SIMULATION AND PERFORMANCE EVALUATION OF A GRAPH REDUCTION MACHINE ARCHITECTURE

Ananda G. Sarangi
B.S., University of Calcutta, 1980
M.S., East Carolina University, 1982

A thesis submitted to the faculty
of the Oregon Graduate Center
in partial fulfillment of the
requirements for the degree
Master of Science
in
Computer Science & Engineering

July, 1984

The thesis "Simulation and Performance Evaluation of a Graph Reduction Machine Architecture" by Ananda G. Sarangi has been examined and approved by the following Examination Committee:

Richard B. Kieburtz, Thesis Research Advisor
Professor,
Department of Computer Science and Engineering

David Maier
Associate Professor,
Department of Computer Science and Engineering

Robert G. Babb II
Assistant Professor,
Department of Computer Science and Engineering

Dan Hammerstrom
Adjunct Assistant Professor,
Intel Corporation

ACKNOWLEDGEMENTS

The completion of this particular piece of work provides a welcome opportunity to acknowledge some long standing personal and intellectual debts.

It was Dr. Richard B. Kieburtz who first introduced me to the field of functional programming and language directed computer architectures. This work reflects, to a great extent, the result of his insight, criticism, and encouragement. I am deeply indebted and very grateful to Dr. Kieburtz for his guidance and patience with this thesis and its author. This work would never have been completed without his support and advice in the development of ideas for this report.

In addition, Richard Vireday and Mark Foster have made many valuable contributions to the discussion which follows, and helped tremendously in keeping things moving forward. Richard Vireday was of immeasurable help in designing and carrying out large amount of testing for the simulator. The instruction trace analyzer reported in this work was written by Mark Foster, part of the data reported in the later chapters were obtained from the trace analyzer. I gratefully acknowledge their assistance.

I would also like to thank Dr. Ashoke Deb for many valuable suggestions he made during the course of this project.

Many thanks are also due to Dr. David Maier, Dr. Robert Babb II and Dr. Dan Hammerstrom for their efforts as my thesis committee members.

To everyone, my thanks.

Ananda G. Sarangi
The Oregon Graduate Center
July, 1984

TABLE OF CONTENTS

List of Tables vi

List of Figures vii

1. Introduction 1

2. Background 5

3. LML: An Experimental Language for Graph Reduction 18

4. The Binding of LML Programs to the G-Machine 25

5. Functional Description of the G-Machine Architecture 32

5.1 General Operation 33

5.2 Memory Organization 36

5.3 The Instruction Set 38

5.4 The G-Machine System Architecture 40

6. The G-Machine Execution Model: A Simple Example 43

7. A Novel Architecture 50

8. The Design of a Simulator for the G-Machine 52

9. Performance Evaluation of the G-Machine Architecture 57

10. Conclusion 69

References 73

Appendices 78

Appendix A Specification of the G-Machine Instructions 78

Appendix B Effective Graph Memory References of the G-Machine Instructions 83

LIST OF TABLES

TABLE Page

9.1 Summary of Dynamic Statistics 60

9.2 Dynamic Percentage of Instruction Executions 61

9.3 Dynamic Percentage of Storage Access over Execution Time 63

9.4 Performance Characteristics of New UNWIND Scheme 68

LIST OF FIGURES

FIGURE		Page
3.1	Construction of a graph	22
3.2	The Graph Reduction of *"add_one"*	23
5.1	Block Diagram of the G-Machine	35
5.2	The G-Machine System Environment	42
6.1	Pointer Stack Configuration for *"count 5 10"*	44
9.1	Arrangement of the Stack after UNWIND (Existing Scheme)	67
9.2	Arrangement of the Stack after UNWIND (New Scheme)	67

ABSTRACT

SIMULATION AND PERFORMANCE EVALUATION OF

A GRAPH REDUCTION MACHINE ARCHITECTURE

Ananda G. Sarangi, M.S.

The Oregon Graduate Center, 1984

Supervising Professor: Richard B. Kieburtz

The Graph Reduction Machine (G-Machine) is an architecture intended to achieve high performance in executing functional language programs. The success or failure of this novel architecture can only be determined by its performance in executing "real" programs. The simulator of the G-Machine, described in this thesis, makes possible detailed studies of the performance of the G-Machine architecture even though the hardware implementation of a G-Machine is not complete.

1. INTRODUCTION

Computational architectures are determined by the interplay of a number of different and interesting constraints. These include the nature of the algorithm to be executed, the data structure to be employed, the nature of the programming language to be used, the architectural units to be designed and the means for their interconnection, and the implementation technology.

Unquestionably, the most important determinant factor of these constraints is the nature of the programming language to be supported by the the machine architecture. This in fact dictates the architectural units and the instruction set that need to be designed to support the efficient execution of the source language.

Due to the advent of VLSI [1], special purpose computer architectures can be realized that are not based on traditional, sequential control flow program organization, but on alternative, naturally parallel organizations such as reduction or data flow to interpret high-level-language (HLL) constructions. In fact the technological advancement gives new ground rules ("language -> architecture -> machine") of system design and makes possible a whole new relationship between language and technology. Architectures that use this new concept of system design have machine organizations that

are distinguished by the form of their instructions, the way the instructions manipulate their operands, and their patterns of control and storage structures. These machine architectures need not designed to be general purpose, instead they may be highly specialized and have the attributes of so-called language-directed architectures that eliminate the semantic gaps of conventional computing systems.

A whole spectrum of special-purpose computer systems are now under development to support Direct Execution [2-5] of one or more high-level-languages. A direct execution interpreter is defined to execute the intermediate language of the source program without further any translation to low level machine language. Graph reduction is now considered as an important computing model for direct evaluation of functional language programs [6]. The basis of graph reduction is that

1) each expression is evaluated at most once, no matter how many times the expression is referenced.
2) evaluation of an expression is simply the reduction of a function application to its value.
3) the expression evaluation mechanism supports non-strict semantics. Thus evaluation of a function application need not cause evaluation of the arguments of the application.

4) every expression is evaluated in a static environment.

In graph reduction, an expression that is under evaluation is represented as a rooted, directed graph. Initially, to evaluate an expression, an instruction demands the value associated with the given expression. Instead of making a copy of the definition, the evaluator merely traverses the graph, with the aim of reducing the graph into a normal form -- one for which no further reduction steps are possible.

One of the ways of identifying the demand, is to maintain a reference to the original source of the demand in the environment throughout the evaluation process. During the course of evaluation each instruction that accesses a particular definition of the graph simply manipulates the reference to the definition. This traversal of the definition and manipulation of the references is continued until all the arguments of an expression are reduced to some canonical form.

The concept of graph reduction that is described here comes from the work of Thomas Johnsson [6] and other works on reduction computer architectures [7-10]. Johnsson's paper describes an abstract machine architecture for executing functional language programs with lazy evaluation. Using Johnsson's abstract model, this thesis is an attempt to present and justify a graph reduction machine architecture that will efficiently execute functional

programming languages, with the ultimate aim of producing a suitable implementation in VLSI. We do this by presenting a brief description of the compilation of the language, the machine organization, and some initial results from our simulation of the architecture.

A compiler [11], peephole optimizer [12], and assembler [13] for the G-Machine have been developed. The simulator for the G-Machine that is described here is capable of simulating the execution of substantial programs and producing instruction and data address traces. Executions of a few programs are simulated and a trace analyzer [14], developed as a part of this project, is used to obtain dynamic instruction frequencies. These results are used for a preliminary evaluation of the novel features of the G-Machine architecture.

We start here, by giving background for the work that makes graph reduction an important computing model for evaluation of functional language programs.

2. BACKGROUND

Over the past few years there has been a growing interest among language designers in functional (or applicative) languages as a potential alternative to conventional programming languages, particularly since the publication of Backus's influential paper[15]. This interest arises from two distinct points of view, one coming from software considerations and the other from rapid developments of semiconductor technology. Despite the advantages of functional languages for their elegance, clarity, and expressive power, current implementation make less efficient use of machine resources than do implementations of imperative languages. Thus it is a challenge to produce an implementation of a functional language and to design a computing system that will execute functional languages more efficiently than existing computing systems.

A principal technique that is used to evaluate functional languages is reduction. The canonical reduction architecture is based on the theory of lambda calculus [16,17]. A lambda calculus-based system is founded on the use of lambda abstractions to represent mathematical functions. The basic reduction rule, known as "beta-reduction", realizes the application of a function to an argument expression, by substitution of the argument for all

occurrences of the bound variable (formal parameter) in a lambda expression. The lambda expression with its associated substitution rules is capable of defining all possible computable functions of all possible types and of any number of arguments. If one starts with a simple lambda calculus evaluator based on substitution rules, and introduces the device of using pointers to the expressions rather than manipulating textual expressions themselves, one is lead to a graphical representation of the expression. In a lambda reducer, the bindings of variables to the expressions they represent constitutes an environment. Reduction transformations take place relative to an environment.

The first practical lambda evaluator was proposed by Landin [18] and is known as the SECD machine. The SECD machine is an abstract architecture where applicative expressions are represented as trees with leaf nodes labeled by identifiers that name variables. The SECD machine has four major components:

S	the stack	used to hold intermediate results in the traversal of expression trees;
E	the environment	used to hold the values bound to variables during the process of evaluation;
C	the control list	used to hold machine instructions to evaluate an expression;
D	the dump	used as a stack to hold values on recursive function calls;

The stack is used in this machine as a temporary store for evaluation of function applications. During the course of evaluation, on each function call a new area of the stack is allocated to hold the parameters for the function environment. If the function body has a nested function call then the suspended environment is saved on the dump and a new environment is brought onto the stack before the evaluation continues. Most of the complexity of the SECD machine arises in constructing the environment correctly. Specifically, if an expression contains free variables then one must either substitute expressions for the free variables or pass around the environments in which the free variables are defined. This creates a substantial amount of overhead in the SECD model in reducing a lambda-expression to its value.

The evaluation scheme used in the SECD model is totally applicative-order, hence strict. In this scheme a function's arguments are evaluated prior to the application of the function, which corresponds roughly to the call-by-value function calling mechanism of most Algol-like languages. This evaluation mechanism for functional language implementation is not safe. That is, for some programs this applicative-order evaluation may not terminate at all [19], although a terminating computation is possible when a different evaluation rule is used.

Besides the SECD machine, another lambda reducer we know of, also based on an applicative-order evaluation scheme, is Berkling's GMD reduction language machine [20]. The evaluation technique used in this machine is very classical in nature and makes use of some well understood mathematical properties of beta-reduction. This architecture assumes that the expressions are represented as strings, and it provides hardware support to dissect expression strings using a set of push-down stacks. A more elaborate description of this machine and its program organization can be found in [21,22].

The model of computation that Berkling's machine supports is known as string reduction. String reduction differs from graph reduction in that each reduction step accessing a particular definition will manipulate a separate copy of that definition. On the other hand, graph reduction is based on the concept of manipulating pointers to the expression and making updates in place for function applications rather than recopying the expressions themselves.

Two other string reduction computing systems that have been proposed to do applicative-order evaluation are Mago's Cellular Tree machine [23] and Treleaven's demand-driven multi-processor reduction machine [24].

The organization of the Cellular Tree machine is a binary tree structure and it directly executes Backus's functional programming language FP [15]. A program for the tree machine is a linear string of symbols that are mapped onto a vector of memory cells in the computer. During the execution, the expression under evaluation is partitioned into a collection of cells and their corresponding microprograms (associated with each cell) execute to reduce the expression to a value.

The multi-processor reduction machine as described by Treleaven is radically different in nature from the Cellular Tree machine. It supports a user-defined reduction language and operates in a highly parallel manner. The machine consists of three major parts,

1) A common memory containing the definitions;
2) A set of identical, asynchronous processing units;
3) A large segmented shift register containing the expression to be evaluated.

Thus the machine provides the storage medium for both the expressions and the definitions, a means of controlling the traversal of the expression and recognizing reducible subexpressions, and a processor to perform the reduction.

The next evaluation scheme is based on the theory of lambda calculus, but is quite different from the applicative-order evaluation scheme, and is known as normal-order evaluation. In normal-order evaluation the left-most redex (reducible subexpression) is reduced first, which specifies that the outer subexpressions are reduced before the inner ones. This gives normal-order evaluation more expressive power than applicative-order evaluation in that the evaluation is safe and has the terminating property (if there exists any terminating sequence of reductions, then a normal-order sequence terminates). To see that the applicative-order evaluation terminates less often, suppose that a function which returns a constant answer (for example: **hd** *(from '0')*, see next section for the definition of *from*) is applied to a non-terminating sub-expression. Here evaluation will terminate under normal order but not under applicative order, where it will be attempted, in vain, to reduce the sub-expression completely before discarding it (i.e. normal order supports non-strict functions while applicative order does not). Graph reduction normally corresponds to normal-order evaluation. This method also has the property that no subexpression is evaluated more than once (value sharing; also called "fully lazy" [25] evaluation).

The correctly-implemented graph reduction evaluators we know of to date are software implementations of abstract

machines. One of these, the AMPS machine, a token-matching loosely-coupled multi-processor [8], was designed to simulate the parallel evaluation scheme of a LISP like language. The second is a hybrid multiprocessing system (a refined version of AMPS with a different interconnection network) of dataflow, reduction and von Neumann processes called REDIFLOW [9], which was designed to exploit the implicit concurrency of functional languages. Another is a token-storage, parallel graph reduction architecture ALICE [26] that evaluates a variety of applicative languages. All of these machines have significant architectural similarities, including the evaluation scheme they use. At the logical level, similarities in the evaluation mechanisms and the treatment of higher-order functions show that all three schemes have their root in the graph reduction model of computation. At the physical level, AMPS is arranged as a tree structure and it exploits parallelism at the level of user-defined function calls. ALICE is organized somewhere between a ring and bus structure, and it achieves parallelism both at the level of function calls and also inside function bodies. Both of these machines are based on the concept of multi-processing and they use a large random access memory in a distributed fashion (through packet communication) during the course of evaluation.

Using a new implementation technique (based on the results derived by Curry and Feys [27]), Turner [7] showed how to

implement a reduction machine using combinator calculus. Combinator calculus uses a technique called bracket abstraction to eliminate variables from lambda expressions, producing an expression consisting solely of constant operators (combinators) and data. There are reduction rules defined for each combinator, but as there are no variables, combinator reduction does not involve substitution. Also, there is no environment. The combinators are called S, K, I, B and C and obey the following equations.[1]

$$\mathbf{S} f\ g\ x = f\ x\ (g\ x)$$

$$\mathbf{K}\ x\ y = x$$

$$\mathbf{I}\ x = x$$

$$\mathbf{B} f\ g\ x = f\ (g\ x)$$

$$\mathbf{C} f\ g\ x = f\ x\ g$$

Combinators lend themselves naturally to lazy evaluation [28] of the lambda calculus. Lazy evaluation is a technique where the evaluation of arguments of a function is postponed until the value of the argument is actually required, and the result of the evaluation is made available to other functions that use the same arguments.

[1]Here we denote application by juxtaposition and assume it associates to the left. Thus, for example $f\ x\ (g\ x)$ means the result of applying (the result of applying f to x) to the result of applying g to x.

Thus the two calculi can be thought of giving two different machine codes for a functional language implementation. The question naturally arises which approach is more efficient. This question has been investigated by Jones [29]. The comparison between the two systems was made by writing reducers for each, and measuring the costs of the computation in terms of the number of accesses to data structures whose size is potentially unbounded.

In the classical combinator reduction scheme as described by Turner [7], the programs are transformed into expressions containing the combinators S, K, I, B, C, S', B', C', Y, such that all variables are removed from the program body. This combinator expression is then evaluated using normal-order graph reduction, i.e., an expression subgraph is transformed to its value when the value is needed. Thus the combinator reduction involves graph reduction instead of environment manipulation, and it requires an extra stack (called the reduction stack) that initially contains (pointers to) the expression to be evaluated.

The hardware realization of a combinator reducer, called SKIM has been proposed by Clark, et al. [10]. The SKIM machine organization is based on a conventional microprocessor with microcoded instruction sets that directly support combinator reduction. These instruction sets are very similar to that of Turner's S-K reduction machine, and they can be further classified as combinator

instructions, list operators and standard arithmetic instructions. The programs are represented in SKIM by a graph built of two element cells, which is implemented by dividing the memory into two banks, HEAD and TAIL. The SKIM architecture is driven by a combinator reducer that scans down the left-most branch of the program tree to find the operator at the leaf. This pointer, after being used to go down one level in the tree, is reversed to indicate the return route. Eventually the sequence of pointers from root to leaf in SKIM is transformed into a sequence of pointers from leaf to root. Thus SKIM uses pointer reversal instead of stacks to traverse the program tree.

The idea of forming efficient combinator expressions for graph reduction has been recently proposed by Hughes [30]. In this scheme, a set of super-combinators are used for program transformations instead of proper combinators (super-combinators are closely related to "proper combinators", but they differ in that constants may occur in their bodies and in that the combinators themselves are regarded as program-defined constants). An attractive feature of this scheme is that once an application is evaluated it can be overwritten by its new value. This ensures that every expression is evaluated at most once regardless of the number of times it has been referenced. It is related to the lazy evaluation of lambda calculus [28], in which every expression

passed as a parameter to a lambda-expression is reduced at most once. This scheme is advantageous over classical combinator reduction in that it achieves full laziness more directly by employing some optimizing steps (such as replacing original lambda expressions with simpler ones) in the translation scheme.

The evaluation approach used in the G-machine bears close resemblance to the super-combinator approach. In this scheme, a program that contains lambda-expressions is transformed into an expression (without lambda expressions) and a set of function definitions. This process is known as lambda lifting, and is analogous to Turner's [7] bracket abstraction process and Hughes's [25] super-combinator abstraction scheme. This process, like the other two processes, ensures that only the variables that are really referred to in expressions are bound as parameters to functions, instead of binding the whole environment.

The G-machine architecture is superficially similar to that of Landin's model but it has different components than the SECD machine. A state in the G-machine can be described in terms of 6-tuple <C, P, V, G, E, D>, where

C	the control list	which holds instructions for the G-machine;
P	the pointer stack	which contains pointers into the graph;
V	the value stack	which is used for evaluations of

		arithmetic expressions;
G	the graph	which is the image of an expression that to be reduced;
E	the environment	which provides the context in which expressions are evaluated;
D	the dump	which is used as a stack to store context on recursive functions calls;

The G-machine simplifies Landin's model of applicative order evaluation. The graph which is under evaluation is represented as a rooted, directed graph. This expression graph is not necessarily a tree, nor even acyclic. Because applicative expressions have the property of referential transparency, the value that an expression represents remains the same whether or not the expression has been evaluated. Graph reduction is the process that transforms an expression graph to a normal form by applying a set of evaluation rules until no further reductions are possible.

The environment, E, is static in the G-machine, because a graph which is under evaluation contains no occurrences of free variables. All arguments of a function are bound in function applications. Thus the environment E contains only the bindings of constants, which in the source program are just the defined functions. In E, each function name is associated with a code sequence that represents its value. This provides the G-machine a major advantage over the SECD machine in setting the environments properly (by the process of lambda lifting) during the computation, and it is one of the principal contributions of the G-machine compiler [11].

For the sake of efficiency, the G-machine has two stacks, one for traversing the graph and the other for storing intermediate results during evaluation of arithmetic expressions. Use of the V stack in this architecture reduces the incidence of indirect reference to values that have already been computed and the P stack provides a mechanism to access the arguments of an expression that is under evaluation.

The control, C is just a sequence of instructions for the G-machine. The G-machine is a sequential evaluator, with an instruction counter that advances the state of the control each time an instruction is executed.

The dump, D, stores the contexts of nested function applications. A context during the computation is simply the image of two stacks, P and V, and the contents of the program counter.

The abstract G-machine that we described here executes an extended subset of the applicative language ML [31] with lazy evaluation, called Lazy ML or LML for short. Although the language semantics are not the central issue in this thesis, we describe the program organizations of LML in the next section to provide a framework for our specifications on the instruction sets for the G-machine and its execution model.

3. LML : AN EXPERIMENTAL LANGUAGE FOR GRAPH REDUCTION

LML is a strongly-typed, purely applicative programming language. It is based upon the (impure) applicative language ML, designed by Milner as the metalanguage of LCF [31], a system for formal reasoning about programs. It permits the definition of higher-order functions, has recursive data types, and uses deferred (lazy) evaluation on lists and tuples, thereby admitting a programming style to deal semantically with unbounded data structures.

The smallest LML program unit that can be compiled and run is a single expression. Any executable LML program consists of an expression to be evaluated, but the expression may be a compound expression, involving the evaluation of locally defined constant expressions or of global definitions. Here is a simple example using the polymorphic function called *map*.

Map takes a function f (of type α->β), and a list l (of type list(α)), and returns the list obtained by applying f to each element of l (which is of type list(β)). *Map* is defined as

map: (α -> β) -> list(α) -> list(β)

```
letrec map = λf.λl.case l in
                nil : nil
        ||      a.rest : (f a) . map f rest
end
```

Here period (".") is the infix list construction operator and the double stroke ("||") is the separator for case instances in case list.

To apply *map* on actual parameters we define the following functional program to add one to each member in a list of numbers, its value being the list of natural numbers incremented by one.

```
let F = λa.λb.map(λx.x + a) b
in
    F 1 [1; 2; 3; 4; 5]
```

All function applications in LML are "curried" [18,32]. That is, all functions are assumed to take just one argument, which has no restriction, since that function may return another function that takes one argument. *Map* is such a function that we just described.

LML scoping rules are similar to those for most block-structured languages, in that expressions may reference any identifier defined locally in the current equation group, or in any surrounding equation group. However, local references are allowed to be mutually recursive. This implies that equations may appear in any order.

LML also has a powerful ***pattern-matcher*** through which complex functions may be defined more easily. A pattern may have one of the following form; identifier, pair, cons, or list. For example, the function ***append*** x,y takes two arguments, x and y (both lists), and produces as a result the single list which has all the items of x followed by all the items of y. ***Append*** can be defined as

```
append:    (list(α) -> (list(list(α)) -> list(α))


letrec   append ([],[]) = []
      || append (L1,[]) = L1
      || append ([],L2) = L2
      || append (L1,L2) = hd L1 . append (tl L1, L2)

in
      append ([1;2;3],[4;5;6])
```

The pattern-matcher is very useful when used with lists. Lists are evaluated "lazily". The selector functions are **hd** ("head") and **tl** ("tail"), for lists and **fst** ("first") and **snd** ("second") for pairs.

"Infinite lists" are defined in the obvious way. The infinite stream of numbers starting at zero can be defined by the function "***from***".

```
from:    int -> list(int)
```

```
        let succ n = n + 1
in
        letrec from n = n . from (succ n)
in
        from 0
```

Where *succ* is a successor function, which when applied to its argument returns its successor.

Of course, elements of an infinite list are not computed until they are selected ("demanded") for evaluation. When a demand for a value of a list type occurs, the function returns a pair of closures (a closure consists of an expression together with bindings for all free variables in the expression). One closure is evaluated to find the first element of the list, and the other one is used to evaluate the remainder of the list.

There are other syntactic and semantic features of LML, but they are beyond the scope of this paper. The interested reader can find more about the language LML, its semantics and the implementation details in [11,33].

To illustrate normal-order graph reduction in LML, we take our first example *map* and define a function *add_one* to add one, to each member in a list of numbers, where *add_one* is equivalent to

```
let add_one = λa.λb. map (λx.x + a) b
in
    add_one 1 [1; 2; 3; 4; 5]
```

Which gives us following series of graphs (see Figures 3.1 and 3.2).

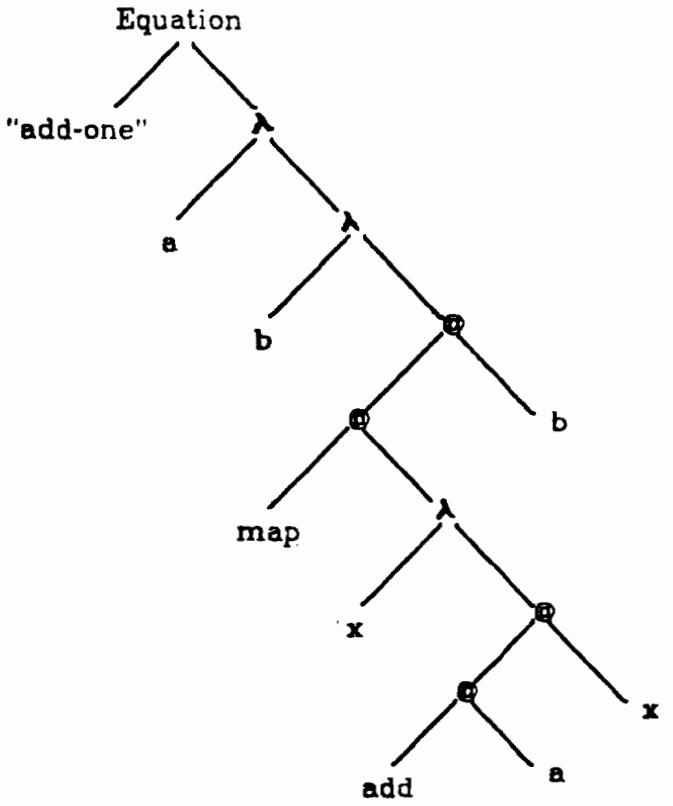


Figure 3.1(a) Definition of 'add_one'

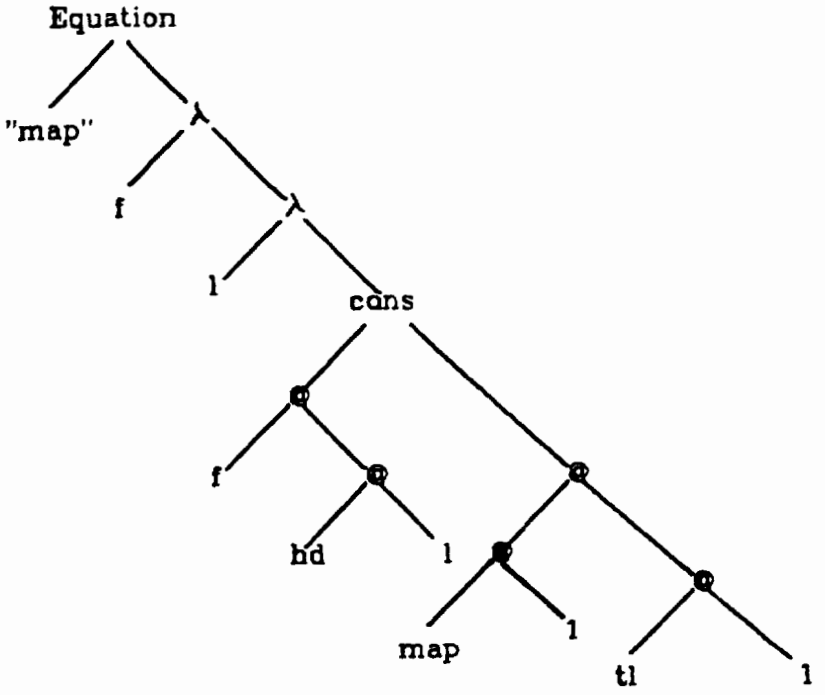


Figure 3.1(b) Definition of 'map'

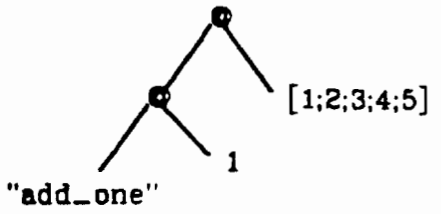


Figure 3.2(a) Construction of the graph for 'add_one'

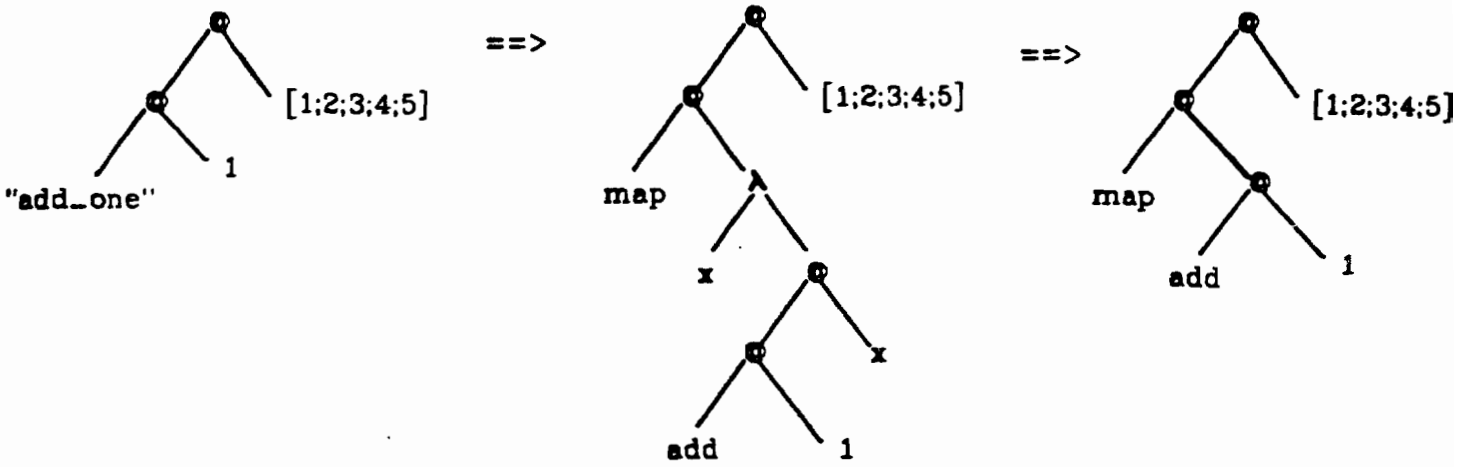


Figure 3.2(b): Substitute definition of 'add_one' and bind its arguments (the second reduction is an instance of the rule of η-reduction in the lambda calculus; $\lambda x.\text{add } 1\ x = \text{add } 1$)

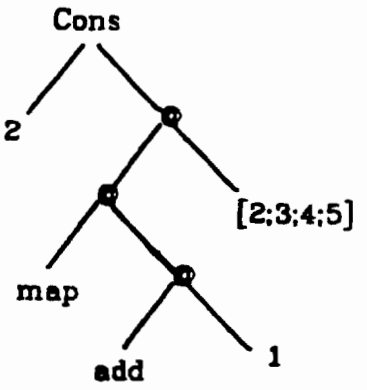


Figure 3.2(c) The graph reduction of 'add_one'

The graph reduction execution of a program is carried out by performing transformations on the expression graph to reduce it to its value. In particular, evaluation of a function application amounts to repeatedly transforming an application graph to an instance of the right-hand side of the function definition, with arguments substituted for parameters, until no further reduction is possible. Thus the function definition represented in Figure 3.1(a) is applied to an argument in Figure 3.2(a) (where @ denotes function application in the Figures). Figure 3.2(a), represents a graph just before the start of evaluation. The Figure 3.2(b) shows the graph after substitution. In Figure 3.2(c), the graph is in the cons form, whose head part is reduced to an integer value, whereas the tail part remains unchanged and needs to be evaluated.

The execution continues in this fashion until there are no more evaluations to take place for the arguments of the list constructor. When a part of the graph has been reduced to a canonical form (by definition the graph is said to be in canonical form when the root application node is reduced to any one of the following form: int, bool or cons), it may then be used for output. A node will be removed from the graph when no further reference to it is possible.

4. THE BINDING OF LML PROGRAMS TO THE G-MACHINE

Over the past few years, many architectures have been developed in order to support direct execution of high-level languages. Such architectures differ from traditional implementations by the elimination of machine language. Various direct execution schemes have been proposed. Some assume that the high level language constructions are to be executed directly by the hardware without going through any form of translation to an intermediate language [3,34,35,36]. However, most of them advocate a less extreme solution that uses an internal form to represent the source program. Typical direct execution schemes are classified into two models. The first one, analogous to the compilation of conventional models, translates the source program into an intermediate representation suitable for immediate interpretation. The second one, holds the high level language program (or the directly executable language –DEL) itself in the hardware and interprets it by a fixed microprogram [37].

In comparing these two schemes, the general feeling is that the overhead in decoding and interpretation by a DEL architecture of a program that does not go through any compilation is so overwhelming that such architectures will never become viable for executing

high-level languages [38]. A principal reason for this realization is that it is not reasonable to begin program executions without first checking for syntax or detectable semantic errors. These architectures are also less efficient than the other approach in that all binding is performed dynamically and repetitively during execution. Thus every time the machine executes a statement, it perform the lexical analysis, parsing, and some semantic functions of a compiler, causing an enormous amount of overhead.

The operating principle of a pure reduction machine [23] is radically different from a traditional von Neumann computer, which synthesizes a result by altering stepwise a state vector into a set of values. A reduction machine always starts out with a denotation of the result, which gets reduced by meaning-preserving transformations to the final result. The latter is characterized by the absence of any instances of reduction rules that could be applied. A reduction language program may be viewed as a set of definitions, name:expression pairs, and an expression to be evaluated. Evaluation is usually based on a set of reduction rules consisting of arithmetic, logic, conditions, beta-reduction, and recursion. Rules have no side effects and preserve referential transparency of expressions. This leads to the main architectural idea to evaluate literally (or virtually) "in place", i.e. in the expression. This idea necessitates an appropriate set of reduction operations

that need to be performed on the input source program. These operations are respectively:

1) Replace a leaf node whose value is an identifier by the graph bound to that identifier in the (static) environment.

2) Replace any graph whose principal operator is an arithmetic or boolean operator by first evaluating its operands, then applying the operator. The resultant value replaces the operator graph.

3) Reduce a graph whose principal operator is "if" (conditional) by first evaluating the predicate subgraph, then selecting one of the two alternate graphs for evaluation.

4) Reduce an application graph by first evaluating the left subgraph (function part). Then, if the left subgraph is:

   (i) a language-defined function, replace the application graph by the value of the function applied to the argument subgraph; else

   (ii) if the left subgraph is a lambda expression, replace the application by a copy of the body of the lambda expression in which all references to the lambda-bound variable are replaced by references to the argument subgraph, then reduce the resulting graph.

This model has in common with combinator reduction that applications of programmer defined functions are evaluated in normal-order and by rewriting the graph, and has in common with lambda calculus reduction that bound variable occurrences are replaced by argument expressions. However, it differs from combi-

nator reduction in that complex expressions (programmer-defined functions) have equal status with elementary combinators, and differs from lambda calculus in that bindings of variables occur by position rather than by name. Hence there is no dynamic environment.

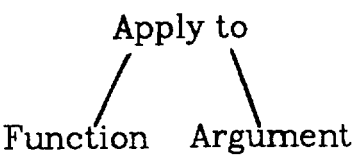
The principal diseconomy in this model of computation is that of recopying function graphs and rebinding their variables. This is particularly noticeable when computation is repetitive. Thus it is difficult to evaluate efficiently certain expressions, such as one in which a function of several arguments is applied to argument expressions each of similar form. The problem is that each such applicative expression requires the construction of a fairly large graph. Constructing such a graph may be much more costly than reducing it.

An alternative strategy is to employ a sequential machine capable of executing individual instructions to apply language-defined operators, implement conditional selection by jumps to labelled instructions, etc. When this is done, it becomes unnecessary to build a graph to represent the application of a function-valued constant to an argument expression, provided that the application is to be evaluated immediately. A function can be represented by a code sequence rather than by a graph. Although evaluation of sequential code may require auxiliary (stack allocated) temporary

storage, such storage would be required also by a mechanical evaluator of pure graph reduction.

The G-machine architecture that has been designed employs the technology of a sequential, stored program machine to avoid much of the overhead in copying graphs that is inherent in the pure graph reduction model. It is this use of sequential machine technology that allows its performance to be competitive with sequential evaluators of imperative language programs.

The G-machine architecture has been developed in order to support the direct execution of LML programs after translating them into an intermediate form. LML is a natural choice for graph reduction because the compilation schemes used for constructing the graph and further reducing them to a value closely correspond to the semantics of the language that support a set of reduction operations. LML is a very simple language in which only a few primitive operators and data types are sufficient for implementing powerful algorithms with complex data structure. Initially, an LML program which is under execution is transformed (using a set of compilations schemes) into the following primitive graphical form on which graph reduction is performed.



Here function and argument may themselves be non-trivial expressions, and the "apply to" operator is regarded as a constructor that relates the two subexpressions.

At runtime, when the evaluator wants to reduce an application involving a user-defined function, it executes the machine code for this function that is produced by the LML compiler to reduce the function application to its value.

Various compilation schemes are used to translate a LML source program into a set of instructions that can be interpreted directly. Here we summarize the compilation schemes that are used for translating function definitions and expressions into executable code. (The complete set of compilation rules can be found elsewhere [33].) The compilation schemes are

1) to generate code for a function definition f = λx_1. ... λx_n that will reduce a graph representing an application f e_1 ... e_n to its value.
2) to generate code that constructs a graph representing an expression and leaves a pointer to the resulting graph on the top of the pointer stack.
3) to generate code that computes the value of an expression and leaves a pointer to the result on the top of the pointer stack.

4) to generate code that computes the value of an integer-or boolean-valued expression and leaves the result on the top of value stack.

For an intermediate language to act as the instruction set of a machine, it is necessary that all computational problems can be executable in this language. Hence the language must contain primitives for:

1) arithmetic and logical operations.
2) conditional operations.
3) iteration and recursions.
4) list manipulations.

These instructions (which can be thought of as an intermediate language in conventional implementation) are obtained from the G-machine compiler by translating the abstract syntax tree. These instructions and their usage are summarized in the next section.

5. FUNCTIONAL DESCRIPTION OF THE G-MACHINE ARCHITECTURE

The G-machine is a 32-bit, stack-oriented machine [39], designed exclusively to execute functional language programs. Its stack architecture was specifically developed to provide efficient means for graph traversal and to maintain local environments during the course of an evaluation.

The G-machine configures a memory into three parts. The control memory (C), holds the sequential instructions compiled from an LML program, and is read only by the processor during program execution. The graph memory (G), is a directed, rooted graph that is both read and updated during an evaluation. At any given time, the graph holds the representations of partially evaluated program structures. As the evaluation continues, the graph grows and shrinks dynamically and therefore requires storage management by garbage collection. The dump (D), which is used as a stack, stores contexts on function applications.

The processor itself also contains a pair of stacks. The pointer stack (P) holds pointers to traverse an expression graph, and it provides a local environment and a dynamically allocated work area for the currently active function application. The value stack (V) holds intermediate arithmetic and logical values produced in

the course of an evaluation. The information sent to or recovered from the dump in a context switch consists of the images of pointer and value stacks, processor status flags and the contents of the program counter.

5.1. General Operation

The internal functions of the G-machine are partitioned logically into two units. The first is the Bus Interface Unit and the second is the Execution Unit, as shown in the block diagram of Figure 5.1. These units can interact independently but for the most part they perform their operations as synchronous operational units.

The bus interface unit provides the functions related to instruction fetching, operand fetch and store, bus control and storage allocation using a 40-bit bus (of which the lower 32 bits are multiplexed for address and data, while the upper 8 bits are solely used for tag; see the next section for tags).

The execution unit receives pre-fetched instructions from the bus interface unit and executes the instructions. It primarily performs the basic arithmetic and logical operations and stores the result back into the value stack or passes it to the bus interface unit for updating the graph. The execution unit also operates on

the pointer stack under program control and manipulates its contents to build a proper environment before a function is called.

The general organization of the execution unit is quite simple. It is composed of two register stacks, a value stack and a pointer stack, used as a small local memory, an ALU, a status register, a stack pointer register (for the dump memory), an instruction decode unit, a microprogram memory (which generates sequences of micro-instruction address words to control various functional modules), and an input/output unit to provide value and pointer stacks an interface to the bus interface unit.

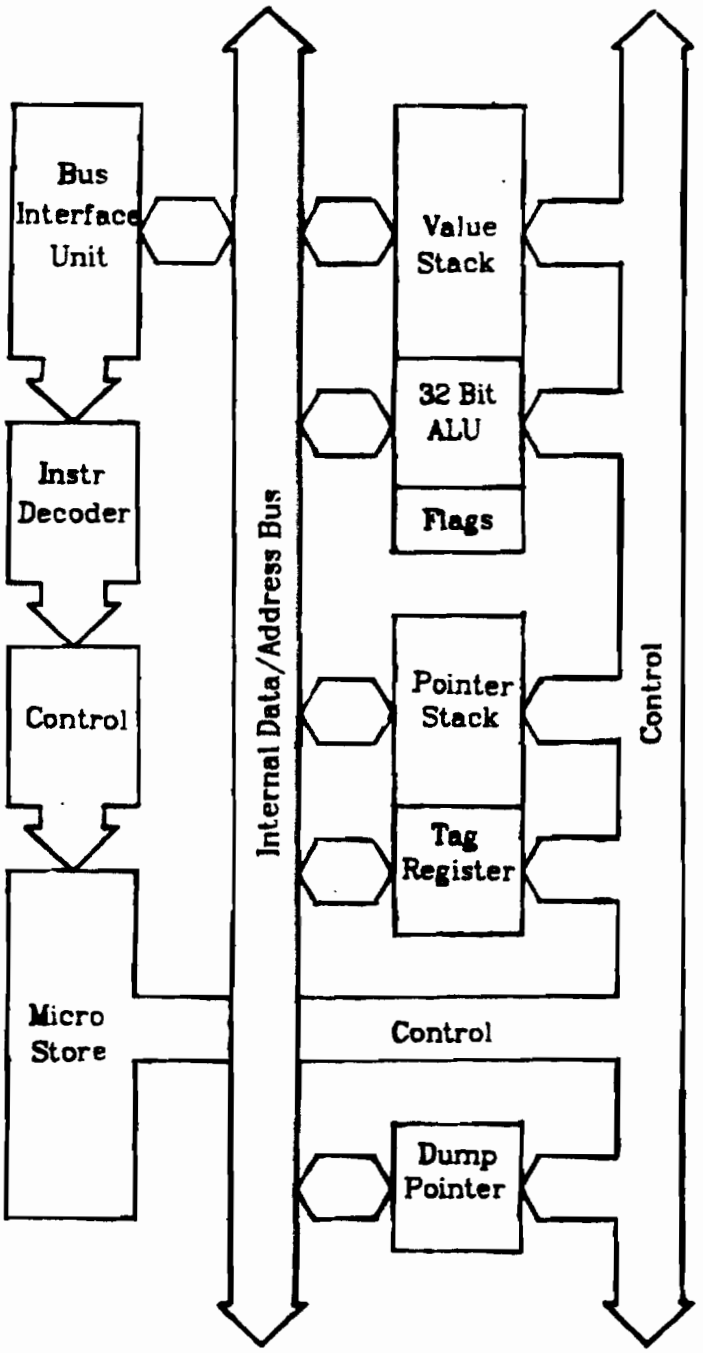
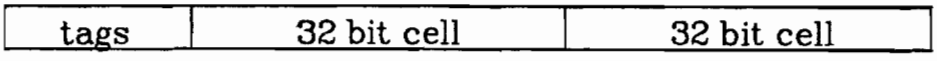


Figure 5.1 Block diagram of the G-machine

5.2. Memory Organization

The control memory for the G-machine is organized as a set of consecutively numbered storage cells [39], where each of these cells contains eight bits of information.

The graph memory is comprised of individually addressable, allocatable units, each consisting of a pair of storage cells and a vector of five tag bits. Each storage cell holds 32 bits of information, and is configured as:



The tag bits are used for graph reduction and to support a parallel, on-the-fly garbage collection. Their uses are as following:

1) A pair of bits is reserved for use in storage allocation and garbage collection.
2) A pair of bits (one for each cell), which if set, represents that the cell currently holds a pointer.
3) A bit is exclusively reserved for graph node status information. If set, this bit represents that an expression is in normal form. Otherwise, the object represents a function application.

The allocated cells in the graph memory are used in constructing the graph according to the environment represented by the

pointer-stack, and are traversed during a function call. An allocated cell-pair can be used to represent:

1) An individual scalar value, in which case only the first of the cell pair holds data;
2) A pair of scalar values;
3) A function application, in which the first cell points to an expression denoting the function to be applied, and the second cell points to its argument;
4) A list constructor, in which the first cell points to the head of the list and the second cell points to the rest;
5) A hybrid representation of a list or value pair, in which one cell holds a scalar value, and the other cell contains a pointer to the other member of the pair;

It is not absolutely necessary to make use of hybrid representations. However, use of the hybrid representations allows some reduction in indirect reference to scalar values embedded in pairs and evaluated lists. The abstract G-machine as described by Johnsson does not make use of hybrid representations on pairs. These representations were incorporated in our definition of G-machine architecture to provide more efficient access to scalar data items embedded in pairs after we noticed the lack of support from the LML compiler to access such data items.

The dump memory for the G-machine is organized as a linear array of storage cells. Each of these cell contains 32 bits of information.

5.3. The Instruction Set

The expression graphs are evaluated (reduced to their values) using a set of instructions; these instructions are closely tied to the semantics of the language and traverse the graph to reduce it into a normal form.

The basic instruction for the G-machine is one byte long. The instructions are grouped into seven principal categories. The upper three bits of the instruction byte are used to reflect the instruction-group it represents and the lower five bits are used to identify the instructions in the group. Details of the instruction specifications are given in Appendix A [39], however we describe the instruction-groups here.

Graph traversal and allocation instructions: These instructions control expansion and contraction of the expression graph. This is done by manipulating the pointers contained in the pointer stack. These instructions also update and allocate new nodes in the graph and control the evaluation of function application.

Data fetch instructions: These instructions bring values from nodes of the expression graph or from the instruction stream to the value stack.

Data structure selector instructions: These instructions operate on data structures, such as a list, and select its components.

Arithmetic and logical instructions: These are conventional instructions, and they do the arithmetic and logical operations on the contents of value stack.

Control transfer instructions: These instructions allow conditional jumps in the control stream, and provide backward jumps in the case of tail recursive function applications.

Function call and return instructions: These instructions support function call and return operations of LML programs. A function call entails saving the state of the calling function, dynamically allocating and initializing local storage for the called function, binding arguments, and executing the called procedure.

Instructions to support user defined data types: These instructions support user defined (abstract and/or recursive) data types. User-defined types can be defined as disjoint unions of sets of values, each formed by the application of a canonical constructor function

to arguments. The instruction INJECT constructs a representation of such a term as a tag-value pair. The instructions IS and OUT respectively test the tag and give a projection of the value component of such a pair.

5.4. The G-Machine System Architecture

In any project that attempts to construct a prototype of a new processor architecture, the question always arises, "How one can possibly evaluate the processor without constructing an entire computer system?" The goal of the G-machine project is to evaluate the performance of some new ideas for a processor architecture. Therefore it requires an environment that will not inhibit performance.

In order to provide an operating system environment the prototype G-machine is planned to be run as an asynchronous, parallel co-processor to a conventional microprocessor. The Figure 5.2 shows the G-machine system environment. The G-machine will obtain services such as initialization of its program memory and the graph memory, file input and output, a real-time clock, display management, code compilation and linkage and even garbage collection of the graph memory from its managing co-processor. The co-processor and the G-machine will be linked by some special

purpose control circuitry, and will share dual-port access to the G-machine's program and graph memories. This environment is expected to provide the necessary performance evaluation of the G-machine processor architecture.

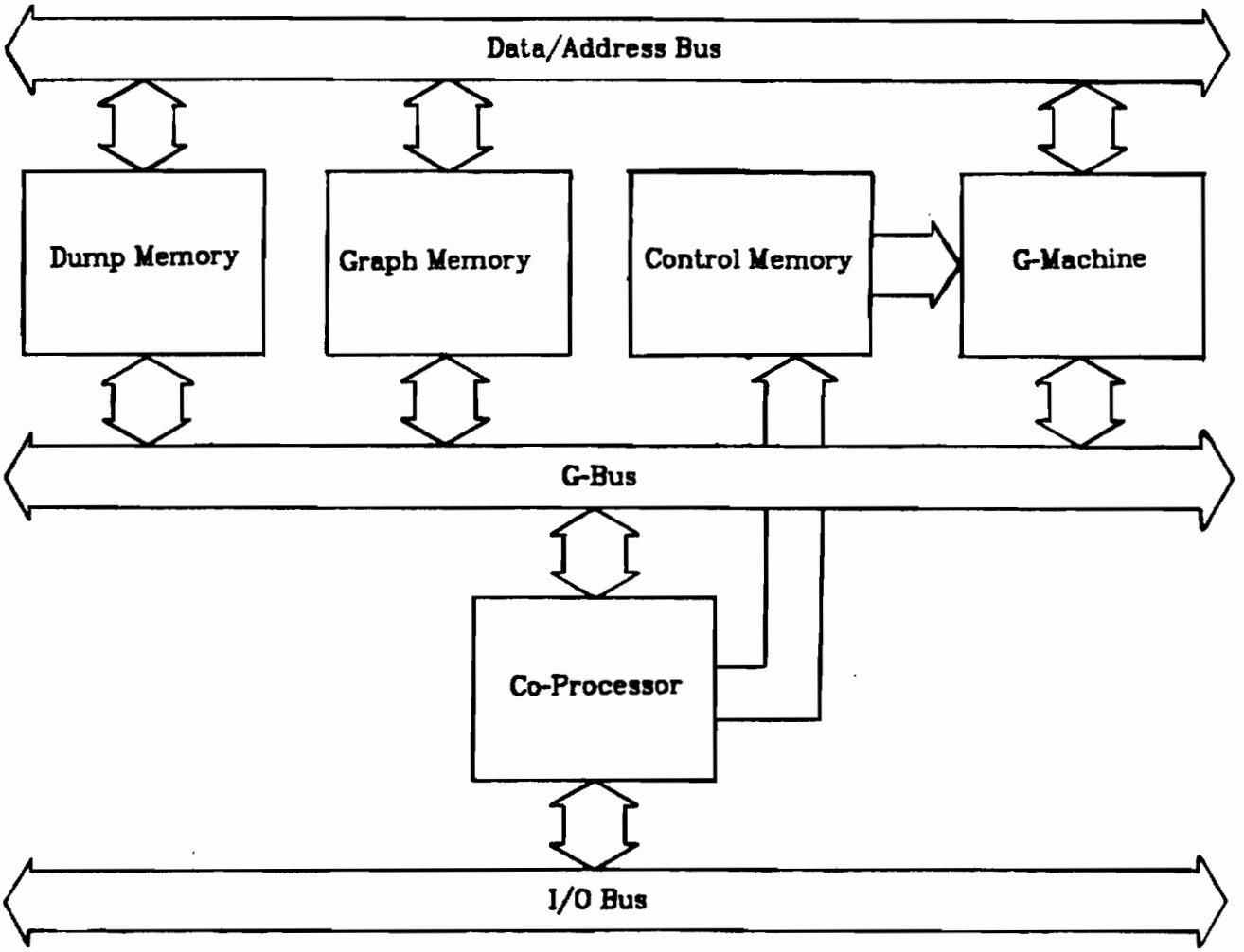


Figure 5.2 The G-machine system environment

6. THE G-MACHINE EXECUTION MODEL : A SIMPLE EXAMPLE

In the graph reduction compilation scheme, each function is translated into a sequence of G-machine instructions which reduces the graph of a function application to its value. Here we illustrate the G-machine execution states with the following program which returns a list of integers between two numbers:

```
letrec   count a b =
             if a > b then []
             else a . count (a + 1) b

in
       count 5 10
```

The G-machine state transitions for the execution of this program is shown in Figure 6.1. Before the start of the evaluation of the function application the pointer stack is configured as in Figure 6.1(a), where the stack top points to the function node 'count'. (The instruction EVAL, which evaluates a function application first checks that the stack contains all the actual parameters of the function before it performs a call. Which is a condition for the reduction to be possible). The G-code compiled for 'count' is:

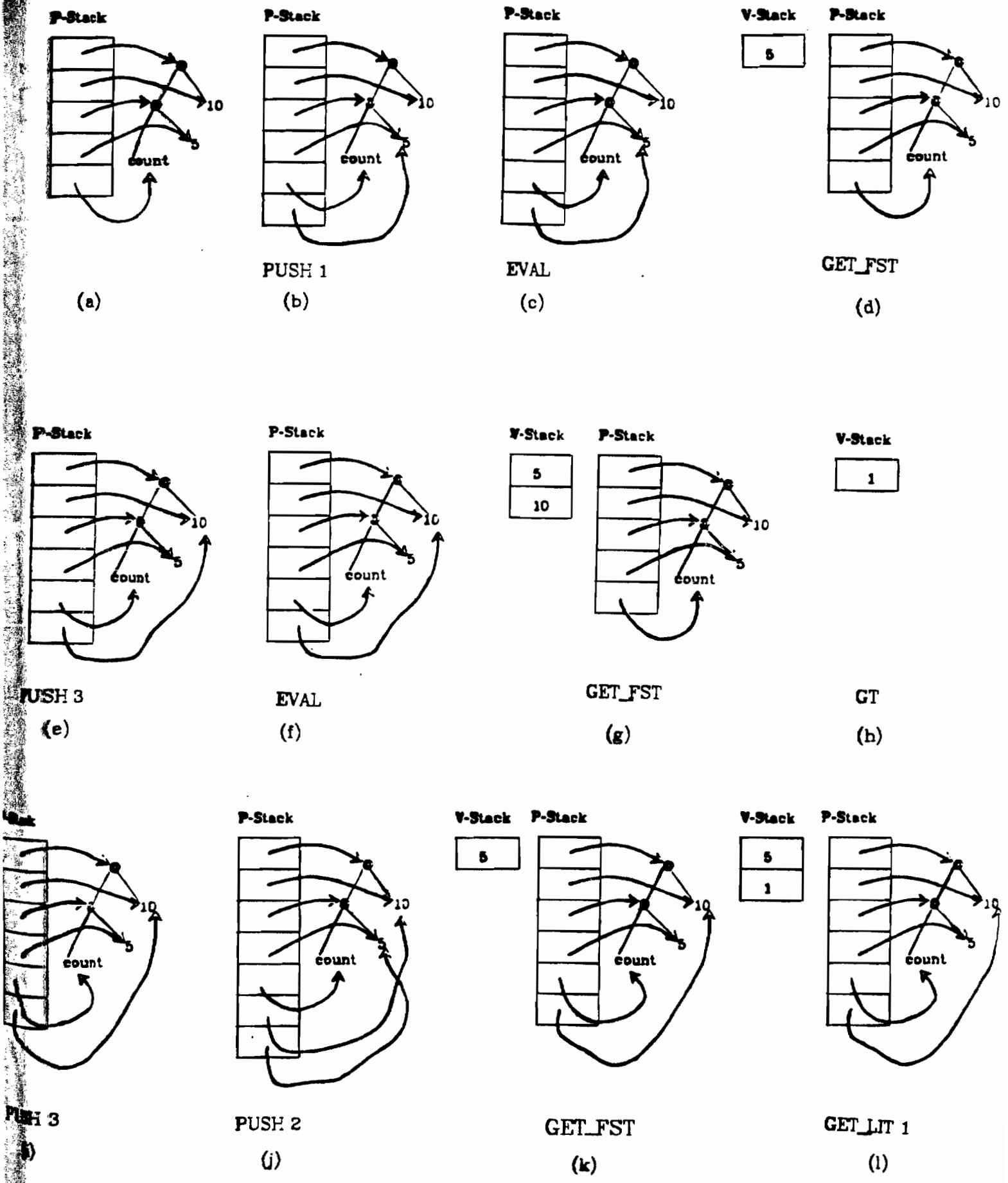
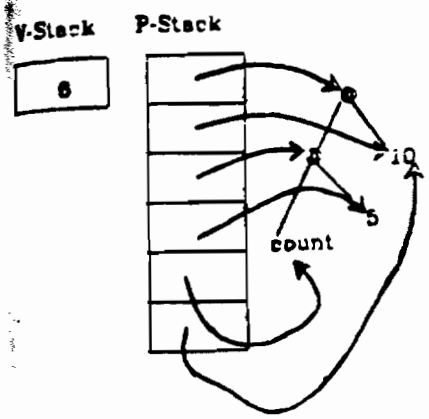
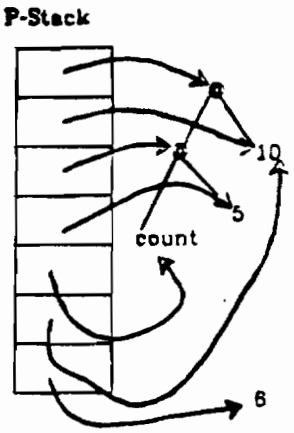


Figure 6.1 Pointer stack configurations for 'count 5 10'



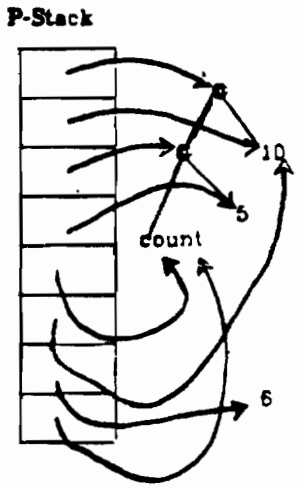
ADD

(m)



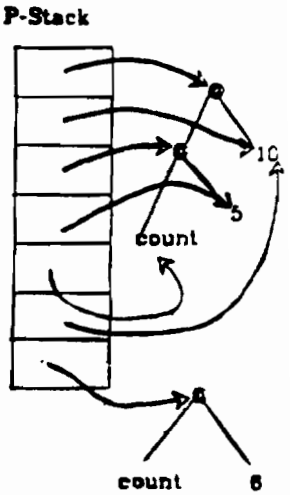
MK_VAL

(n)



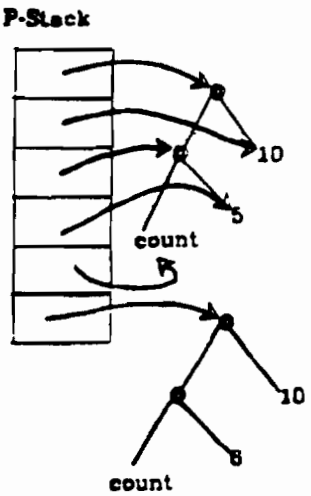
PUSHGLOBAL count

(o)



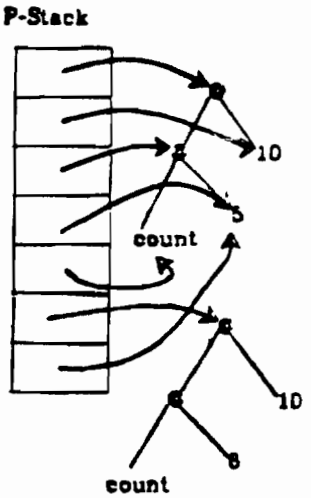
MK_AP

(p)



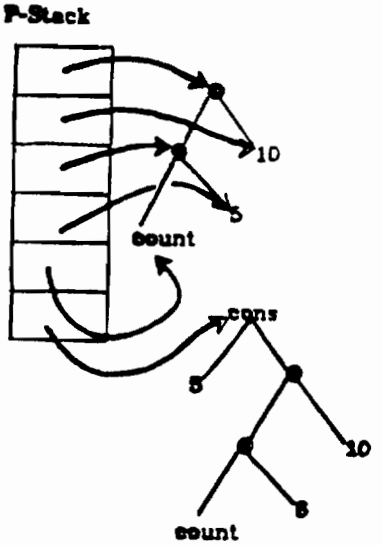
MK_AP

(q)



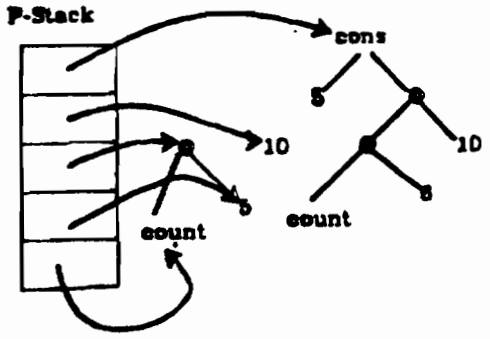
PUSH 2

(r)



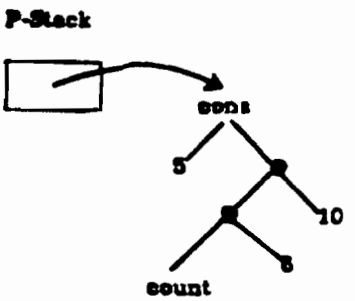
MK_PR

(s)



UPDATE 5

(t)



RETURN 4

(u)

```
count:  PUSH 1
        EVAL
        GET_FST
        PUSH 3
        EVAL
        GET_FST
        GT
        JFALSE label
        PUSH_NIL
        UPDATE 5
        RETURN 4
label:  PUSH 3
        PUSH 2
        GET_FST
        GET_LIT 1
        ADD
        MK_INT
        PUSHGLOBAL count
        MK_AP
        MK_AP
        PUSH 2
        MK_PR
        UPDATE 5
        RETURN 4
```

The purpose of the first few instructions before the JFALSE, which is a conditional jump instruction, are to obtain the arguments of the function 'count' into the value stack and to test for its range as defined in the source program. The PUSH m instruction pushes the m'th pointer of the stack, relative to the top (which is indexed with 0), onto the stack. The EVAL instruction evaluates the top of the stack into a canonical form. Since, by definition, the integers are always in canonical form, an immediate return from EVAL is performed in this case, which leaves the stack configuration unchanged, as shown in Figure 6.1(c). The

instructions GET_FST and GET_LIT are examples of data fetch instructions. The GET_FST instruction fetches a scalar value into the value stack from the storage cell pointed to by the top of the pointer stack and GET_LIT stores a literal value into the value stack from the instruction stream. The top two entries of the value stack are tested for greater than by the instruction GT, which leaves a boolean value on the value stack after the comparison is done; this is shown in Figure 6.1(h).

The instruction JFALSE, takes either true or false branch according to the boolean value present on the top of the value stack. In the false case (the state transitions given here deal only with the false case, assuming the true case can be dealt with accordingly), the instructions once again set the stack top to access the function arguments using PUSH instructions. The GET_FST instruction at this stage brings the first argument of the count into the value stack and the ADD instruction following the GET_LIT adds the top two entries of the value stack and leaves the result which is a new element of the list onto the top of value stack. The MK_VAL instruction creates a new node into the graph with the value obtained from the top of the value stack and leaves a pointer to this newly created node on the top of the pointer stack. The instruction PUSHGLOBAL 'count' pushes a pointer to the function 'count' onto the stack and the MK_AP constructs an application

node into the graph with the two topmost pointers from the pointer stack as its subparts. The Figure 6.1(o) shows the stack configuration after the PUSHGLOBAL instruction and Figure 6.1(p) shows the same after an application node is created.

The MK_PR instruction at this moment constructs a cons pair into the graph whose left part is in evaluated form and the right part yet to to be evaluated. This is shown in Figure 6.1(s). One thing we need to note here at this point is how the right part of the cons pair is structured after the MK_PR instruction. This graph is similar to our initial graph with the difference that one of the arguments of 'count' is the result of evaluating the expression. This illustrates that in our computation, expression graphs are evaluated (reduced to their values) at most once and all expressions that share a particular subexpression benefit from the evaluation.

After constructing the graph for the value of the right hand side in the definition of 'count', the cons node is copied to the root application node of the expression by the UPDATE instruction. This completes the evaluation of the function 'count' to its value. The RETURN instruction at this moment points to the newly constructed node, as shown in Figure 6.1(u) and returns control to the instruction following the call to the function 'count'.

The instruction that follows the function call is PRINT (it is not shown here to preserve clarity). The PRINT instruction is a complex

instruction. It is complex in that it trys to print the graph pointed to by the stack top and may cause a graph evaluation if the graph is not in a canonical form. The state transitions that were shown in Figure 6.1, is just before the PRINT can be invoked. Invoking PRINT on this graph will cause 5 to be printed and the right hand side of the cons pair will be traversed for further evaluations.

7. A NOVEL ARCHITECTURE

"A general trend in processor design today is to increase the complexity of architectures commensurate with the increasing potential of implementation technologies" [40]. This leads to increased design time, increased design errors and inconsistent implementations. The G-machine architecture attempts to show that with the proper architectural support for key aspects of the machine organization, that performance benefits can be obtained even with a simpler architecture.

The G-machine is actually a very simple architecture. Its novelty lies in the architectural-level support for graph traversal and update-in-place. This uses a pair of stacks: a pointer stack and a value stack. The pointer stack holds pointers to traverse the graph and provides a mechanism not only for procedure linkage and operand binding, but also for pointer manipulation and the allocation of a work area for the currently active function application. The value stack holds the intermediate arithmetic and logical values produced in the course of an evaluation and reduces the incidence of indirect reference to values that have recently been computed. The consequence is that the G-machine has relatively few instructions, a single addressing mode, and very few and very

short instruction operands.

In fact, the G-machine instructions are compiled code for a graph reduction that can be interpreted directly. These instructions allow a compiler to keep the code volume smaller for the G-machine architecture than for other architectures. This has a positive effect on execution speed because it reduces the amount of information that must be transmitted between storage and the processor.

Certain instructions in the G-machine have multiple execution phases. These are complex instructions which invoke other instructions during their execution (PRINT and EVAL are examples of such instructions). They may induce traversal of the graph to construct an environment for function application. It is this set of instructions which most differentiate the G-machine from other architectures.

In programming languages there is a sharp distinction between data and instructions. Unlike a conventional computing system, instructions and data in the G-machine are distinguished explicitly by storing them in different memories. The control memory in the G-machine holds only the instructions compiled from the LML program, and the graph memory holds a representation of the graph that is under evaluation, along with the necessary constants.

8. THE DESIGN OF A SIMULATOR FOR THE G-MACHINE

The simulator of the G-machine architecture was designed in order to investigate the performance and validity of the architecture before committing to a particular hardware implementation. The simulator is a register-transfer-level (RTL) behavioral description of the G-machine architecture and its components are written in ISP' [41] to be run on a VAX-11/780 under Berkeley UNIX 4.2.

The simulator provides a design environment that enables the user to

1) specify a hardware/software systems at multiple levels of abstraction;
2) test different algorithms in an interactive environment in order to evaluate architectural trade-offs and to study the algorithms for their functionality and performance.

The simulator consists of four highly-interrelated modules. They are respectively:

1) An ISP' description of the G-machine that decodes and executes instructions.
2) An assembler for the G-machine code that transforms mnemonic instructions to processor object code.

3) A software loader for the G-machine that loads assembler output into simulated memories.
4) The runtime environment, and various support and testing procedures for use during runtime.

The simulator reads an input file created by the assembler and executes the program, interpreting each instruction every time it is executed. The stacks, program counter, stack pointers and condition code flags as well as bus interface unit and bus protocols are all designed using simple data structures.

The first task in the design of the simulator was to define the type of programs to be simulated and the level at which they would be simulated. Ideally, the execution of a complete operating system with typical user processes should be simulated. This is not practical mainly for two reasons:

1) The ratio of simulation time to simulated time will simply take too much time and too many resources on the host computer.
2) An operating system for the G-machine is not intended to be designed as a part of this project.

Even though an operating system is not simulated, the simulator still deals with the operating system issues, such as input/output, dynamic memory allocation and storage management. The simulator we have developed handles this problem by

translating the I/O calls to another program, which in turn provides the interface between the simulator and the system and does the necessary I/O. The storage allocation was done statically in the simulator by maintaining an allocation register in an auto_increment fashion.

The simulator was designed not only to execute the programs, but also to obtain dynamic statistics and address traces from the execution of the programs. There are two places where statistics can be collected:

1) In the simulator itself.
2) In the trace analyzer, which reads the instructions and data address traces produced by the simulator.

The advantage of collecting statistics in the simulator itself is that it is faster than producing address traces and analyzing them by special programs. The disadvantage of collecting statistics in the simulator is that it clutters the interactive environment. Furthermore, when new types of statistics are desired, the simulator itself must be modified, thereby possibly introducing bugs.

Various statistical pieces of information can be collected from the simulator. They are respectively:

1) Maximum/Average depths of the value stack during program execution.

2) Maximum/Average depths of the pointer stack during program execution.

3) Number of JUMP instruction executed during a program execution.

4) Maximum length of a JUMP interval.

5) Minimum length of a JUMP interval.

6) Maximum number of function arguments in a function call.

7) Number of read accesses to the graph memory.

8) Number of write accesses to the graph memory.

9) Number of read accesses to the dump memory.

10) Number of write accesses to the dump memory.

11) Number of context save requests.

12) Number of context restore requests.

13) Amount of graph memory used during a program execution.

14) Amount of dump memory used during a program execution.

15) Total number of function calls present in a program.

16) The number of executions of an particular instruction.

17) Total timing involved in the program execution.

18) Total timing involved in instruction fetch.

Trace analyzers are simple to develop. As part of this project a trace analyzer which produces instruction frequencies and jump

lengths was developed [14]. This program reads the trace output from the simulator and produces a summary table of dynamic instruction frequencies and jump lengths.

The simulator and the trace analyzer were used to collect various statistics about the G-machine architecture. We summarize this information in the following section.

9. PERFORMANCE EVALUATION OF THE G-MACHINE ARCHITECTURE

As we outlined in previous sections, the novel features of the G-machine architecture are: an instruction set that supports graph traversal and update in place and a pair of stacks for holding the environment during the course of evaluation. Out of the many possible measures of the effectiveness of these features, this thesis only concentrates on performance (i.e., the time it takes to execute a given task), since high performance is the major goal of the G-machine architecture. The simulated execution time of a given task in the G-machine is based on some assumption about the execution time of each instruction. The assumption is that each memory access takes 4 clock cycles (since the implementation technology for the G-machine is not determined yet, an assumption on clock speed may not be correct at the present time), and all other instructions (except MUL, DIV and MOD)[2] take a single clock cycle for execution.

Ideally, each component of the G-machine architecture should be evaluated independently. Thus it would be interesting to measure the effectiveness of each instruction, and the number of times a graph is traversed to construct an environment and to reduce it.

---

[2]each of these instructions is assumed to take O(log N) time for execution, where N is the word

Since the G-machine architecture supports a pair of stacks in the processor itself, information about maximum stack depths will also be worthwhile to collect. The effectiveness of a proposed design can be deduced from data obtained by simulation of the G-machine architecture.

Simulation of three example programs has been performed using the simulator. Each of these programs has moderate code size and takes several thousand machine cycles for its execution. These programs were selected to exercise the execution of various instructions and to measure the complexity of constructing the environment during the execution.

The first two programs used in the simulation are quite simple. They are, respectively, a linear time algorithm to calculate the Nth Fibonacci number and an algorithm to compute Ackermann's exponential. The first one was selected to exercise the basic reduction model with recursive applications and the second one was chosen to evaluate the use of higher order functions. The third program is a Treesort program of moderate code size that sorts the elements of its input list in increasing order. Treesort was selected for its use of a recursively defined data type and to observe the dynamic growth and shrinkage (it constructs suspensions before evaluating expressions) of the graph that is under

size in bits.

evaluation.

The basis for the discussion in this section is the data from Table 9.1. All the G-machine data was obtained using the simulator and opcode trace analyzer [14]. The execution time for the programs that obtained from the simulator were compared with the VAX execution time. The instruction fetch time for the G-machine (from control memory with one byte at a time) counted 4 cycles are not overlapped with the execution. The VAX execution time was obtained using the *csh* **time** command. The number of data memory references and the sequence of instruction executions for the VAX has not been determined. Both optimized and unoptimized versions of the programs were run on the VAX.

The G-machine statistics presented in Table 9.1., are only a small part of the wealth of information produced by the simulator and trace analyzer. The interpretation of some of the items is not obvious: UNWIND is a phase of EVAL instruction that traverses a graph and constructs the proper environment (by pushing pointers to the argument and function parts of each application node) on the pointer stack before the evaluation takes place. The RETURN instruction performs a return from EVAL. The number of executions of the PRINT instruction was collected to observe how many leaves of the list in the data structure were evaluated. When the value is a list, the head element and the tail list are in turn

evaluated and printed. The number of PUSH instructions executed by the program was collected to observe the need for hardware support for stacks. (Recall that PUSH is a stack-to-stack movement of a single datum.)

Table 9.1.

Summary of Dynamic Statistics			
Item	Fib(30)	Ack(2,16)	Treesort[..]
Number of JUMP instr.	32	1276	502
Maximum JUMP interval	97	122	219
Minimum JUMP interval	5	5	5
# of read accesses to graph	1962	7437	12153
# of write accesses to graph	844	2766	4762
# of context save/restore requests	152	306	927
Total # of Dump read/write	1200	2765	7003
Number of function calls	1	1	19
Number of PUSH instructions	336	2195	2230
Number of EVAL instructions	271	1258	2054
Number of RETURN instructions	152	306	927
Number of PRINT instructions	3	1	51
Number of UNWIND phase present	150	305	908
Maximum depth of Pointer stack	9	7	11
Maximum depth of Value stack	2	2	2
Total # of allocations	301	1544	1472
Instruction fetch count (byte at a time)	2406	16975	15521
Total # of instr. simulated	1766	13520	10655
% of context-switch over total instr.	17.2	4.5	17.4
Code size (# of instructions)	94	51	297
Number of VAX instr. (optimized)	136	76	448
Execution time (# of cycles)	36175	196615	216989
VAX execution time (msecs)	500	700	700

The performance of the G-machine architecture is enhanced by providing hardware support operations on stacks. To determine what instructions are used most frequently, we counted the executions of each instruction during a program execution. Data collected from the simulated programs are shown in Table 9.2.

Table 9.2.

Dynamic Percentage of Instruction Executions			
Instruction	Fib(30)	Ack(2,16)	Treesort[..]
EVAL	15.3	9.3	19.3
UNWIND phase	8.5	2.2	8.5
UPDATE	8.5	2.2	8.7
RETURN	8.5	2.2	8.7
PUSH	19.0	16.2	21.0
PUSHGLOBAL	8.7	2.4	8.8
MOVE	0.0	4.8	0.0
MK_AP	11.9	4.5	10.2
MK_PAIR	1.7	0.0	2.2
MK_VAL	3.4	6.9	0.0
GET_FST	6.8	14.0	1.7
GET_LIT	3.4	14.0	0.0
EQ	1.7	7.0	0.0
JUMP	1.8	9.4	4.7
FST	3.4	0.0	2.2
SND	1.7	0.0	3.0
IS	0.0	0.0	3.1
OUT	0.0	0.0	2.7
INJECT	0.0	0.0	1.2
ADD	1.7	2.3	0.0
SUB	1.7	4.6	0.0

The most important observation from Table 9.2 is that the instructions that operate only on the pointer stack without reference to G-memory (PUSH and MOVE) are quite significant (about 20 percent of the total number of instructions executed). The instructions that appeared most frequently after the stack configuring instruction are those which operate on the stack and also require access to storage. These instructions include graph traversal, storage allocation, data fetch and data structure selector instructions (such as UPDATE, MK_AP, MK_VAL, GET_FST, FST, PUSHGLOBAL and the UNWIND phase of EVAL). This information suggests that a cost-effective system can be designed for efficient execution of the G-machine code by providing a hardware address cache (i.e., the pointer stack) in the processor itself, and it is the reason why the architecture that we simulated provides direct support for the pointer stack.

In order to make meaningful assessment of an architecture that is under design it is important that all of the variables that affect performance are properly addressed. These include the clock speed of the system and the number of memory system wait states. As we outlined before, since the G-machine architecture is not refined yet for an implementation technology, any assessment on the clock speed will not be fair at this time.

One of the most expensive operations in the execution of an instruction in any computing system is the time spent for memory access. The G-machine instructions (excepting the instructions that support graph traversal and context switching) and their respective number of memory references are summarized in Appendix B. Here we include in Table 9.3., the data obtained from the simulator showing the percentage of the execution time that was spent to access storage in our example programs.

Table 9.3.

Ratio of Storage Access Time to Execution Time (Percent)			
Memory Traffic	Fib(30)	Ack(2,16)	Treesort[..]
Read accesses to graph	21.7	15.1	22.4
Write accesses to graph	9.3	5.6	8.8
Total # of dump read/write	13.2	5.6	13.0

It has been suggested that context-switching can be the most time consuming operation in executing high level language programs [42,43,44]. In our example programs, the percentage of context-switching requests over the total number of instructions simulated are 4.5-18%. The definition of the G-machine does not provide architecture-level support for context-switching. Context-switching is an expensive operation in the G-machine as it is in most architectures. Performance benefits can be certainly obtained by providing appropriate hardware support for context-

switching.

A possible solution is to include a cache store in the processor itself. The addition of a cache will make any architecture run faster, but a cache is ineffective if it is too small. An effective cache would require a much larger area and complicated control mechanism in the processor. Another possible solution could be to use the strategy of overlapped register windows in hardware, which the designers of the Berkeley RISC-I processor [45] found very impressive. The RISC-I architecture as implemented is tailored to the execution of imperative language programs (such as Pascal or C) and not towards the efficient execution of functional language programs based on recursive function calls. The principal diseconomy of this model is that the register file overflow will become so overwhelming in recursive function applications that the efficient execution of the programs will not be possible without maintaining a online pool of registers that is freely available.

A much simpler solution can be used to provide the hardware support for context-switching in the G-machine architecture by following the traditional architectural support used in ALGOL-like languages. This includes Dijkstra's "display-technique" and maintaining a runtime stack in some place in storage. But maintaining a runtime stack in memory will not certainly improve the performance of the G-machine architecture, since a large number of

instructions in the program execution either directly manipulate the contents of the pointer stack or access it for indirect reference to the graph memory. Thus it would require a larger number of storage accesses and will not meet the desired performance goals.[3]

Solutions that are not practical to implement in hardware to achieve performance benefits can sometimes be incorporated in software. One of the ways context-switching can be minimized in the G-machine program execution is by having a compiler recognize tail-recursive function applications and translating them into iterative code. This eliminates the need for a context-switch at a tail-recursive call, instead requiring only the modest instruction scheduling overhead of a jump instruction.

The present LML compiler as implemented already recognizes tail-recursive applications in the source code, and generates an efficient, iterative evaluation sequence. However, it does not attempt program transformations to generate tail-recursive applications from a source program that was not written in that form. Since none of our example programs uses tail recursive function calls, they do not take advantage of this feature.

In our definition of the G-machine architecture, we justified the need for architecture-level support for the stacks in the processor

[3]Though this idea has not been simulated but using the data from the simulator we find that execution time increases by about 30% (Fib: 36.6%, Ack: 27.8% and Treesort: 33.6%) if we use this concept.

itself. Thus it will be interesting to know if we can bound the depth of the stack during the graph traversal and reduction of function applications. This will allow the use of a hardware supported pointer stack without an automatic overflow mechanism. Keeping the stack depth to a minimum will improve the performance of the G-machine program execution by having to store only a minimum amount of pointer stack contents during a context-switch. This can be readily done during the UNWIND phase of the G-machine execution of an EVAL instruction. An example of the existing UNWIND phase and the new scheme are shown in Figures 9.1 and 9.2. As shown in Figure 9.2, the new scheme differs from the old one by having only to push a pointer to the root application node of the graph onto the stack and then just pushing a pointer to the argument part of each application node rather than pushing new pointers to the function and argument parts of each application node. This is semantically correct because it is only the root application node that is updated with the result when the graph is reduced to some canonical value. Thus it is not necessary to create an environment in the pointer stack with redundant pointers to the successive chain of application nodes for simply applying a function to its arguments.

We have simulated the proposed UNWIND scheme during the execution of our example programs. As expected the results we

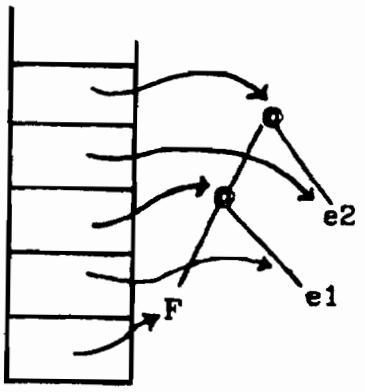


Figure 9.1 Arrangement of the stack after UNWIND (existing scheme) on the application of function 'F' on two arguments

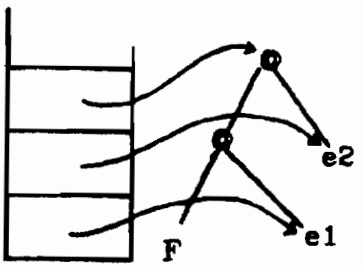


Figure 9.2 Arrangement of the stack after UNWIND (new scheme)

obtained show a significant performance improvement by having only to save/restore a relatively small amount of pointer stack contents during each context-switch. The data we obtained during the simulation of new UNWIND scheme are shown in Table 9.4., and should be compared with the data of Table 9.3.

Table 9.4.

Performance Characteristics of New UNWIND Scheme			
Item	Fib(30)	Ack(2,16)	Treesort[..]
Total # of dump read/write	992	2151	6139
Percentage of execution time	11.0	4.4	11.3

During the simulation of our example programs we compared its code size with the VAX codes (a successful modern minicomputer) to measure the effectiveness of each G-machine instruction (as shown in Table 9.1). Clearly our observations on these features of the G-machine architecture is very encouraging and we indicate that the G-machine architecture is a good choice for a functional language evaluator.

10. CONCLUSION

An experimental investigation of a graph reduction machine architecture has been described. A simulator for the G-machine architecture, capable of simulating the execution of "real" programs and obtaining dynamic statistics, has been developed. This simulator was used to demonstrate the effectiveness of the G-machine architecture and its instruction set.

The abstract model of computation used in the LML compiler has been studied. We have identified many potential advantages of the LML compiler for the G-machine architecture [46]. The values of expressions are shared among multiple references. In fact, because of the lazy evaluation strategy, an applicative expression is evaluated at most once. Dynamic environments are avoided by anticipating all references to non-local variables, and binding exactly the variables needed as arguments in applicative expressions. Some economy in control flow is achieved by recognizing tail recursive function calls, and thereby avoiding chains of barren returns from iterative function calls.

The LML compiler as it is implemented presently does not generate very efficient code. It requires more memory to build an application tree for applicative expression than it does to evaluate

the expression straight away. Thus the lazy evaluation strategy is wasteful in those cases in which an applicative expression is suspended at first, but eventually evaluated. Less memory probably would have been used if it were evaluated immediately.

When an expression is evaluated, it is almost always represented by a newly allocated node in G-memory. Often, this node is then copied to overwrite an application node in an UPDATE operation. If only the copy is wanted of this newly allocated node, allocation of this new node could have been avoided.

Similarly, when a sequence of tail-recursive calls is evaluated, the values computed in all but the final call of the sequence are only used as arguments to the succeeding call in the sequence. However, new storage is allocated in G-memory to represent each of these intermediate values. If this specialized use were recognized, the new values could just overwrite the storage previously allocated for arguments, avoiding a sequence of unnecessary storage allocations.

Another possible source of optimization in the use of storage is to recognize when new basic values (arithmetic and boolean) are used immediately as arguments of arithmetic or boolean expressions, or tested by an "if". Such values need not be allocated for storage at all, but may be retained in the V-stack until used. For repeated uses of the same operand, a more sophisticated

algorithm is required. This strategy may also be extended to values produced by function applications.

It is extremely important for performance to be able to take advantage of these opportunities to avoid unnecessary storage allocations. Performance is degraded in three ways by the profligate use of storage. First, there are extra instructions executed to perform allocations. Second, there is either an eventual loss of efficiency due to storage fragmentation, if a program is run in a large virtual address space without the use of garbage collection, or there is time lost in collecting cells to prevent this fragmentation and restore locality. The overhead of collection will be at least proportional to the total number of cells allocated (unless collection is done by a parallel processor). Finally, we have seen that there are optimizations of the function-call protocol which are available when the compiler implements an application of a globally-defined function. These optimizations are not possible when the call is produced by UNWINDing a previously constructed application graph, which is the case when a suspension is finally evaluated.

The costs of a computation are identified in terms of a number of accesses to a data structure whose size is potentially unbounded. Much of the future research should concentrate to provide effective means to improve program locality. The results

obtained from the simulator show how some such improvements were obtained by providing hardware stacks for graph traversal and to hold local environments during evaluations.

It is also expected that higher performance benefits can be obtained in the execution of the G-machine code that supports the new UNWIND scheme. Some of our future research will be to incorporate these ideas into a new G-machine compiler that will produce highly optimized codes and will reduce redundant allocations.

Performance benefits can also be improved by an order of magnitude if various architectural units of the G-machine can be run in parallel. Such as instruction pre-fetch and execution can be overlapped with the sequentiality of the machine state if a separate bus is provided between the processor and the control memory.

Further studies on various architectural issues can be easily performed using the simulator developed. The operating system issues have not been addressed in this thesis. Once the operating system features are finalized they should be incorporated in the simulator so that some measures of the new computing environment could be obtained.

REFERENCES

[1] Mead, C.A. and Conway, L.A., "Introduction to VLSI Systems". Addison-Wesley, Reading, Massachusetts., 1980.

[2] Bashkow, T.R., Kroft, D. and Sasson, A., "Study of a Computer for Direct Execution of List Processing Language", Tech. report 103, Columbia University, January 1968, AFCRL-68-0063.

[3] Chu, Y., and Abrams, M., "Programming Languages and Direct-Execution Computer Architecture", Computer, Vol. 14, No. 7, July 1981, pp. 22-32.

[4] Ditzel, D.R., "Reflections on the High-Level Language Symbol Computer System", Computer, Vol. 14, No. 7, July 1981, pp. 55-66.

[5] Flynn, M.J., "Directions and Issues in Architecture and Language", Computer, Vol. 13, No. 10, October 1980, pp. 5-22.

[6] Johnsson, Thomas, "The G-Machine: An Abstract Machine for Graph Reduction", Programming Methodology Group, Dept. of Computer Science, Chalmers Univ. of Technology, Gothenburg, Sweden, August 1983.

[7] Turner, D.A., "A New Implementation Technique for Applicative Languages", Software - Practice and Experience, Vol. 9, September 1979, pp. 31-49.

[8] Keller, R.M, Lindstorm G. and Patil S., "A Loosely-Coupled Applicative Multi-processing System", Proc. of NCC. AFIPS Press, June 1979, pp. 613-622.

[9] Keller, R.M., Lin, F.C.H, and Tanaka, J., "The Rediflow Multiprocessing System", Dept of Computer Science, Univ. of Utah, 1983.

[10] Clarke, T.J.W., Gladstone P.J.S., Maclean C.D., and Norman, A.C., "SKIM - The S, K, I Reduction Machine", Proc. 1980 LISP Conference, August 1980, pp. 128-135.

[11] Augustsson, L., "Functional Compiler Status Report #1", Memo 24, Programming Methodology Group, Dept. of Computer Science, Chalmers Univ. of Technology, Gothenburg, Sweden., 1982.

[12] Augustsson, L., and Olofsson, G., "A Peephole Optimizer", Programming Methodology Group, Dept. of Computer Science, Chalmers Univ. of Technology, Gothenburg, Sweden., 1982.

[13] Vireday, Richard, "An Assembler for the G-machine", Dept. of Computer Science & Engineering, The Oregon Graduate Center. February 1984.

[14] Foster, Mark, "Instruction Trace Analyzer for the G-machine", Dept. of Computer Science & Engineering, The Oregon Graduate Center. March 1984.

[15] Backus, John, "Can Programming be Liberated from the Von Neumann Style? A Functional Style and its Algebra of Programs", Comm. A.C.M., August 1978, pp. 613-641.

[16] Church, A., "The Calculi of Lambda Conversion", Annals of Mathematical Studies, Princeton University Press, 1941.

[17] Stoy, J.E., "Denotational Semantics", MIT press, 1977.

[18] Landin, P.J., "The Mechanical Evaluation of Expressions", Computer Journal, June 1964, pp. 308-320.

[19] Clark, D.W., "An Empirical Study of List Structure in LISP", Comm.A.C. M., February 1977, pp. 78-87.

[20] Berkling, K.J., "Reduction Languages for Reduction Machines", Interner Bericht ISF-76-08, Gesellschaft fur Mathematik und Datenverarbeitung mbH Bonn, 1976.

[21] Kluge, W.E., "The Architecture of A Reduction Language Machine Hardware Model", Internal Rep. ISF-GMD-79.03, D-5205 St. Augustin 1, August 1979.

[22] Berkling, K.J., "Experiences With Integrating Parts of the GMD-Reduction-Language Machine, VLSI Architecture, Prentice-Hall, Englewood Cliffs, New Jersey, 1983.

[23] Mago, G.A., "A Cellular Computer Architecture for Functional Programming", Proc. IEEE COMPCON 80, February 1980, pp. 179-187.

[24] Treleaven P.C., and Mole G.F., "A Multi-processor Reduction Machine for User-defined Reduction Languages". Proc. Seventh Int. Symp. on Computer Architecture, May 1980. pp. 121-130.

[25] Hughes, R.J.M., "Super Combinators: A New Implementation Method for Applicative Languages", Proc. 1982 A.C.M. Symp. on Lisp and Functional Languages, August 1982, pp. 1-10.

[26] Darlington J., and Reeve M., "A Multi-processor Reduction Machine for the Parallel Evaluation of Applicative Languages", Proc. of Conf. on Functional Programming Languages and Computer Architecture, October 1981.

[27] Curry, H.B., and Feys, R., "Combinatory Logic", North-Holland, Amsterdam, 1958.

[28] Henderson, P., "Functional Programming -- its Implementation and Application", Prentice-Hall 1980.

[29] Jones, S.L.P., "An Investigation of the Relative Efficiencies of Combinators and Lambda Expressions", Proc. of 1982 A.C.M. Symp. on Lisp and Functional Languages, August 1982, pp. 150-158.

[30] Hughes, R.J.M., "The Design and Implementation of Programming Languages", Ph.D. thesis, Oxford University, July 1983.

[31] Gordon, M.J.C., Milner, A.J.R., and Wadsworth, C.P., "Edinburgh LCF, Lecture Notes in Computer Science" No. 78, Springer-Verlag, 1979.

[32] Schonfinkel, M., "Uber Die Bausteine Der Mathematischen Logik", Mathematische Annalen 92:305, 1924.

[33] Kieburtz, R.B., "LML User Guide", CSE 511 Class Notes, Dept. of Computer Science & Engineering, The Oregon Graduate Center, January 1984.

[34] Hoevel, L.W., "IDEAL–Directly Executable Languages, An Analytical Argument for Emulation", IEEE Transactions on Computer, Vol C-23, August 1974.

[35] Schoellkopf, J.P., "A Tutorial on High Level Language Machine for PASCAL", ENS-IMAG Report 131-Grenoble, October 1978.

[36] Wilner, W.T., "Design of the Burroughs B1700," Proceedings of the 1972 Fall Joint Computer Conference, Montvale, NJ: AFIPS, 1972, pp. 489-497.

[37] Sansonnet, M.J., Castan, M., and Percebois, C., "M3L: A List-Directed Architecture", International Symposium on Computer Architecture, La Baule, France, May 1980, pp. 105-112.

[38] Ditzel, D.R., and Patterson, D.A., "Retrospective on High-Level Language Computer Architecture", Proceedings of the International Workshop on High-Level Language Computer Architecture, New York: ACM, April 1980, pp. 97-104.

[39] Kieburtz, R.B., "The G-Machine: A fast Graph-Reduction Processor", Dept. of Computer Science & Engineering, The Oregon Graduate Center, November 1983.

[40] Patterson, D.A., and Sequin, C.H., "RISC I: A Reduced Instruction Set VLSI Computer", Eighth Annual Symp. on Computer Architecture, May 1981, pp. 443-457.

[41] Rose, C.W., and Ordy, G.M., "N.mPC: A Retrospective", Proceedings of 20th. Design Automation Conference., IEEE, June 1983, pp. 497-514.

[42] Knuth, D.E., "An Empirical Study of FORTRAN Programs", Software -- Practice & Experience, Vol. 1, No. 2, 1971, pp. 105-134.

[43] Robinson, S.K., and Torsun, I.S., "An Empirical Analysis of FORTRAN Programs", The Computer Journal, Vol. 19, No. 1, pp. 56-62.

[44] Shustek, L.J., "Analysis and Performance of Computer Instruction Sets", PhD dissertation, Stanford University, 1978.

[45] Patterson, D.A., and Sequin, C.H., "A VLSI RISC", Comuter, September 1982, pp. 8-21.

[46] Kieburtz, R.B., "Reducing Memory Allocations for G-Machine", CSE 512 Class Notes, Dept. of Computer Science & Engineering, The Oregon Graduate Center, May 1984.

APPENDIX A

SPECIFICATION OF THE G-MACHINE INSTRUCTIONS

The following are the definitions of the G-machine instructions. The action of each instruction is defined as a transformation of a six-tuple,

<C, P, V, G, E, D>

in which C is the control, P and V are the traversal and value stacks respectively, G is the expression graph, D the dump, and E the environment. Sequences are represented using a dot for the sequence constructor, and () denotes the null sequence. The majority of the instructions are taken directly from [6]. Some instructions have been added to allow value-pairs to be handled without indirection. Less attention has been paid to the arithmetic and logical group than to other instructions; these will no doubt be refined somewhat before a design is committed for implementation.

- EVAL -

<EVAL.C, n.P, V, [n: AP n_1 n_2]+G, E, D> ==>
<UNWIND.(), n.(), (), [n: AP n_1 n_2]+G, E, (C.P,V).D>

<EVAL.C, n.P, V, [n: *OP operands*]+G, E, D> ==>
<C, n.P, V, [n: *OP operands*]+G, E, D>
at a node for which *OP* is not AP.

- UNWIND -

{when UNWIND is executed, the active node must be an AP or a FUN}

<UNWIND.(), n.P, V, [n: AP n_1 n_2]+G, E, D> ==>
<UNWIND.(), n_1.n_2.n.P, [n: AP n_1 n_2]+G, E, D>

<UNWIND.(), n.n_1 ... n_k.(), V, [n: FUN f]+G, [f: (a, C')]+E, (C'',P'',V'').D> ==>
k >= 2*a+1 : <C', n.n_1 ... n_k.() V, [n: FUN f]+G, [f: (a,C')]+E, (C'',P'',V'').D>
k < 2*a+1 : <C'', n_k.P'', V'', [n: FUN f]+G, [f: (a,C')]+E, D>
{reduction of an application occurs only when the requisite number of arguments is present}

- CALLGLOBFUN -

<CALLGLOBFUN f.(),.P, V, G, [f: C']+E, D> ==> <C',.P, V, G, [f: C']+E, (C,P,V).D>

- RETURN -

<RETURN m.C, n_0 ... n_m.(), (), [n_m: *OP operands*]+G, E, (C',S',V').D> ==>
<C', n_m.P', V', [n_m: *OP operands*]+G, E, D>
{the node at the bottom of the stack should have been evaluated
before a RETURN is executed}

- PRINT -

<PRINT.C, n.P, V, [n: CONS n_1 n_2]+G, E, D> ==>
<PUSH 0.FST.EVAL.PRINT.SND.EVAL.PRINT.C, n.P, V, [n: CONS n_1 n_2]+G, E, D>

<PRINT.C, n.P, V, [n: *OP operands*]+G, E, D> ==>
<C,.P, V, [*OP operands*]+G, E, D>
if *OP* is not CONS.

- instructions to support graph traversal -

<PUSH m.C, n_0 ... n_m.P, V, G, E, D> ==> <C, n_m.n_0 ... n_m.P, V, G, E, D>

<PUSH_NIL.C,.P, V, G, E, D> ==> <C, nil.P, V, G, E, D>

<PUSHGLOBAL n.C, P, V, G, E, D> ==> <C, n'.P, V, [n': n]+G, E, D>

<MOVE 1.C, n_0.n_1.P, V, G, E, D> ==> <C, n_0.P, V, G, E, D>

<MOVE m.C, n_0.n_1 ... n_m.P, V, G, E, D> ==> <C, n_1 ... n_{m-1}.n_0..P, V, G, E, D>
when m > 1

<SLIDE m.C, n_0 ... n_m.P, V, G, E, D> ==> <C, n_0.P, V, G, E, D>

<ROT m.C, n_1 ... n_{m-1}.n_m.P, V, G, E, D> ==> <C, n_m.n_1.n_{m-1}.P, V, G, E, D>

<POP m.C, n_0 ... n_m.P, V, G, E, D> ==> <C, P, V, G, E, D>

- UPDATE -

{this is the sole operation that overwrites a node value in the graph}

<UPDATE m.C, n_0.n_1 ... n_m.P, V, [n_0: N_0, n_m: N_m]+G, E, D> ==>
<C, n_1 ... n_m.P, V, [n_m: N_0]+G, E, D>

- Node allocation instructions -

{in which n', n_1', and n_2' represent new unique node names}

<MK_LIT i.C,.P, V, G, E, D> ==> <C, n'.P, V, [n': CONST i]+G, E, D>

{inserts a literal value into the graph}

<MK_VAL.C,.P, i.V, G, E, D> ==> <C, n'.P, V, [n': CONST i]+G, E, D>

<MK_AP.C, n_1.n_2.P, V, G, E, D> ==> <C, n'.P, V, [n': AP n_1 n_2]+G, E, D>

<MK_PR.C, n_1.n_2.P, V, G, E, D> ==> <C, n'.P, V, [CONS n_1 n_2]+G, E, D>

<MK_VAL_PR.C,.P, i_1.i_2.V, G, E, D> ==> <C, n'.P, V, [n': PR i_1,i_2]+G, E, D>

<MK_V1_PR.C, n.P, i.V, G, E, D> ==> <C, n'.P, V, [n': H-PR1 i n]+G, E, D>

<MK_V2_PR.C, n.P, i.V, G, E, D> ==> <C, n'.P, V, [n': H-PR2 n i]+G, E, D>

<ALLOC m.C,.P, V, G, E, D> ==> <C, n_1' ... n_m'.P, [n_1: HOLE, ...,n_m':HOLE]+G, E, D>

- data fetch instructions -

<GET_FST.C, n.P, V, [n: PR i_1 i_2]+G, E, D> ==>
<C,.P, i_1.V, [n: PR i_1 i_2]+G, E, D>
{the node may also be a H-PR1}

<GET_SND.C, n.P, V, [n: PR i_1 i_2]+G, E, D> ==>
<C,.P, i_2.V, [n: PR i_1 i_2]+G, E, D>
{the node may also be a H-PR2}

<GET_PR.C, n.P, V, [n: PR i_1 i_2]+G, E, D> ==>
<C,.P, i_1.i_2.V, [n: PR i_1 i_2]+G, E, D>

<GET_LIT i.C,.P, V, G, E, C> ==> <C,.P, i.V, G, E, D>

<GET_ZERO.C,.P, V, G, E, D> ==> <C,.P, 0.V, G, E, D>

- data structure selection -

<FST.C, n.P, V, [n: H-PR2 n_1 i_2]+G, E, D> ==> <C, n_1.P, V, [n: H-PR2 n_1 i_2]+G, E, D>

<SND.C, n.P, V, [n: H-PR1 i_1 n_2]+G, E, D> ==> <C, n_2.P, V, [n: H-PR1 i_1 n_2]+G, E, D>

<FST.C, n.P, V, [n: CONS n_1 n_2]+G, E, D> ==> <C, n_1.P,.V, [n: CONS n_1 n_2]+G, E, D>

<SND.C, n.P, V, [n: CONS n_1 n_2]+G, E, D> ==> <C, n_2.P, V, [n: CONS n_1 n_2]+G, E, D>

- test for a null list -

<NULL.C, n.P, V, [n: CONS n_1 n_2]+G, E, D> ==>
<C,.P, false.V, [n: CONS n_1 n_2]+G, E, D>

<NULL.C, n.P, V, [n: NIL]+G, E, D> ==> <C,.P, true.V, [n: NIL]+G, E, D>

- arithmetic and logical operations[1] -

<*binop*.C,.P, i_1.i_2.V, G, E, D> ==> <C,.P, (i_1 *binop* i_2).V, G, E, D>
where *binop* is one of ADD, SUB, MUL, DIV, EQ, NE, LT, GT, LE, GE.

<INCR.C,.P, i.V, G, E, D> ==> <C,.P, (i+1).V, G, E, D>

<NEG.C,.P, i.V, G, E, D> ==> <C,.P, (-i).V, G, E, D>

<NOT.C,.P, b.V, G, E, D> ==> <C,.P, (not b).V, G, E, D>

- control transfer instructions -

<JFALSE l.C,.P, true.V, G, E, D> ==> <C,.P, V, G, E, D>

<JFALSE l ... LABEL l.C,.P, false.V, G, E, D> ==> <C,.P, V, G, E, D>

<LABEL l.C,.P, V, G, E, D> ==> <C,.P, V, G, E, D>

<JFUN f.(),.P, V, G, [f: C']+E, D> ==> <C',.P, V, G, [f: C']+E, D>

---

[1] Condition codes (zero, negative, carry, overflow) are to be added.

- instructions to support abstract data types -

<lS n.C, i.P, V, [i: H_PR1 m d]+G, E, D> ==> <C, P, (m = n).V, G, E, D>

<OUT n.C, i.P, V, [i: H_PR1 n d]+G, E, D> ==> <C, d.P, V, G, E, D>

<OUT n.C, i.P, V, [i: H_PR1 m d]+G, E, D> ==> <fail.C, d.P, (m <> n).V, G, E, D>

<INJECT *t*.C, i.P, V, [i: H_PR1 m d]+G, E, D> ==> <C, j.P, V, [j: H_PR1 *t* i]+G, E, D>

{*t* is the type value}

APPENDIX B

EFFECTIVE GRAPH MEMORY REFERENCES OF THE G-MACHINE INSTRUCTIONS

The following are the G-Machine instructions that access graph memory. Each instruction is given here with the total number of graph memory references. The information in parenthesis describes the number of read and write to the memory. It is given in the form (R/W), where "R" is the number of read access and "W" is the number of write access.

Instruction	No. of G-Memory ref.	Remark
EVAL	1 (1/0)	This is true only if the node is in canonical form
RETURN	1 (1/0)	-
UPDATE	2 (1/1)	It is 4 (2/2) for pair
UNWIND	2 (2/0)	For each application node
MK_LIT	1 (0/1)	-
MK_VAL	1 (0/1)	-

Instruction	No. of G-Memory ref.	Remark
MK_AP	2 (0/2)	-
MK_PAIR	2 (0/2)	-
MK_V1_PR	2 (0/2)	-
MK_V2_PR	2 (0/2)	-
MK_VAL_PAIR	2 (0/2)	-
GET_FST	1 (1/0)	-
GET_SND	1 (1/0)	-
GET_PAIR	2 (2/0)	-
FST	1 (1/0)	-
SND	1 (1/0)	-
IS	1 (1/0)	-
OUT	2 (2/0)	-
INJECT	2 (0/2)	-