LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Functional Languages and Architectures Progress Report 1984-85

MIT/LCS/TR-346

September 1985

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

FUNCTIONAL LANGUAGES AND ARCHITECTURES

Academic Staff

Arvind, Group Leader
R. Nikhil

Research Staff

R. Iannucci, Manager - Hardware Development
J. Pinkerton

Graduate Students

M. Beckerle
S. Brobst
A. Chien
D. Culler
S. Heller
R. Iannucci
V. Kathail
G. Maa
G. Papadopoulos
K. Pingali
R. Soley
K. Traub
B. Vafa

Undergraduate Students

G. Bromley
D. Clarke
E. Hao
F. Herrmann
R. Indech
R. Katz
C. Lee
D. Morais
G. Ng
F. Park
J. Sheffield
J. K. Soon
Y. M. Tan
R. Wei
J. Weisz
C. Wong
S. Younis

Support Staff

S.M. Hardy
N. F. Tarbet

Visitors

E. Hagersten (Ericsson/Ellemtel,Sweden)
N. Skoglund (Ericsson/Ellemtel, Sweden)
B. Blaner (IBM)
D. Lowther (IBM)
M. Mack (IBM)

Functional Languages and Architectures

1. INTRODUCTION

The Functional Languages and Architectures group is pursuing two interrelated projects, namely, the Tagged-Token Dataflow Machine and the Multiprocessor Emulation Facility (MEF). The goal of the Tagged-Token Dataflow project is to demonstrate the feasibility of general purpose parallel machines by simulation and emulation. The goal of the MEF project is to construct a "sandbox" to facilitate research and development in parallel architectures and languages. Following this introduction, the report is divided into two major sections describing the ongoing projects.

Our cooperation with IBM on both these projects has increased significantly over the past year. Several joint meetings to discuss the technical details of the two projects were held, and as a consequence of these meetings mid-course maneuvers were required. For example, we have decided to switch to non-IBM technology for the packet communication network for the MEF. On the dataflow project, the discussions resulted in identification of two sub-goals which may be achieved by somewhat different strategies. These sub-goals were (1) the demonstration of the scalability of the dataflow machine, and (2) the investigation of a dataflow machine which will require the same amount of hardware as a high performance sequential computer for the same level of performance. Our immediate attention has been focused on the first sub-goal, and consequently changes in the architecture or the compiler which are required only for the demonstration of the second sub-goal, have been deferred.

A lively and productive IBM-MIT workshop was organized last April, in Essex, CT, to review multi-processor research at the two institutions. We expect such interactions and cooperation with IBM will continue at an even greater pace in the coming year.

1.1. Dataflow Project Overview

Since the success of parallel machines will depend on the effective programmability and efficient utilization of resources, the dataflow project has been deeply concerned with high level language support and resource management issues. We have been experimenting with Id, a functional language, for a number of years, and this year have finally produced a documented version of the Id-to-dataflow graph compiler, which can be used as a service program by us and other researchers. We have now entered what may be called the second phase of development for both the language and the compiler. On the language side, some

ground work has been done to make polymorphic type-checking and reference count garbage collection possible. A proper semantic view of I-structures, the main data structure in Id, also seems to be emerging. Fortunately, this new view of I-structures, even though fundamentally very important and novel, should have minimal impact on the compiler. The compiler is being rewritten to improve its internal modularity to make possible easy implementation of both anticipated and unanticipated changes in Id. This new phase of language and compiler development is relying, much more heavily than in the past, on the solid theoretical work that has been done at MIT and elsewhere in the area of functional languages.

We think that programmable and scalable machines should be such that a change in machine configuration, *e.g.*, changing the number of processors or memory modules, interconnection topology, does not require the programmer either to rewrite or recompile application programs. Thus, sophisticated run- time resource managers are essential for the dataflow machine. We have made significant progress in efficiently executing loop programs; however, further progress in this area is crucially tied to the large scale simulation and emulation experiments. Our simulator, which runs on IBM machines, is in the final stages of debugging and documentation, and should allow us and IBM researchers to experiment with resource management policies in the near future. Large scale emulation experiments will have to wait until MEF is fully functional.

1.2. MEF Overview

The past year has been one of continual upheaval for the Multiprocessor Emulation Facility (MEF). We have endured one change of CPU, a major change in personnel, several changes of technology, several changes of design strategy, and, most importantly, two changes of name. Despite these changes, or perhaps because of them, we have moved forward and will soon see first fruits. Spurred on by the arrival of funds from DARPA, the Hardware Laboratory was completed and is now in use.

New CPU: At the beginning of this reporting period, we had planned to construct the Emulation Facility out of Symbolics 3670 processors. We have subsequently switched to TI Explorers. The rationale for the change was based partially on price, but predominantly on the open architecture of the Explorer coupled with the fine cooperation we have received and expect to receive from Texas Instruments. Currently, we have 8 Symbolics Lisp machines connected by Ethernet, and 2 preproduction models of TI Explorers.

Personnel: During the previous reporting period, we had proposed a joint study with IBM whereby they would provide design engineers to work at LCS on the packet switch. The first three engineers have been with us for a year now and have made

significant contributions in the areas of high speed serial link protocol development / verification, central switching mechanism design, creation of a library of very high performance Programmable Logic Array cells for custom VLSI, partitioning, packaging, and analysis for the packet switch. In addition, they have researched the existence of standard parts within IBM which will greatly simplify the design of this switch.

Technology: As an adjunct benefit of the Joint Study, IBM was to provide design tools and manufacturing for the gate array logic used in the packet switch. Due to confidentiality constraints, we have declined this offer of technology and have instead negotiated for chip fabrication through LSI Logic Corporation.

Design Strategies: Prior to the switchover to TI machines, we were designing two different network cards for the 3670: the first was an adaptation of the BBN Butterfly's circuit switch. The second was a packet switch of our own design (both reported previously). The TI machine is smaller than the Symbolics machine in many respects - most importantly, the circuit cards are smaller, and the potential for expansion (card slots, power) is much more limiting. Consequently, we have re-partitioned the design into three circuit cards: (1) *a packet switching network adapter*, (2) a *circuit switching network adapter*, and (3) a *NuBus channel adapter*. The channel adapter, or NuCA, provides direct-memory access between the NuBus and up to three network adapters which may be circuit switches, packet switches, or a combination.

Name Changes: In that the design effort for the MEF is directed at interconnecting an array of Lisp machines and that these machines have been named after trees, *e.g.*, Cherry, Mahogany, Live-Oak, Netleaf-Hackberry, etc., a proposal was put before the group to name ourselves *Project Tanglewood*. The Boston Symphony, however, refused to permit our use of the name. We had temporarily changed names, only to have to revert back to the old (and uninspiring) "MEF".

2. TAGGED-TOKEN DATAFLOW PROJECT

2.1. Language Work

Our work on languages this last year, under the guidance of Professor Rishiyur Nikhil, has been aimed mostly towards improving the expressive power of Id, improving the reliability of Id programs and generating better code for the execution of Id programs on the tagged-token dataflow machine. To this end we are experimenting with the introduction of types and type-checking into Id, but with a firm commitment not to sacrifice the expressive power of the language. In this

endeavor we are inspired by the work of Milner on the type system of the programming language ML [3, 1]; however, we expect our type system to be much more convenient to use without sacrificing type-safety.

2.1.1 Towards An Unobtrusive but Useful Type System

We consider a language to be *type-safe* if it is not possible to misinterpret the underlying representation of an "abstract" object, *i.e.*, if only "proper" operations may be applied to it. The issue of type-safety is orthogonal to the question of compile-time vs. run-time type-checking which specifies at what time a violation of type-safety is caught. Thus Lisp, Apl, Ada, CLU and Pascal[1] are all type-safe, whereas C is not. However, Lisp and Apl are run-time type-checked, whereas Ada, CLU and Pascal are compile-time type-checked. Id is a type-safe language; however, the current version of Id is weakly typed in the Lisp sense, *i.e.* there are a few primitive types (numbers, booleans, strings and arrays), no user-defined types, and all type-checking is deferred until run-time. This lack of typing information has posed some serious problems in generating efficient code for procedure calls and data structures in Id. Some of these problems have been resolved in the current version of Id in a rather *ad hoc* way. For example, function calls often have to be annotated with information about the number of results returned. In some other cases, such as proper code generation for *reference count* garbage collection we have not been able to come up with an acceptable solution. The primary difficulty arises from the fact that we are unable to distinguish at compile time between identifiers, *i.e.*, variables, that represent arrays and those which don't.

However, the benefits of compile-time type-checking often come at the expense of certain unreasonable restrictions on what the programmer may say. For example, in Pascal one cannot write generic routines to manipulate stacks; one must write separate routines to manipulate integer-stacks, boolean-stacks, etc. To a large extent this is attributable to a lack of sufficient polymorphism in the type system, *i.e.*, most type systems do not allow functions to be parameterized with respect to types. Another criticism of languages with compile-time type-checking is that the programmer is required usually to declare explicitly the type of every identifier that is used. While this is generally a good idea from the point of view of readability and long-term maintenance of code, it can often be a hindrance in interactive program development.

The type system of ML is one attempt to address these issues. Its type system has sufficient polymorphism to allow writing programs as succinctly as, say, in Lisp. Further, it has a type-inference system built into the type-checker that allows the programmer in general to omit declarations of identifiers and expressions: the system fills them in.

---

[1] With a few well-known holes.

While ML's type system is a step in the right direction, it is still not unobtrusive. Most seriously, there are some limitations in its expressive power that result in the rejection of some quite reasonable programs. Further, because the type-inference relies on inter-procedural type information, a straight-forward implementation of the type-checker[2] ends up doing unacceptable amounts of re-analysis on every change made in interactive program development. We are currently experimenting with solutions to these problems.

In the context of Id a number of changes to the language must be made before we can explore the limitations of Milner's system. For example, arrays in Id can be heterogeneous (*i.e.*, x[0] could be a number and x[1] an array) and so may be treated as n-slot CONS cells. Currently, the use of heterogeneous arrays in Id is necessary generally because of the lack of a record type. It is practically impossible to generate efficient code for reference counting under such conditions. Thus, for better type deductions in Id, we are exploring the introduction of two more primitive type constructors: records and tagged unions.

Our type-checker will also give us a good handle on optimizing code generation for storage management in the tagged-token dataflow machine. As will be explained in the next section, we are experimenting with reference counting techniques, and the code necessary for this is centered around two constructs in Id dataflow graphs that are very common: forks and function calls/returns. Without information about which edges here carry structure pointers and which carry scalars, it is necessary to generate a lot of "conditional" code to maintain reference counts at *every* such construct. The type information inferred by our type-checker will allow us to cut down significantly on this overhead by identifying just those places where such code is really necessary.

2.1.2 Garbage Collection

Like some other modern languages, Id requires dynamic storage management for its structures, *i.e.*, structures are allocated in a special memory called "I-structure memory" at run-time and reclaimed when they are referred to no longer; this activity does not follow a stack discipline. The problem of detecting when a structure no longer is referred to generally requires reference-count or mark-sweep garbage collection.

We think that a mark-sweep technique is not very suitable for a multiprocessor architecture because it appears to be inherently sequential as opposed to reference counting which is inherently parallel [4], moreover, our machine model is not well suited to identifying the roots of all reachable objects: pointers reside in tokens

[2] As in existing ML implementations.

which are continuously flowing through different memories (waiting-matching, constant memory), queues and the network. The classic disadvantage of reference count methods, of course, is that they are unable to handle circular structures. Fortunately, Id structures cannot be circular[3]. The two variations on this theme that we are exploring are: 1) program transformation to optimize reference counting and 2) reference Weights. In addition to Michael Beckerle, Vinod Kathail, and Keshav Pingali at MIT, K. Ekanadham and M. Kumar of IBM have also participated in this work.

I.Optimizing Reference Counting

There is an aspect of the reference counting method that does not fit well into the highly pipelined and asynchronous execution model of the tagged token dataflow machine. After an "Increment Reference Count" instruction (Inc-RC) sends a token to structure memory asking for the the count to be incremented, it must *wait* for an acknowledgement before it can enable any successor instructions. The reason is that a successor instruction may be a "Decrement Reference Count" instruction (Dec-RC) which also sends a token to structure memory, asking for the count to be decremented. If the Inc-RC instruction does not wait for an acknowledgement before enabling successor instructions, these tokens may arrive at structure memory out of order, thus decrementing the count to zero spuriously, resulting in the structure being freed prematurely. Thus each Inc-RC instruction becomes an (undesirable) synchronizing point.

The Dec-RC instruction, on the other hand, does not have to wait for an acknowledgement from structure memory. This means that we run the risk of holding on to a structure longer than is absolutely necessary; but this is still safe, provided we are not critically short of storage.

We are working on program transformations to decrease the number of places in Id code where a synchronizing Inc-RC needs to be done, at the expense of increasing the number of places where a Dec-RC is done. Normally, an Inc-RC must be done at every fork in the dataflow graph (where tokens get duplicated). Our aim is to be able to do a single "Increment Reference Count by N" instruction (Inc-RC-n) at the beginning of a larger code block, say a function or a loop, by analysing the graph of the code block to see how many copies of a token might ever be made. Of course, because of conditionals, this in general will be an over-estimate, and so it may be necessary to compensate for this by inserting "Decrement Reference Count by N" (Dec-RC-n) instructions into some branches of conditionals.

II.Reference Weights

---

[3]By convention only; the compiler does not enforce this.

A possible alternative to the reference count method is that of reference weights. When a structure is allocated, we store a large "reference weight" *w* in the structure and in the returned pointer. When the pointer is duplicated, the weight is simply split between the two resulting pointers. When a pointer containing weight *w1* is discarded, we send a message to structure memory to decrement the weight stored there in the object by *w1*. The invariant is that every pointer to a structure carries weight of at least 1, and the the sum of the weights in all extant pointers to an object is equal to the weight stored in the object itself. When the weight stored in the structure drops to zero, therefore, there cannot be any remaining pointers to it, and so the structure may be deallocated.

The advantage of this scheme is that all the operations are purely local: there are no synchronizing points. The disadvantage is that when a pointer carrying weight equal to 1 needs to be duplicated, some extraordinary action needs to be taken. A solution to this problem now is to allocate a new "indirection" structure with full weight *w*, containing the original weight 1 pointer. The pointer to this new structure now has full weight and may thus be duplicated. Of course, when this pointer is later traversed to access the structure, it must use the indirection to forward the request to the original structure, and this constitutes a source of overhead. However, experimental figures for reference counts for sequential implementations of Lisp and other functional languages seem to indicate that very few (less than 5%) of all structures need reference counts (and hence reference weights) of more than 1, and thus such indirections would be necessary but rarely. If this is true also in dataflow execution of Id, the indirection overhead would be negligible enough to be acceptable. We plan to perform simulation experiments to test this hypothesis.

2.2. Compiler Progress

Steven Heller and Ken Traub have debugged and ported Vinod Kathail's original Id-to-Graph Compiler to Lisp Machines. Now simply called the Id Compiler (Version 1), it has been used successfully to compile several hundred programs since its release in January 1985. The compiler output has been verified both by execution on the simulator and emulator, and by detailed hand inspection. The largest program that this compiler has processed is Simple, a 1200-line hydrodynamics program from Lawrence Livermore Laboratory. Simple consists of 19 individual Id procedures, which in turn compile into 55 code blocks representing the procedures and the loops nested therein. The total size of the compiled code is about 86,000 bytes of code for the Tagged Token Dataflow Architecture (TTDA), representing over 10,000 dataflow operators. The current release of the compiler implements *bounded loop schema* which was developed this year and is discussed in Section 2.4

Coincident with the release of Version 1 of the Id Compiler was the publishing of the *ID Compiler Users's Manual*, which contained the first definitive description of Id

syntax, as well as instructions for using the compiler. A revised and expanded version of the user's manual has been published as CSG Memo 248.

The past year we have been increasingly aware of the need to adapt the Id Compiler to changing requirements. We are now interested in particular in changing the basic schema to incorporate new ideas about resource management, in changing the object code generation to reflect the evolution of the TTDA, in optimizing code through common subexpression elimination and code-block merging, in enhancing the language with a deductive type-checking facility to permit efficient management of structures, and in performing a host of other compiler-related experiments. To meet these requirements, we have begun work on Version 2 of the Id Compiler. The major design goals of Version 2 are:

- Provide a well-documented modular structure that allows the easy addition and removal of compiler phases, such as type-checking or common subexpression elimination. In this way, experimental compilation techniques can be introduced at each step of the compilation procedure.

- Make the major transformation phases of the compiler (parsing, abstract graph generation, machine code graph generation, and assembly) as specification- driven as possible. This allows easy modifications to syntax, schema, and machine architecture without the need for detailed knowledge of data structures and algorithms internal to the compiler.

We are now designing Version 2 and plan to complete it by June 1986.

2.3. Simulator Progress

The simulation facility has evolved into a solid testbed for the TTDA. The simulation software has stabilized and we are proceeding with experiments aimed at enhancing our understanding of the architectural tradeoffs to be made in the construction of a dataflow multiprocessor. Experiments conducted by Stephen Brobst, David Culler, and Behrouz Vafa have helped us to identify conditions under which the machine will deadlock, bottlenecks in the execution pipeline of the processing elements, and the need for controlling the amount of parallelism exploited during program execution. Programs that we are currently running on the simulation facility include matrix manipulations, the *Livermore Kernels*, and parts of the *Simple Code* developed at Lawrence Livermore National Laboratory. The resource requirements within the TTDA necessary for executing the partial-differential equation simulations implemented by the Simple Code are a subject of current investigation.

In support of these simulation experiments, Richard Wei with the help of others has

built a sophisticated user interface on top of the software implementation of the TTDA. This interface allows complete configuration of a dataflow multiprocessor, including the number of processing elements, relative speeds and technology of the hardware stages in a PE, and resource management policies. Simulation experiments consist of both a machine configuration and a specification of the source program to be executed. These experiments can be run either in a remote batch mode via a simulation server on its own virtual machine or in a user's own virtual machine to facilitate interactive resource management decisions, as well as debugging.

In addition, a suite of data collection and analysis tools have been developed by Robert Katz as a means for evaluating the performance of various implementations of our dataflow multiprocessor. A database system specifically built for the simulation facility was put in place early last fall. The data collected includes performance profile information, as well as a summary of all relevant benchmark characteristics of an experimental run. A number of statistic-calculating packages are available for analyzing this performance data. Cora Wong has developed a facility for graphic output of experimental data, and it is one of the most useful tools we have for providing insight into the dynamic execution characteristics of the TTDA. Most recently, an interface between Hewlett-Packard plotters and the IBM 4341 has been developed to facilitate high-quality graphic representation of performance data.

Steve Brobst, David Culler and Gino Maa are currently in the midst of a software engineering effort to tighten up the interfaces between simulation modules, as well as to strengthen the abstractions between the architectural, scheduling, and data collection components of the simulator. This will be the last stage of our software development effort before we release the simulation facility software to the IBM Research Center at Yorktown Heights. In the near future we will be upgrading the simulation hardware resources to an IBM 4381 in an attempt to increase the performance of simulation experiments.

Most of our experimentation to date has been directed at verifying the simulator and the code produced by the compiler. Some of the benchmarks we employed are mentioned above. A variety of experiments were conducted to verify claims concerning processor performance. For machine configurations of up to 16 PEs (the largest we are currently able to simulate on the IBM 4341), good scalability was demonstrated. We also observed that independent threads of computation interlace nicely within the processor pipeline, as expected. Until the PE becomes fully utilized, additional threads of computation can be processed with no increase in processing time. A number of experiments were conducted to verify claims made by Culler on the resource requirements of loops. We observed that (1) the resource requirements of loops do increase essentially linearly with the amount of unfolding,

(2) for simple inner loops pipeline constraints serve to restrict the unfolding, (3) for loops that involve a significant amount of parallel computation in each iteration, the unfolding is dramatic and independent of the amount of parallelism the machine can exploit, (4) loops can be constrained using the techniques presented in [2] to achieve substantial reductions in resource usage without performance degradation. For large programs, it appears neccessary to constrain program unfolding to keep the resource requirements within reason. We are currently modifying the compiler and the simulator to support new graph schemata which allow parameterized. This unfolding will allow us to study strategies for controlling dataflow programs in the large.

2.4. Resource Management

Our recent work in resource management focuses on certain differences between the U-interpreter and the TTDA. The model of computation embodied in the U-interpreter is extremely powerful; it places minimal constraints on the execution order, and thereby allows maximal parallelism. It imposes no resource constraints whatsoever; an unbounded number of activities may be performed concurrently, unbounded queuing of tokens on the arcs is permitted, and activity names may grow, *i.e.*, if a *greedy* schedule is followed, programs unfold in accordance with whatever parallelism is present. The U-interpreter also provides a convenient framework for reasoning about dataflow programs and for proving properties of programs. In part, the ease with which the U-interpreter lends itself to formal analysis stems from its rather idealized view of computational resources. The U-interpreter also provides a convenient framework for reasoning about dataflow programs and for proving properties of programs. The TTDA captures the essential execution mechanism of the U-interpreter, allowing programs to unfold in accordance with whatever parallelism is present in the program, but imposes rather strict resource constraints; tokens must reside in the waiting-matching store, activity names are represented by fixed-size tags, etc. David Culler and Behrouz Vafa have attempted to overcome the differences between the model and the machine in regard to resources through a mixture of techniques: program analysis to deduce the resource requirements of sizable portions of dataflow programs, dynamic resource management to distribute work over the machine without overcommitting individual resources, and program transformation to make programs more suitable for execution on the TTDA.

Last year we reported on a dynamic resource management system, developed as part of the simulation and emulation efforts. That system provides a means of distributing work and data over the machine (using fairly straight-forward load-leveling techniques) and takes care of allocating and deallocating all explicitly managed resources. Deallocation of resources requires embellishing the program graphs to detect when certain resources are no longer in use. Basically, this requires augmenting the graph for each code-block with arcs so that a certain node is guaranteed to be the last activity in the code-block.

The hardware allocates implicitly certain resources (notably, waiting-matching storage) on demand. When a code-block is invoked, a certain collection of processing elements are designated to perform the invocation. These processing elements must provide storage for waiting tokens generated in the course of the invocation. The load on the waiting-matching store is particularily crucial; if the matching store fills up and the overflow is used, performance degrades considerably; if the overflow store fills up, the program deadlocks. One of the motivations for developing the simulation and emulation was to determine how large the waiting-matching store would have to be for large programs to run efficiently on the machine. It became clear that the resource management system should not ignore the load on the waiting-matching stores when distributing work over the machine. However, in order to account for the load placed on the waiting-matching, the token storage requirements of code-blocks must be determined in advance. As the token storage requirement depends on the order on execution, it was necessary to deduce from the program graph the worst-case storage requirement under all possible legal execution orders. The basic idea is to model the space of legal configurations as a linear program and to solve for the maximum number of tokens on the arcs. For acyclic blocks without conditionals, this can be solved efficiently. For blocks with conditionals, tight bounds are NP-complete, but approximate bounds are easily computed with branch-and-bound techniques.

Loops present another class of problems. One of the virtues of the U-interpreter and the TTDA is that loops unfold automatically, exposing what parallelism is present. Iterations are destinguished by an *iteration number* carried as part of the activity name or tag. However, this automatic unfolding introduces cerain complications. Since tags are of fixed size in the TTDA the iteration number field may overflow. Moreover, the resource requirements of an activation (in particular, the token-storage requirements) increase linearly with the number of concurrent iterations. To overcome these difficulties, we desired a mechanism for controlling the extent of unfolding and automatically recycling iteration numbers. The key result is that a loop has bounded unfolding if and only if the graph forms a single connected component. Working from this, David Culler developed techniques for augmenting loop code-blocks so that the degree of unfolding is controlled by a single run-time parameter.

Finally, we considered measures for constraining the automatic unfolding of programs in order to reduce the resource requirements. A greedy scheduling strategy tends toward breadth-first unfolding, which offers maximal parallelism, but has large resource requirements. A depth-first strategy reduces the resource requirement, but also reduces the amount of parallelism. Of the variety of techniques developed for controlling the evaluation strategy, the most promising relies on the resource manager to queue invocation requests when the machine is saturated with work.

2.5. Reduction Systems and Combinators

During the last year, we have continued our investigation of reduction systems and combinators. In particular, Keshav Pingali has been examining the relationship between *supercombinators* and dataflow. A supercombinator can be thought of as a closed λ-abstraction whose definition is of the form $\lambda x.\lambda y....\lambda w.e$ where e is a λ-expression without any nested λ-abstractions. The number of leading λ's in the definition of a supercombinator is called the *arity* of the supercombinator. There are many techniques, such as *λ-lifting* and *mfe-abstraction*, for converting conventional functional language programs into supercombinatory form, *i.e.*, into a set of supercombinator definitions and an expression involving supercombinator applications.

To evaluate supercombinator programs, the definitions of the supercombinators can be used as rewrite rules - any application of a supercombinator to as many arguments as its arity can be replaced by the body of the supercombinator with the arguments substituted in. This is a straightforward reduction implementation of supercombinators. It was noticed by Johnsson that is possible to do better than this. Each supercombinator definition can be compiled into code for a stack machine (or for that matter any sequential machine), so that many intermediate steps of the reduction can be avoided altogether. He calls this process "short-circuiting" graph reduction.

We have found that the essence of short-circuiting the reduction of a supercombinator application is captured by converting the body of the supercombinator into *continuation-passing* style. The code generated for the stack machine is an implementation of this transformed program. This view of short-circuiting has two advantages. First, it relates the work done by Johnsson [JOH83] to the work done by Sussman and Steele, Wand and others on compiling Scheme-like languages [STE75; WAN78]. Second, it is easy to extend this idea to parallel implementations of functional languages. We are currently trying to understand dataflow implementations in this light. If successful, this will permit us to relate dataflow to reduction.

3. MULTIPROCESSOR EMULATION FACILITY PROJECT

3.1. The Hardware Laboratory

During the previous reporting period, Robert Iannucci, James Pinkerton, Gregory Papadopoulos and others put together the plan for a new MEF Hardware Laboratory (Tanglewood Design Note 4) This year we saw the dream come to life in the form of a 600 sq. ft. room on the second floor which now provides

- Two AED 767 color VLSI layout stations running CAESAR, HPEDIT, and (soon) MAGIC, interfaced to a Hewlett-Packard 8 pen plotter.

- Two Apollo design workstations for schematic capture, simulation, timing verification, and component placement. We are also adding a 500 MB file server to our Apollo ring network.

- A drafting table (when all the fancy electronic stuff fails).

- Four fully instrumented logic design / debug stations, each with an oscilloscope, a logic analyzer, function generator, counter, digital multimeter, programmable power supplies, an instrumentation computer, and a full set of hand tools. In addition, we have a high performance digitizer and a time domain reflectometer.

- A stockroom.

- A VLSI microprobe station.

We also recently learned that our request for a grant of $200,000 worth of additional Apollo workstation hardware has been approved; we will take delivery on the equipment before the end of the summer.

3.2. Packet Switch

Bart Blaner, Michael Mack, and David Lowther joined the MEF team from IBM during the summer of 1984. During the last year, they have made noteworthy progress on the development of the packet switch. Major accomplishments include

- Development of a protocol for the high speed serial links of the switch, which is an improvement over the echo-acknowledge protocol previously reported. Bart's contributions here are very significant; he will be continuing his work on protocol verification over the next few months.

- Development of the central switching mechanism for the packet switch card. Michael's high-level and detailed designs are sufficiently far along that we expect to release a chip description to the manufacturer by the end of the summer. The result will be an 8-in, 8-out, 4-bit-wide crossbar on a single 84-pin chip. Performance will far exceed the requirements of the packet switch.

- Partitioning of the card's function down to the chip level. The

partitioning is such that the high-speed serial logic is entirely modular and is separate from the central switch. This makes it conceivable to design both smaller and larger packet switches which are protocol compatible with the existing design; such redesigns would be nearly trivial. This opens the possibility of using the serial link as a building block for many other applications *e.g.* dataflow processor cards, backbone local area networks. We expect to have a working link in prototype form by early Fall 1985, with the cMOS version to follow in the second quarter of 1986.

- Selection of our target technology. We are currently planning on using LSI Logic's LL7600 series and LL5220 series for the high speed serial logic and the central switch, respectively. We will also be using several components available through IBM to make the design task simpler.

- Design of a set of very high performance PLA cells for custom VLSI. We have just received our first test chip from MOSIS. Measurements will be taken within a month.

Performance of the switch is dominated by two major factors: the speed of the serial links (target is 32 Mbits/sec) and the latency of allocating a path through each switch upon the arrival of the head of a message. The switch is optimized for high-traffic message-passing communication where total throughput is more important than per-message latency. We are currently re-investigating design enhancements that will noticeably improve our ability also to support a *remote memory reference* model of communication *i.e.*, shorter messages, with round-trip delay more important than total message throughput.

During the next year, the packet switch team will continue to grow. It is likely that we will have our first prototype card by summer of '86; we expect to be running packet switch / NuCA test shortly thereafter. Between now and then, a significant amount of effort will be required to implement the necessary NuCA microcode, Explorer microcode, and LISP code to make efficient data transfer accessible from LISP.

3.3. Circuit Switch and NuCA

We have adapted the BBN Butterfly circuit switching network as an interim communications medium until the packet switch becomes available. The primary contributors to this effort have been Gregory Papadopoulos, Gregg Bromley, Chih Lee, and Frank Park. The BBN switch node is a 4x4 4-bit serial, globally synchronous crossbar. The raw bandwidth of each link is three megabytes per second, and a maximum utilization of 20% of this raw rate is expected. This yields an

aggregate useful bandwidth of over one gigabit per second for a 32-processor configuration.

We have added error detection, noise immunity, and electrical isolation to make the BBN switch more suitable for a higher integrity and more physically dispersed multiprocessor. The circuit switch can be attached directly to the TI NuBus, controlled by Lisp or Explorer Microcode. In addition, Saed Younis has engineered a high precision distribution network designed to deliver the globally synchronous six megahertz clock to sixty-four TI Explorers, within a 25 nanosecond maximum skew.

To off-load the Lisp processor from low level message formatting and protocol we are developing a microcoded I/O processor, the NuBus Channel Adapter, or NuCA. Gregory Papadopoulos, Erik Hagersten and Nils Skoglund (the latter two on loan from Ellemtel, Sweden) have developed the NuCA. The NuCA provides an intelligent DMA interface between the NuBus and a local bus. The local bus is a simplified byte-wide block oriented bus consisting of an input and output link, each capable of delivering or receiving ten megabytes per second, and an asynchronous Spy bus for diagnostics, error recovery, and configuration. The NuCA will support both the circuit and packet switches simultaneously. Internally, the NuCA provides two 5 Mips 64-bit micromachines, 4096 bytes of high speed FIFO buffering, 64K bytes of scratch pad memory, in addition to an autonomous, multiported NuBus master interface capable of four-way request interleaving and burst transfers without Lisp processor intervention.

In Fall 1984, we demonstrated a re-engineered circuit switch prototype for the Symbolics 3600. The circuit switch in its standalone NuBus configuration should enter production by Fall 1985, becoming available to MEF users during the first quarter of 1986. A NuCA prototype should be operational by the first quarter of 1986.

3.4. MEF Software

The MEF software development in progress is comprised of three parts, (1) generic software for emulating multiprocessor architectures on the MEF, (2) the software for the TTDA experiment, and (3) a general-purpose text illustration program called ILLUSTRATE.

3.4.1 Generic Software

Early in the year, we found that the low speed of the dataflow emulation (less than 100 dataflow instructions per second per processor) was at least partially due to design decisions taken in the MEF generic architecture emulation code. Richard Mark Soley (the original author), Mike Beckerle, and Dinarte Morais did some to speed tuning up this substrate. Due to these efforts, we are now emulating dataflow

instructions at the rate of about 500 instructions per second per processor. Scaling of this speed by increasing the number of processors, however, is severely bandwidth-limited by the current interconnection hardware (10 Megabit ethernet running Chaos and TCP protocols).

In order to support stepped-up use of the MEF substrate software at MIT and elsewhere, Richard Soley's thesis covering the theory and practice of its use is now available for distribution. LCS TR 339 describes the basic abstractions of the MEF software, as well as extensions and examples of its use. Among our future plans are the upgrading of the MEF substrate software for more speed, support of the Texas Instruments implementation language for portability, and support of the NuCA network interface. This work will be performed over the next six months by Richard Soley.

3.4.2 Dataflow Emulation Experiment

In Spring 1985, we decided to halt work temporarily on the dataflow emulation software. First, we knew that the upcoming switch to the TI Lisp Machines from the Symbolics machines would require some changes to the software. Since the dataflow emulation software is in need of some major revisions, it seemed best to wait until the new software environment was available before undertaking major changes.

In addition, within the next few months there are two sources for potential modifications to the architecture. At the spring meetings with researchers from I.B.M., we formed a committee to simplify the instruction set for the architecture. Also, with the simulator now producing real results, we felt that the simulation experiments might lead to some design changes for the architecture. Therefore, rather than completing the emulator for the current architecture, it seemed more practical to wait for a new specification, and in the mean time, to concentrate on learning as much as possible from simulation experiments.

3.4.3 The ILLUSTRATE Program

During the past year, work has continued on the ILLUSTRATE graphical illustrate program. Dinarte Morais, the original author, carried out most of it, with the assistance of Dawn Clarke and Richard Soley.

ILLUSTRATE is a highly interactive object-oriented graphical illustration program, designed primarily for adding illustrations to theses, reports, books, etc. Built originally around the Alto Draw concepts, design ideas have been added by many members of the group. Running on the Symbolics 3600 Lisp Machine processor, Illustrate is becoming quite popular at MIT and other sites around the country. Over the next year, Illustrate will also be ported to the TI Explorer processor.

References

1. Cardelli, L. "ML Under Unix," Technical Report, AT&T Bell Laboratories, Murray Hill, NJ, 1983.

2. Culler,D.E. "Resource Management for the Tagged-Token Dataflow Architecture," S.M. thesis, MIT Department of Electrical Engineering & Computer Science, Cambridge, MA, January 1985.

3. Milner, R. "A Theory of Type Polymorphism in Programming," J Comp & Sys Sci, 17, 1978, 348-75.

4. Mohamed-Ali, K.A. "Distributed Garbage Collection Algorithms for a Loosely- Coupled Multi-Processor System," CSALAB Working Paper 1983-03-09, Royal Institute of Technology, Stockholm, Sweden, 1983, 51.

Publications

1. Arvind and D. E. Culler. "Final Report: Program Decomposition for Multiple Processor Machines," CSG Memo 244, MIT Laboratory for Computer Science, Cambridge, MA, December 1984.

2. Brobst, S. A. "Tagged-Token Dataflow Architecture Simulation Facility User's Manual," CSG Memo 250, MIT Laboratory for Computer Science, Cambridge, MA, March 1985.

3. Culler, D.E. "Resource Management for the Tagged-Token Dataflow Architecture," MIT/LCS/TR-332, MIT Laboratory for Computer Science, Cambridge, MA, January 1985.

4. Heller, S. and K. Traub. "Id Compiler User's Manual," CSG Memo 248, MIT Laboratory for Computer Science, Cambridge, MA, May 1985.

5. Pingali, K. and Arvind. "Efficient Demand-Driven Evaluation (I)," TOPLAS, May 1985.

6. Soley, R. M. "A Third Opinion on Dataflow Machines and Languages," CSG Memo 241, MIT Laboratory for Computer Science, Cambridge, MA, October 1984.

7. Soley, R.M. "Generic Software for Emulating Multiprocessor Architecures," MIT/LCS/TR-339, MIT Laboratory for Computer Science, Cambridge, MA, June 1985.

8. Traub, K. "An Abstract Architecture for Parallel Graph Reduction," MIT/LCS/TR-317, MIT Laboratory for Computer Science, Cambridge, MA,June 1984.

Theses Completed

1. Bacon, S.. "A Supercombinator Compiler for Scheme," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1985.

2. Bromley, G. "Waiting/Matching for Tagged-Token Dataflow Architectures," S.B. Thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1985.

3. Culler, D. E. "Resource Management for the Tagged-Token Dataflow

Architecture," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, January 1985.

4. Ng, G. W. "A C-Language Instrument Control Function Laboratory," S.B. Thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1985.

5. Soley, R. M. "Generic Software for the Emulation of Multiprocessor Architectures", S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1985.

6. Vafa, B. "A Resource Management Policy for the Tagged-Token Data Flow Machine," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1985.

7. Wei, R. C-S. "The Design of An User Interface Facility for the Tagged-Token Dataflow Architecture Simulator," S.B. Thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1985.

8. Weisz, J. "A Hardware Description Translation System," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1985.

Theses in Progress

1. Beckerle, M. J. "Logical Structures for Functional Languages," M.S. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected December 1985.

2. Brobst, S. A. "Token Storage Requirements in a Dataflow Supercomputer," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected December 1985.

3. Hughes, G. "Wavefront Synchronism," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected January 1986.

4. Pinkerton, J. "A High-Speed Serial Link Optimized for Packet Switching," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected May 1985.

5. Pingali, K. "Design and Implementation of Dataflow Languages with

Streams," Ph.D. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected January 1986.

Talks

1. Arvind. A series of 4 lectures on "Computer Architecture" at DEC, Shrewsbury, MA, September and October 1984.

2. -. "Why Dataflow Architectures?," Honeywell Symposium, Minneapolis, Minnesota, September 1984.

3. -. "Why Dataflow Architectures?," IBM-Poughkeepsie 40th Anniversary Symposium, Kutsher's Country Club, NY, November 1984.

4. -. "Fundamental Issues in the Design of Multiprocessor Systems," ElectroTechnical Laboratory, Tsukuba, Japan, November 1984.

5. -. "The MIT Tagged-Token Dataflow Machine: Current Status," ElectroTechnical Laboratory, Tsukuba, Japan, November 1984.

6. -. "Why Dataflow Architectures?," University of Minnesota, Minneapolis, MN, January 1985.

7. -. "Dataflow Experiments on the Multiprocessor Emulation Facility," MIT Laboratory For Computer Science, Cambridge, MA, February 1985.

8. -. "The Goal of the Tagged-Token Dataflow Project," IBM-MIT Workshop, Essex, CT, April 1985.

9. -. "Why Dataflow Architectures?," MIT Department of Aerodynamics and Astrophysics, Cambridge, MA, April 1985.

10. -. "Why Dataflow Architectures?," University of Massachusetts, Amherst, MA, May 1985.

11. Brobst, S.A. "Waiting-Matching Requirements of the Tagged Token Dataflow Dataflow Architecture," Harris Corporation, Advanced Technology Division, Melbourne, FL., January 1985.

12. -. "Application Domain of the Tagged-Token Dataflow Architecture," Harris Corporation, Advanced Technology Division, Melbourne,FL, January 1985.

13. -. "The Next Generation of High Performance Computer Systems," MIT Sloan School of Management, February 1985.

14. -. "Performance Evaluation of the Tagged Token Dataflow Architecture," Workshop on Performance Evaluation of High-speed Computers, Institute for Computer Sciences and Technology, National Bureau of Standards, Gaithersburg, MD., June 1985.

15. Culler, D. E., "Overcoming Latency in Parallel Computer Systems," Lawrence-Livermore Laboratory, Livermore, CA, July 1984.

16. -. Experience with Tagged-Token Dataflow Architecture Simulator," Essex, CT, MIT /IBM Workshop on Multiprocessors, May 1985.

17. Iannucci, R. A. "The MIT Multiprocessor Emulation Facility," MIT Summer Dataflow Course (6.83s), August 1984.

18. -. "The MIT Multiprocessor Emulation Facility," IBM Glendale, Endicott, NY December 1984.

19. -. "High Speed Packet Communication," IBM Burlington, Burlington, VT, January 1985.

20. -. "High Speed Point-to-Point Serial Data Communication," IBM Raleigh, Raleigh, NC, February 1985.

21. -. "Dr. Strangehost (or *How I stopped worrying and learned to love CMS*)," MIT Laboratory for Computer Science, May 1985.

22. Heller, S. "Automatic Storage Reclamation," Boston University, Boston, MA, August 1984.

23. Nikhil, R.S. "Functional Databases, Functional Languages," Microelectronics and Computer Technology Center, Austin, TX, June 1985.

24. -. "Functional Databases, Functional Languages ," University of Texas, Austin, TX, June 1985.

25. Papadopoulos, G. "The MEF: A Multiprocessor Sandbox," with R.M. Soley, MIT, Laboratory for Computer Science, February 1985.

26. -. The Multiprocessor Emulation Facility," with R.A. Iannucci, MIT Summer Dataflow Course (6.83s), August 1984.

27. Pingali, K. "Sharing of Computations in Functional Language Implementations," Workshop on Functional Language Implementations, Goteborg, Sweden, February 1985.

28. –. "Sequential Implementations of Functional Languages," Royal Institute of Technology, Stockholm, Sweden, February 1985.

29. Soley, R. M. "The MEF: A Multiprocessor Sandbox," with Gregory Papadopoulos, MIT Laboratory for Computer Science, August 1984.

30. Traub, K. "An Abstract Parallel Graph Reduction Machine," Intern. Symp. on Comp. Arch., Boston, MA, June 1985.

TABLE OF CONTENTS