mode whenever you go back to editing a buffer containing ID.

You can also tell which package you are currently in by looking down around the center of the very bottom line on the screen. There you will find a symbol with a colon after it. This symbol tells you what the current package is. When you are editing ID programs and are in ID mode you should see **GITA:** down there. While a procedure is being compiled you will see it change to **IDV1:**. What this means is that the ID Compiler Version 1 is busy compiling the procedure. When it finishes, the package will automatically be reset to **GITA:**.

2.2 Commands in ID Mode

There are a few special commands in ID Mode which "understand" ID procedures. When the following commands talk of the *current procedure* what they mean is the ID procedure on which the cursor is currently placed. If the cursor is between two procedures, then the earlier one is said to be the current one.

c-m-A *ZMACS Command*

Moves the cursor to the beginning of the current procedure. That is, the cursor is moved so that it is on the 'p' of the keyword 'procedure'.

c-m-E *ZMACS Command*

Moves the cursor to the end of the current ID procedure. That is, the cursor is moved just past the last character in the current procedure.

c-m-H *ZMACS Command*

Sets the region to be the current procedure (marks the current procedure). That is, the point is set to the beginning of the current procedure, and the mark is set to the end. See the ZMACS manual for an explanation of points, marks, and regions.

The following commands are used to edit comments at the end of a line of code.

c-; *ZMACS Command*

If the current line contains a comment, the cursor is moved to the beginning of the text of the comment. Otherwise, a comment is started on the current line. When a comment is started, the cursor is moved to the comment column (a horizontal position where single-line comments are started by default), and beginning and ending comment characters are automatically added around the cursor.

c-m-; *ZMACS Command*

If the current line contains a comment, this keystroke removes it. Use this if you accidently type **c-;** by mistake.

The following commands are used for compiling procedures, regions, buffers, or entire files of ID code. Except for **m-X Compile File** the object code from the compiler is automatically loaded into GITA. If the ID Compiler detects a syntax error while compiling a procedure a message will be printed in the typeout window (a window which grows down over the text in the buffer), and the compilation of the procedure with the error is aborted.

c-sh-C *ZMACS Command*

If there is a region, then each procedure in the region is sent to the ID compiler. Otherwise the current procedure is sent to the ID Compiler. The object code for each procedure successfully compiled is automatically loaded into GITA.

m-X Compile Buffer *ZMACS Command*

Each procedure in the current buffer is sent to the ID Compiler. The object code for each procedure successfully compiled is automatically loaded into GITA.

m-X Compile File *ZMACS Command*

Prompts for a file to compile, and sends the entire file to the ID Compiler. Unlike the previous two commands, this command does not automatically load the object code into GITA. Instead, a .CMC file is generated which contains the object code output by the compiler. Use **m-X Load File** to load the .CMC file into GITA.

m-X Load File *ZMACS Command*

Prompts for a file to load. If the file has a .NMC or .CMC extension it is given to GITA for loading. See the function **LOAD-NMC** below.

2.3 Restrictions in ID Mode

There are two restrictions imposed by ID Mode on the way ID code is organized in a file. While the ID Compiler does not care about either of these restrictions, they must be adhered to whenever using ID Mode in order for it to work correctly.

The first restriction is that the keyword **procedure** at the beginning of each ID procedure must be flush with the left margin. That is, the **p** in **procedure** must always be in column 0. ID Mode uses this fact in order to find the beginning and end of ID procedures without having to perform lexical or syntactic analysis.

The second restriction has to do with the way in which procedures may be commented out. As far as the ID Compiler is concerned the following is a correct way of commenting out the procedure **foo**:

```
!
procedure foo(x)
  x * x
!
```

As a result of the first restriction, however, ID Mode sometimes gets confused as to

whether procedure **foo** is "inside" or "outside" of a comment. The only way to know for sure is to scan forward from the beginning of the current buffer until you get to the procedure in question. But this scanning can be very time-consuming, so instead ID Mode requires that the procedure be commented out as follows:

```
! procedure foo(x)
  x * x !
```

The idea is to make sure that the keyword **procedure** does *not* begin in column 0 if it is commented out. When uncommenting out the procedure you must remember to once again position the keyword **procedure** so that the **p** is in column 0.

Before you can execute an ID procedure you must first compile it using the ID Compiler, and then tell GITA to load the output of the ID Compiler. This can be done in several ways.

The easiest way is to load a file containing ID procedures into ZMACS and use ID Mode commands to compile one, two, or even all of the procedures in the buffer. As explained above, ID Mode will arrange for the given procedures to be sent to the ID Compiler, and will automatically load the output of the ID Compiler into GITA.

If you already have a file which contains compiled ID procedures (which you can get by using the command **m-X Compile File** in ID Mode), then you can use the ID Mode command **m-X Load File**, described above, or the function **load-nmc** to load this file into GITA.

(**load-nmc** *pathname* **&optional** *silent?*) *Function*

Tells GITA to load all of the compiled ID procedures in the file *pathname*. *pathname* should have a .CMC or .NMC extension. If no extension is provided, then a file with extension .CMC is looked for, followed by a file with extension .NMC. If *silent?* is non-nil, then the ID procedure names contained in the file will not be printed out as they are loaded.

When GITA is told to load a file containing compiled ID procedures, it creates a structure known as a POBJ (procedure object), which contains all of the necessary information to allow it to interpret the procedure. In order to make it easy to execute these procedures, GITA creates a LISP procedure corresponding to each ID procedure loaded. The LISP procedure has the same name and takes the same number of arguments as the corresponding ID procedure. When called, the LISP procedure tells GITA to interpret the ID procedure with the given arguments, and when GITA finishes executing, the LISP procedure returns the results of the ID procedure as multiple values.

For example, suppose we wish to execute the following ID procedure:

```
procedure factorial(n)
   (if n = 0 then 1
       else n * factorial(n - 1))
```

One way to do this is to enter ZMACS, tell it to set the mode to ID Mode and the Package to GITA (as described above), type in the above procedure, and use the ID Mode command **c-sh-C** to compile and load this procedure into GITA. After doing this, GITA will create an internal representation (a POBJ) of the ID procedure **factorial**, along with a LISP procedure like the following:

```
(defun factorial (n)
  (gita:run-code-block 'factorial n))
```

You can then call the LISP procedure **factorial** with an argument, which will cause GITA to interpret the ID procedure by pushing tokens around the dataflow graph stored in the POBJ, until the answers came out the bottom of the graph. The answers which come out of the graph will be the values returned from the LISP procedure.

Note that this mapping of ID procedures to LISP procedures allows you to mix both ID

and LISP. For example, if you type:

```
(+ (factorial 3) (factorial 4))
```

to a Lisp Listener, you will cause GITA to be run twice, once for each call to **factorial**, and then LISP will compute and return the sum of the results of the two ID procedures.

Furthermore, you can take the result of one ID procedure and use it as an argument to another ID procedure. For example, the following is perfectly legal:

```
(factorial (factorial 5))
```

It will simply cause GITA to be run twice, first to compute the factorial of 5, and then again to compute the factorial of 120.

You can even call ID procedures from LISP procedures (but not the other way around!). The bottom line is that you are free to treat ID procedures just like LISP procedures. As far as the user of ID WORLD is concerned, there is no difference.

If the only reason you wish to execute an ID procedure is to know the result, then the previous section has already explained how to do this. All you have to do is get to a Lisp Listener, call the LISP procedure corresponding to the ID procedure, and wait for a result to be returned.

When writing ID programs in ZMACS you will probably want to be doing this quite often to debug procedures as you write them. Whenever you write an ID procedure and wish to test it, you should use the ID Mode command **c-sh-C** to compile and load the current procedure into GTA. The next thing to do is find a place where you can invoke the ID procedure via its LISP procedure. One place to do this is in a Lisp Listener. You can select a Lisp Listener (by typing **<SELECT>-L** or **<SYSTEM>-L** on Symbolics or T.I. Lisp Machines, respectively), execute the procedure, and then return to the editor.

If you are already in the editor, however, a convenient place to quickly test an ID procedure is in the editor typeout window. The editor typeout window is a window which "grows" down over the text in the buffer, and behaves just like a Lisp Listener. On a Symbolics Lisp Machine you can get to the editor typeout window by typing the **<SUSPEND>** key. On a T.I. Lisp Machine you would use the **<BREAK>** key. Once the editor typeout window is exposed, you can type any LISP form, such as the LISP functions which execute the ID procedures you have just written and compiled. When you wish to go back to editing your programs, just type the **<ABORT>** key until the typeout window goes away.

After you have finished using ID Mode to write, compile, and test your programs, you should select the GITA Frame to execute your ID programs. You can select the GITA Frame by typing **<SELECT>-G** on a Symbolics Lisp Machine (**<SYSTEM>-G** on a T.I. Lisp Machine).

6.1 Organization of the GITA Frame

When the GITA Frame first comes up, it is divided into three sections. The top section is the *profile pane*, and is used to draw parallelism profiles (described below). About two-thirds of the way down the screen is a menu with items such as *Load* and *Execute*. Finally, the bottom portion of the screen is a Lisp Listener which you can use to execute ID procedures via their LISP functions.

There are actually two configurations which the GITA Frame can be in. The first, which is what you see when you first select the GITA Frame, is called the *profile configuration* because a large portion of the screen is reserved for the displaying of parallelism profiles.

The second configuration is called the *debugger configuration*. When the GITA Frame is in this configuration, the menu will be all the way at the top of the frame, and the rest of the frame will be the Lisp Listener. This configuration is used whenever the GITA Debugger is entered, since it more convenient to use the debugger in a large window.

6.2 The GITA Frame Menu Items

This section briefly describes each of the items in the GITA Frame menu.

Load *GITA Menu Item*

Prompts for a file to load, then calls the function **load-nmc** with that file.

Execute *GITA Menu Item*

Pops up a menu of all loaded ID procedures. Click on one to execute it. You will be prompted in the Lisp Listener for each argument required by the procedure. GITA will then execute the procedure, and print the results in the Lisp Listener.

GITA Debugger *GITA Menu Item*

Changes the configuration of the GITA Frame to the debugger configuration, and starts up the GITA Debugger. See below for a description of the GITA Debugger.

Show Profile *GITA Menu Item*

Changes the configuration of the GITA Frame to the profile configuration. Then pops up a menu of all the profiles which can be shown. Click on one to cause the profile be drawn in the profile pane. See below for a description of profiles.

6.3 Collecting and Viewing Statistics

So far the only reason for using GITA was to execute an ID procedure. But GITA can also be used to collect and view various statistics. This section describes how to tell GITA to collect these statistics.

6.4 Idealized vs. Emulated Statistics

When GITA collects statistics it does so either in *idealized mode* or in *emulation mode*. It is important to understand the difference between these two modes since the same profile can look very different depending on which mode was used to generate it.

The main difference between idealized mode and emulation mode is in how each defines the term *timestep*. In emulation mode a timestep is simply a certain fixed amount of time, such as 2 seconds. In idealized mode, however, a timestep varies in the amount of time it takes to complete. It is based on the following assumptions:

- All activities ready to fire are fired in the same timestep.
- All activities take the same amount of time to execute.
- There is zero communications overhead.
- Unbounded resources are available.

Emulation mode measures how GITA actually performed in executing some ID procedure. If the timestep were 1 second, for example, then the first sample in the ALU-operations profile (described below) gives the number of ALU operations which fired during the first second. The next sample gives the number of operations which fired during the second second, etc.

For idealized mode, however, the ALU-operations profile is interpreted as follows. The first sample indicates how many ALU operations fired when the procedure was first started. (GITA arranges for there to be one operation ready to fire at the beginning -- it drops the first token into the dataflow graph to get things started.) The second sample indicates how many ALU operations fired during the second timestep. That is, all those operations which became ready to fire as a result of operations firing in the first timestep are all fired in the second timestep. An informal definition of a timestep in idealized mode is. *Fire all and only those operations which are ready to fire at the beginning of each timestep.*

The following functions control which statistics mode GITA is in.

(emulation-mode) *Function*
Tells GITA to collect statistics in emulation mode.

(idealized-mode) *Function*
Tells GITA to collect statistics in idealized mode.

(no-stats) *Function*
Tells GITA not to collect any statistics.

6.5 Statistics in GITA

This sections describes the statistics which can be collected while an ID procedure is being executed by GITA. Note that the term *timestep* has a different meaning depending on which statistics mode GITA is in.

ALU Operations Profile *Statistic*
Collects a profile of the number of ALU operations which were fired during each timestep.

Wait-Match Profile *Statistic*

Collects a profile of the number of tokens which were in waiting-matching sections at the end of each timestep.

Invocations Profile *Statistic*

Collects a profile of the number of procedure invocations which occurred during each timestep.

Terminations Profile *Statistic*

Collects a profile of the number of procedures which terminated during each timestep.

I-Fetch Profile *Statistic*

Collects a profile of the number of I-Structure fetches were done during each timestep.

I-Store Profile *Statistic*

Collects a profile of the number of I-Structure stores were done during each timestep.

Deferred-Reads Profile *Statistic*

Collects a profile of the number of I-Structure fetches which were deferred during each timestep.

I-Structure Storage Profile *Statistic*

Collects a profile of the amount of I-Structure storage in use at the end of each timestep.

The following statistics are only collected when in emulation mode.

Queued Tokens Profile *Statistic*

Collects a profile of the number of tokens in the token queue at the end of each timestep.

Active Code-blocks Profile *Statistic*

Collects a profile of the number of code-blocks which were active at the end of each timestep.

The following statistic is only useful when in emulation mode and running on more than one physical processor.

Idle Profile *Statistic*

Collects a profile of the amount of time each PE was idle per timestep.

6.6 Viewing a Statistic

After setting the statistics mode and executing a procedure, you can view any of the statistics by clicking on *Show Profile* in the GITA menu and selecting a statistic to view. The profile will be shown in the profile pane.

The GITA debugger is in many ways like the normal LISP debugger. The main difference is that the LISP debugger allows you to look up and down a *stack* of frames, while the GITA debugger allows you to look around a *tree* of contexts.

7.1 GITA Debugger Definitions

In GITA, a *context* roughly corresponds to a *stack frame*. Whenever a procedure is invoked a *context* is created to hold its arguments and results, etc. Because a procedure can execute sub-procedures in parallel, however, GITA must maintain a tree rather than a stack of contexts.

In the description of the GITA Debugger which follows, the *current context* means the context which is currently being examined. As you move around the tree of contexts, the *current context* is changing to reflect your position in the tree. The *root context* never changes and is the one context which has no father. It corresponds to the top-level call you made when you first told GITA to execute an ID procedure. Finally, the *anchor context* is a context which usually corresponds to a context at which an error occurred, although is can be changed by using the `c-.` debugger command. This anchor context is used in order to facilitate moving up and down the tree of contexts. See the section on Movement commands, below.

7.2 Invoking the GITA Debugger

There are two ways to enter the GITA Debugger. The first has already been described -- when in the GITA Frame, you click on the menu item *GITA Debugger*. Alternatively, you can use the following function from wherever you were when you executed an ID procedure -- the editor typeout window, for example.

(gita-debugger) *Function*

Invokes the gita debugger, making the current context be the root context.

7.3 GITA Debugger Command Loop

After invoking the GITA Debugger the root context will be displayed, including the name of the procedure and its arguments, and the cursor will be to the right of a right-arrow, the GITA Debugger's prompt. Whenever the cursor is just to the right of the right-arrow, you are at top-level in the debugger. Whenever you are at this level, there are several things you can do.

- You can hit the **<ABORT>** key to exit the debugger.
- You can type one of the GITA Debugger commands, described below.
- You can type in a form to evaluate, just as if you were typing to a Lisp Listener.

Whenever you type a form to evaluate, the command loop automatically sets the global variable • to the value returned from the evaluation of the form. Thus • can be thought of as holding on to the last thing returned. Similarly, the global variable •• holds on to the second to last thing returned, and ••• holds on to the third to last thing returned. In the descriptions of debugger commands which follow, whenever a command says that it "returns" an object, the variable • can be used to refer to that object. For example, after typing the debugger command **c-A**, which returns the ISD for the arguments of the current context, you can then type **(setq foo-args •)** to make the variable **foo-args** hold on to the ISD.

Some of the commands which follow refer to a *numeric argument*. A numeric argument is a number which is typed just before a command is issued, usually specifying which of a set of *n* things should be done. To type a numeric argument, you hold done one of the control keys (either *control*, *meta*, *super*, or *hyper*) and type the digits of the number. You then type the command. For example, to type the **c-m-A** command with a numeric argument of 12, you could type **c-1 c-2 c-m-A**. The section on ID Mode explains how to type commands such as **c-m-A**.

7.4 Debugger Commands for Error Handling

When GITA detects an error during execution of a procedure a message is printed saying in which part of the machine the error occurred. Execution continues, however, until there are no more activities ready to fire. At the end of execution GITA reports the total number of errors it encountered. The following function allows you view the errors.

(show-errors) *Function*

Prints a report for each error encountered during the last GITA run. The errors are printed in reverse chronological order and include the time the error occurred and an explanation of the error.

7.5 Dealing with Error objects

When the GITA debugger is active, the following commands let you deal with errors.

c-E *GITA Debugger Command*

Prints a list of all errors encountered in the last GITA run. This command simply executes the **show-errors** function.

c-m-E *GITA Debugger Command*

Without a numeric argument, this command returns the current error object. Otherwise, the error object given by the numeric argument is returned. This is one way to get your hands on the arguments to the ALU operation which failed.

c-m-G *GITA Debugger Command*

Causes the anchor context to become the one where a particular error occurred. You must use a numeric argument to specify which error you are interested in.

7.6 Backtrace Commands

A *backtrace* is a listing of the contexts in reverse order starting from the current context and ending at the root context. There are two backtrace commands which differ only in how much detail they provide about each context.

c-B *GITA Debugger Command*

Displays a brief backtrace, showing the name of the procedure for each context.

m-B *GITA Debugger Command*

Displays a verbose backtrace, showing the name of the procedure and its arguments for each context.

7.7 Examining the Current Context

There are many commands in the GITA debugger designed to return information from the current context.

c-m-A *GITA Debugger Command*

Returns the n^{th} argument in the current context. Use a numeric argument to specify *n*.

c-A *GITA Debugger Command*

Returns the ISD used to hold the arguments for the current context.

c-m-V *GITA Debugger Command*

Returns the n^{th} value being returned from the current context. Use a numeric argument to specify *n*.

c-V *GITA Debugger Command*

Returns the ISD used to store the return values of the current context.

c-m-L *GITA Debugger Command*

Returns the n^{th} local variable (token) from the current context. Use a numeric argument to specify *n*.

c-m-F *GITA Debugger Command*

Return the procedure object from the current context.

c-m-C *GITA Debugger Command*

Returns the current context object, an object of type **dbg-context**. Note that this is not the same as the actual context object--a **dbg-context** only *refers* to the actual context object through its PE and INDEX slots. The GITA debugger uses this object to store the replies from remote servers so that it won't have to ask again.

7.8 Movement Commands

Moving around the tree of invocations in the GITA debugger is not quite as easy as moving up and down the stack in the LISP debugger. A context will have at most one father (the caller), but may have several sons (each corresponding to a procedure invocation which has not yet terminated).

Moving "up" the tree of invocations is straightforward, the current context gets set to its father. Moving "down" the tree of invocations, however, requires that a branch be selected from among its sons. To make moving down the invocation tree easier, the GITA debugger

will sometimes pick the branch which is considers the "obvious" choice. Taking the "obvious" choice is done by using the **c-N** command without any numeric argument. If the GITA debugger cannot determine an "obvious" choice, then a numeric argument specifying which branch to go down must be given. The GITA debugger determines the "obvious" choice as follows:

- If the current context has no sons, then there is no way to go down, and thus no "obvious choice".
- If the current context has only one son, then the "obvious" (and only) choice is to go down to the son.
- If the current context has more than one son, then the "obvious" choice is the son which is closest to the anchor context. If none of the sons are in the path to the anchor context, then there is no "obvious" choice.

By defaulting the "obvious" choice, you should be able to stick with using **c-P** and **c-N** and never have to specify a branch unless you want to go off and inspect a context at some other part of the tree.

c-P *GITA Debugger Command*

Goes up one context towards the caller. With a numeric argument, goes up that many contexts towards the caller.

m-P *GITA Debugger Command*

Like **c-P** except that detailed information about the target context is displayed.

m-< *GITA Debugger Command*

Goes to the root context (the oldest in the invocation tree).

m-> *GITA Debugger Command*

Goes to the anchor context.

c-N *GITA Debugger Command*

With no numeric argument, goes down the "obvious" branch of the invocation tree. Otherwise goes down the n^{th} branch, where n is specified by the numeric argument.

m-N *GITA Debugger Command*

Like **c-N** except that detailed information about the target context is displayed.

c-. *GITA Debugger Command*

Sets the anchor context to the current context.

7.9 Searching

The following commands search for a context with a procedure whose name contains a given substring. The contexts are searched starting from the father of the current context toward the root context. This is useful for quickly jumping to some context shown in a backtrace. For example, if you type **c-B** and the backtrace is:

```
FOO [1] <- BAR [301] <- BAZ[8] <- QUUX [503]
```

Then searching for "BA" will make the current context be **BAZ [8]**, and searching for "F" will make the current context be **FOO [1]**.

c-S *GITA Debugger Command*

Prompts for a substring and searches up from the current context for one whose procedure name contains the substring. If it finds one, then that context becomes the current one.

m-S *GITA Debugger Command*

Like **c-S** except that detailed information about the target context is displayed.

7.10 Other Debugger Commands

An invocation tree shows the full tree of invocations still active at the time the last GITA run finished executing. The sons of a context are all indented to the same column underneath the father.

c-T *GITA Debugger Command*

Shows an invocation tree of all known contexts.

Use these next two commands to clear the screen, in one of two ways.

c-L *GITA Debugger Command*

Clears the screen and displays the current error message along with the procedure name and arguments of the current context.

m-L *GITA Debugger Command*

Clears the screen and displays detailed information about the current context, including the procedure name, its arguments, results, and locals (tokens).

These next few commands let you look inside of various objects.

c-m-D *GITA Debugger Command*

Describes the last thing printed out. This keystroke is equivalent to typing **(DESCRIBE *)**.

c-I *GITA Debugger Command*

Pretty-prints the last thing printed out. This command is especially useful for viewing ISDs. When an ISD is pretty-printed the contents of the I-Structure it references is printed out.

c-m-I *GITA Debugger Command*

Typing this command toggles the ISD pretty-print flag. When the flag is set all ISDs are automatically pretty-printed.

Finally, you can get online help by typing the **<HELP>** key.

<HELP> *GITA Debugger Command*

Prints a concise description of each of the debugger commands.

GITA can be made to run on multiple machines. One of the design goals of ID WORLD, however, was that the user should not be concerned with the number of machines which are in use. Accordingly, ID Mode, the GITA Debugger, and the statistics have all been designed to be used in exactly the same way no matter how many machines are participating in the execution of an ID procedure.

8.1 Setting up Multiple Machines

Note that this section is likely to change in the near future. The interface to multiple machines at the moment is neither very robust nor general, and is being redesigned. Therefore, this section will provide only a quick introduction to some of the functions which are currently used to run experiments on multiple machines.

default-processors
Variable
This variable contains a list of the machines, including the local machine, which may be used by the local machine to execute ID procedures.

(select-firstn-processors *n*)
Function
Sets the variable ***default-processors*** to the names of the first *n* processors in the MEF.

(select-processors &rest *machines*)
Function
Sets the variable ***default-processors*** to the list of machines given as arguments to this function.

(initialize-gita-servers)
Function
Starts up a GITA-Server processes on each of the machines in ***default-processors***.

(reset-gita-servers)
Function
Makes sure that the GITA-Server process is still running on each of the machines in ***default-processors***. Will reestablish a connection to any of the machines if it has failed.

(initialize-network) *Function*

Causes each of the machines in ***default-processors*** to derive the connectivity of the Circuit Switch network. This must be done once before GITA can execute ID procedures on all the machines.

(show-netstate) *Function*

Prints out a textual representation of the connectivity of the Circuit Switch for each of the machines in ***default-processors***.

(draw-netstate) *Function*

Draws a pictorial representation of the connectivity of the Circuit Switch for each of the machines in ***default-processors***.

(pes *n*) *Function*

Selects the first *n* processors in ***default-processors*** to participate in the next execution of an ID procedure. Setting *n* to 1 will force the local machine to be the only one involved in the execution.

8.2 Using the GITA Server

The following functions may be used to force some or all of the machines in ***default-processors*** to perform some action.

(all *form*) *Macro*

Causes all processors, including the local machine, to evaluate *form*. The results are discarded. Does not wait for the servers to finish evaluating *form* before returning.

(all-eval *form*) *Macro*

Causes all processors, including the local machine, to evaluate *form*. The results from each machine are collected and returned.

(others *form*) *Macro*

Causes all processors except the local machine to evaluate *form*. The results are discarded. Does not wait for the servers to finish evaluating *form* before returning.

(others-eval *form*) *Macro*

Causes all processors except the local machine, to evaluate *form*. The results from each machine are collected and returned.

(execute-on *peform*) *Macro*

Causes *form* to be evaluated on the pe^{th} machine in ***default-processors***. The results are discarded. Does not wait for the server to finish executing *form* before returning.

(eval-on *pe form*)

Macro

Causes *form* to be evaluated on the pe^{th} machine in ***default-processors***. The results are collected and returned.

(load-on *pe file*)

Macro

Causes *file* to be loaded on the pe^{th} machine in ***default-processors***.

(all-load *file*)

Macro

Causes all processors, including the local machine, to load *file*.

1. Arvind, D. E. Culler, R. A. Iannucci, V. Kathail, K. Pingali and R. E. Thomas. The Tagged Token Dataflow Architecture. Laboratory for Computer Science, MIT, Cambridge, MA, July, 1983. (Prepared for MIT Subject 6.83s)

2. Arvind, M. L. Dertouzos and R. A. Iannucci. A Multiprocessor Emulation Facility. Tech. Rep. TR-302, Laboratory for Computer Science, MIT, Cambridge, MA, October, 1983.

3. Arvind, and K. P. Gostelow. The U-interpreter. *Computer 15*, 2 (February 1982), 42-49.

4. Heller, Steve and Ken Traub. ID Compiler User's Manual. Tech. Rep. TR-248, Computation Structures Group, Laboratory for Computer Science, MIT, Cambridge, MA, February, 1986.

5. Soley, Richard M. Generic Software for the Emulation of Multiprocessor Architectures. Master Th., MIT Laboratory For Computer Science, June, 1985.