A RISC Architecture for Symbolic Computation

Richard B. Kieburtz

Oregon Graduate Institute
Department of Computer Science and Engineering
20000 NW Walker Rd
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 87-001
February 1987

A RISC Architecture for Symbolic Computation

Richard B. Kieburtz

Oregon Graduate Institute
Department of Computer Science and Engineering
20000 NW Walker Rd
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E-87-001
February 6, 1987

A RISC ARCHITECTURE FOR SYMBOLIC COMPUTATION

Richard B. Kieburtz

Oregon Graduate Center
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 87-001

A RISC architecture for symbolic computation

Richard B. Kieburtz

Oregon Graduate Center
19600 N.W. von Neumann Dr.
Beaverton, OR 97006

1. Introduction

During the next five to ten years, we can expect to see symbolic computation experience the sort of explosive growth that has occurred in numerical computation in the past ten years of the supercomputer revolution. Large-scale applications that we foresee for symbolic computation include the use of automated deduction for design verification in information systems and digital electronics, expert systems for business and industry and algebraic manipulation systems for science and engineering. Demands for cheap symbolic processing will drive the development of specialized architectures and of parallel computing solutions.

This paper describes a particularly promising new architecture for symbolic computation. It is amenable to a variety of implementations. The one we describe here is a sequential processor with a RISC architecture [PaS82] and a tagged, dynamically allocated, list-structure memory [Kie85]. Simulation of the implementation provides a preliminary evaluation of its performance [Kie87]. Parallel implementations are also being studied.

1.1. Why LISP isn't enough

LISP is a mature programming language and will continue to be important because of a large user base. But LISP's do not incorporate several new advances in programming methodology that save programming effort, including:

A polymorphic type system with automatic type inference
Declarative programming with demand-driven scheduling
Pattern-matching syntax for ease of data-structure access
Algebraic data types
A logic programming option

Because LISP lacks a type system, LISP-oriented architectures typically use type tagging and support low-level type checking at runtime -- an expensive overhead. Newer and more powerful languages can be executed more efficiently than compiled LISP. The newer languages, when coupled with modern, interactive programming environments provide a new programming methodology.

1.2. Graph reduction -- a paradigm for symbolic computation

Symbolic processing involves the manipulation of expressions, which includes specifically the operations of binding variables, applying substitutions to expressions, matching patterns, regrouping and rewriting terms. The task of a symbolic processor is to evaluate expressions to new expressions, according to a prescribed set of evaluation

The research reported here has been partially supported by the National Science Foundation under grant No. DCR-8405247.

rules. When expressions are represented by strings of ASCII characters, a symbol processor is a symbolic *interpreter* for expressions. This has been a traditional approach to symbolic computation. On the other hand, expressions can be represented directly by data structures in a memory, with several significant performance advantages.

(i) Subexpressions are immediately accessible via pointers.

(ii) Multiple copies of an expression can be shared by copying a pointer, rather than by copying its entire representation.

(iii) Potentially infinite expressions can be represented by cyclic data structures.

(iv) With tagged memory, distinguished data structures representing particular expression classes can be cheaply implemented.

In general, the representation of a expression by a data structure is a rooted graph, whose nodes are architecture-defined data structures (such as a *cons* cell) and whose arcs are pointers embedded in these data structures. When the rules for evaluating expressions can be expressed as directed rewrites, we call them *reduction* rules. The evaluation of an expression represented as a graph by the repeated application of reduction rules is the process of *graph reduction.* It is a basic computational paradigm.

A graph-reduction architecture must distinguish an unreduced graph from one that is reduced. This distinction must potentially be recognizable at every step of a computation. It can be performed efficiently if the property (of being reduced) is designated in memory by a tag on the root node of each graph. Analogously, an architecture to support logic programming must distinguish an unbound variable from any other kind of expression. This justifies another memory tag.

1.3. Programmed graph reduction

Graph reduction was first introduced in connection with combinator reduction for evaluating applicative language programs [Tur79]. In the past eight years, combinator reduction has been extensively studied and several implementations have been built in software and hardware [Sch86]. Direct reduction of combinator graphs is an attractive computational paradigm because no separate program is required to provide control. Control is inferred directly from the graph undergoing reduction, by interpretation of the combinator in the applicative position in the graph identified for reduction (the *redex*). This is an elegant scheme for computation.

Unfortunately, this scheme does not achieve very high performance with any hardware implementations known to us. **S, K, I** combinator reductions are very small computational steps. (The **I** combinator, which optimizing compilation schemes seek to eliminate, is effectively a NO-OP instruction.) Many times, a reduction step serves merely to regroup local data. Using a richer set of combinators relieves this problem somewhat, but does not solve it completely.

Furthermore, the interpretation of a combinator is an inherently sequential activity. This is not to say that parallel computation is not possible by identifying multiple redices for concurrent reduction. Rather, at each redex, application of a combinator requires fetching the datum that represents the combinator, performing a case analysis of this datum (this corresponds to instruction decoding in a programmed computer), and executing the consequent graph transformation. In a von Neumann computer, the analogous fetch, execute and write sequence could be overlapped in time with the preceding and following such sequences in a data-path pipeline. But in a

combinator reduction machine, the control is not independently available to direct the pipeline as it is in a von Neumann computer. One of the principal schemes used to make computers run faster, pipelining the data path, is made unavailable by pure combinator reduction.

Graph reduction does, however, make possible one of the most important paradigms of the new programming methodology: demand-driven scheduling. It is also strongly identified with a model for integrated logic and functional programming [Lin85]. Demand-driven scheduling allows the programmer to specify the simultaneous solution of a system of constraints without specifying in detail the order in which the interdependent components of a solution must be produced. It supports transparent backtracking control. It is a relatively expensive computational paradigm, but one that should be available for the power it provides.

Compiled graph reduction provides a satisfactory resolution of the conflicting demands for flexibility (demand-driven scheduling, logic variables) and performance. Compiled graph reduction differs from combinator reduction in that control is directed by an instruction stream separate from the data graph. Pipelining is possible, and the pipeline is interrupted only when data dependent control decisions cannot be avoided. In the G-machine implementation, interruptions of the pipeline have been made extremely brief.

Compilation can detect the majority of subexpressions for which demand-driven scheduling (or provision for the occurrence of logic variables) is not called for. The compiler produces code directing call-by-value computation for such expressions. When the compiler detects an occurrence of a variable whose value may not have been provided (a call-by-need variable) and a value is required, it inserts an EVAL instruction into the compiled instruction stream. EVAL is essentially a combinator. The G-machine executes call-by-value code like a RISC, and executes the EVAL combinator by expanding it into a static code sequence, just as would be done in a microcoded implementation. The result is a high performance architecture for the new programming methodology.

2. Architecture of the G-machine

The G-machine is an abstract, stack machine architecture for computing by programmed graph reduction. The architecture was originally defined by Thomas Johnsson (Gothenburg) as the evaluation model for a compiler for LML, a purely functional dialect of the programming language ML with lazy evaluation rules. This compiler translates a source-language program into G-code, the instructions of the abstract machine. It further translates the G-code representation of a program into target code for a conventional architecture, using attribute-driven code generation.

An initial design of a hardware G-machine [Kie85] executes G-code by expanding it into microcode to be executed by an internal execution unit. The internal execution unit is RISC-like, in that most of its instructions execute synchronously, and in a single cycle of the internal clock. It had been hoped that the bandwidth required of a control memory would be significantly less than that needed at the input to the execution unit because each G-code instruction expands into a sequence of RISC-like microinstructions.

Traces of the dynamic instruction sequences executed (in simulation) by this implementation showed that the expansion of G-code into microcode did not significantly reduce the bandwidth required of the control memory. Although the ratio of executed

microinstructions to G-code was as high as 5.5:1 for many example programs, there occur sequences of G-code that translate almost 1:1 into microcode. During these bursts, the control memory must furnish G-code at virtually the same rate as the execution unit consumes microcode. The permissible bandwidth reduction is mostly attributable to more compact coding of the G-code, relative to the microcode. This observation has led us to a second-round design of the G-machine as a true RISC, without microcode.

2.1. Performance enhancement opportunities

A hardware implementation of the G-machine has significant performance advantages over a software implementation for a conventional microprocessor because:

(a) It provides hardware support for stack operations in processor registers, thereby avoiding a level of indirect access through memory to access stack-addressed data.

(b) It is inherently a tagged architecture. Tags are aggregated with data so that they are available in a single memory fetch.

(c) A dynamic, list-structure memory architecture is directly implemented, and performs garbage collection concurrently with processing activity.

(d) The RISC internal architecture executes most instructions in a single clock cycle, with relatively high instruction dispatch efficiency.

(e) A novel design for an instruction fetch pipeline produces very high instruction throughput even in the presence of control jumps and literal data in the instruction stream.

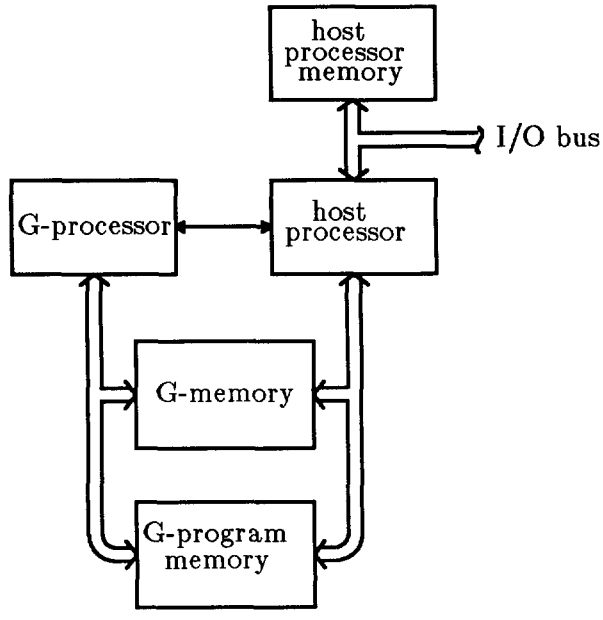


Figure 1 -- The G-machine embedded in a host computer system

These are optimizations of an architectural implementation that has been designed for fast evaluation of languages intended for symbolic processing. Not all of these optimizations are feasible for a general-purpose architecture that must execute an operating system. The G-machine processor is a slave that operates under the control of general-

purpose host, and requests any O/S services it needs from that host. In particular, the current design of the G-processor has no special hardware to support process management, interrupts or virtual memory. It could be operated in a multi-processing mode, could interact with external device processes, and could execute from virtual memory by extending the simple interface provided between the G-processor and its host. In the present design, however, the interface allows the G-processor only to signal (interrupt) the host after having deposited in an agreed-upon location in G-memory the data describing a service request. The G-processor then enters a wait state from which the host reawakens it after having provided the requested service, using shared access to the G-memory.

2.2. The abstract architecture

Components of the abstract architecture of the G-machine are:

(P) a traversal stack that holds pointers into the expression graph;

(V) a stack of registers for algebraic expression evaluation;

(ALU) a mechanism implementing a set of primitive algebraic operators;

(C) a sequential controller;

(G) a dynamically allocated store for nodes of the expression graph;

(E) an environment memory containing control sequences compiled from function definitions;

(D) a dump stack which contains the context of nested function calls.

The abstract architecture defines a set of operators (the G-code instructions) in terms of register transfers among these components.

2.3. Architectural implementation -- the G-processor

The hardware implementation of the G-machine reflects the abstract architecture, but is not merely a hardware interpreter. The P-stack is implemented as a virtually unbounded stack by providing an automatic overflow mechanism from an initial segment implemented as a register file within the G-processor. Dump contexts are also saved in this stack, making it unnecessary to explicitly save or restore processor context at a function call or return. The V-stack is a bounded-length stack coupled closely with the ALU. The V-stack is implemented separately from the P-stack in order to allow overlapped execution of ALU operations with non-ALU operations that may use only the P-stack.

There is no explicit hardware representation of the environment, E, of the abstract architecture. It corresponds to the use of function descriptor words in the G-memory, and the (separate) G-code program store. The control mechanism, C, of the abstract machine is represented in the hardware processor by the instruction fetch unit (IFU) and the processor controller.

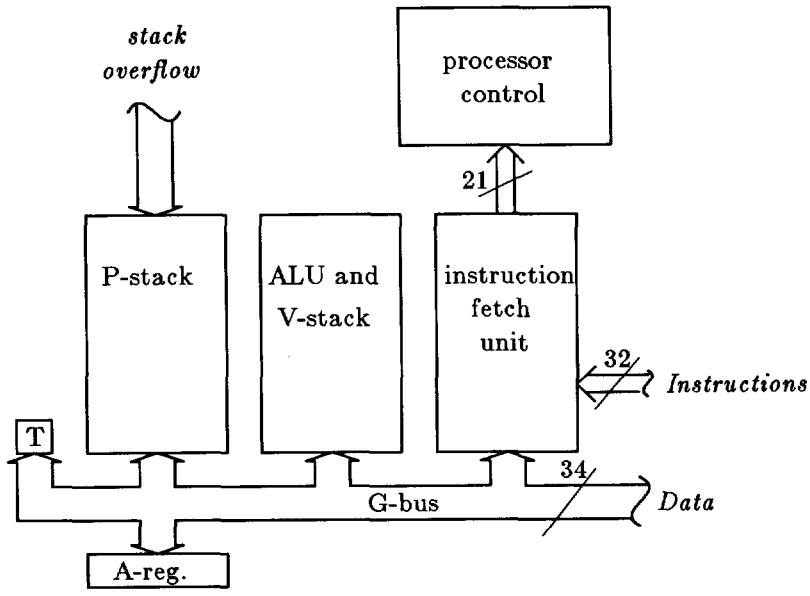


Figure 2 -- Internal organization of the G-processor

2.4. G-machine data types

The data types discriminated by the G-machine are various forms of nodes of an expression graph. The basic node form is a pair of cells, but a node may designate (a) an application node not yet reduced, (b) a basic value (a fixed-format number, a character or a boolean), (c) a structured value. A fourth form, (d) a vector node, provides contiguous storage cells for structured values. A fifth type, whose format is similar to (a) except for the use of an additional tag, represents an unbound variable and will be used in executing procedures compiled from first-order Horn-clause logic.

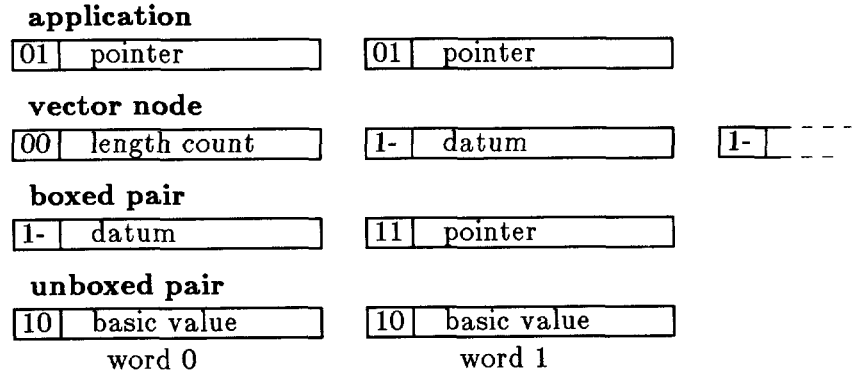


Figure 3 -- formats of the G-machine data types

The term "boxed value" has been adopted from Luca Cardelli [Car84] to refer to a structured value, as distinguished from a basic value (number, character or boolean) which is "unboxed".

2.5. Vector nodes

Vector nodes were added to the abstract G-machine by Lennart Augustsson [Aug86]. Vectors can provide constant-time access to storage for tuples of values, function applications with multiple arguments, or functional arrays [Hug85]. The representation is not entirely straightforward.

The basic operation in graph reduction is to replace an unevaluated expression graph by the graph of its value. These graphs are represented as linked data structures in the list-structured, heap memory. Graph replacement is accomplished in the G-machine by overwriting the storage cell at the root of the expression graph with the root of the result graph. The G-code UPDATE performs this operation.

It is straightforward to implement an UPDATE if each graph node is represented by a fixed-size storage cell. However, if vector nodes of various sizes are introduced, then the value that is to replace the root of an expression graph might be represented as a vector that requires more storage than has been allocated for the root node.

To avoid this problem, a vector value is not represented by a single vector node, but by a pair of nodes, the first of which is a boxed pair and the second of which is a vector node. Figure 4 illustrates the update of the root of an application graph by a vector value.

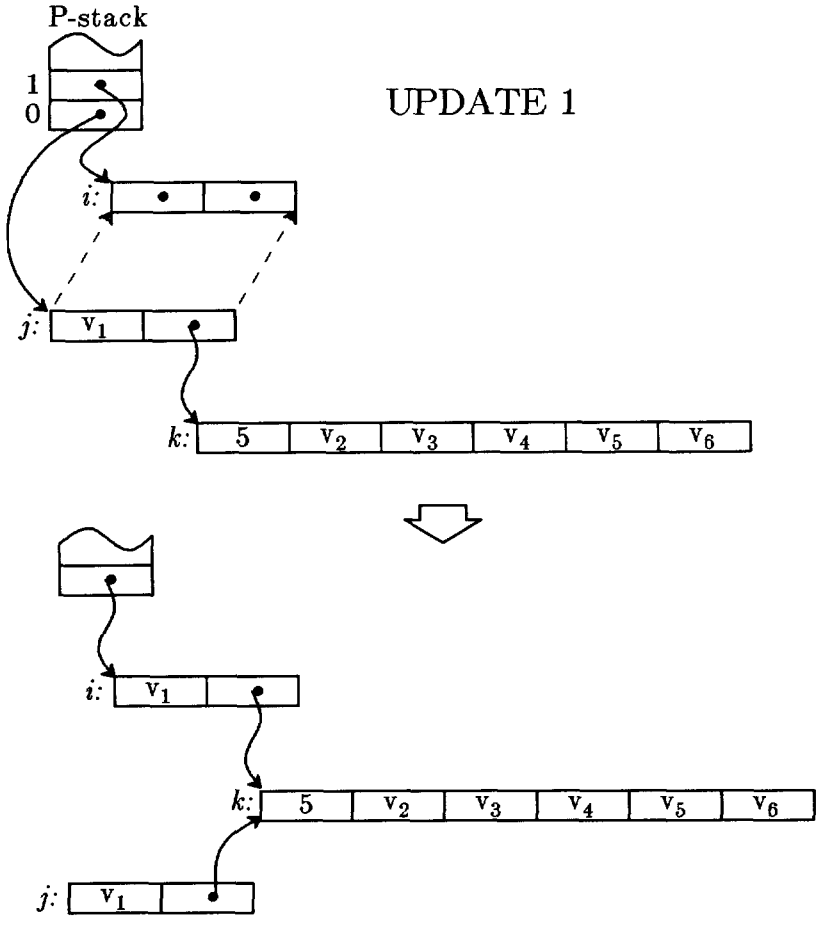


Figure 4 -- Storage representation for vector values
The diagram illustrates update by a vector value

The first element of a vector is present in the first graph node, which is always a standard pair of cells, tagged as a boxed value. The second cell of the pair points to a vector node, whose first cell contains a count of how many vector elements follow.

2.6. Evaluating applicative expressions

To better understand the computational requirements of graph reduction, it is helpful to consider ways to evaluate applicative expressions. There are several ways that the reduction of a function application can be initiated. If the compiler determines that the value of an expression will be needed when the expression is first formed, then it can be evaluated directly by a function call. The argument expressions (or their values, in a call-by-value computation) are stacked, building a local environment for evaluation of the function body. Then the function is invoked by a CALL instruction.

An applicative expression that designates the value to be returned in a function body is called a *tail-call*. Tail-calls can be further optimized, because the call to the function can be implemented simply by a control jump to the entry point of the function code, without saving a return address. When a recursively-defined function

contains recursive tail-calls, these are automatically transformed into iterative control loops by the tail-call optimization.

When lazy evaluation is the rule, an applicative expression is not to be evaluated when first encountered. Instead, an expression graph (called a *suspension*) is constructed. This graph has the form of a vine, with application nodes along its spine, and a function descriptor at its left corner, as shown in Figure 5(a). When the value of an expression that may be a suspension is needed, an EVAL combinator is applied to the expression. EVAL tests the evaluation tag on a cell of graph memory. If the tag is set, indicating that the cell has already been evaluated, then the EVAL completes without further action.

On the other hand, if the cell is the root of an application graph, then EVAL traverses the graph, pushing pointers to the arguments into the P-stack, and inspects the node that it finds at the end of the normal-order spine of the graph (Figure 5(b)). A function descriptor consists of the number of arguments expected by the function, paired with the initial address of the code sequence compiled for the function body.

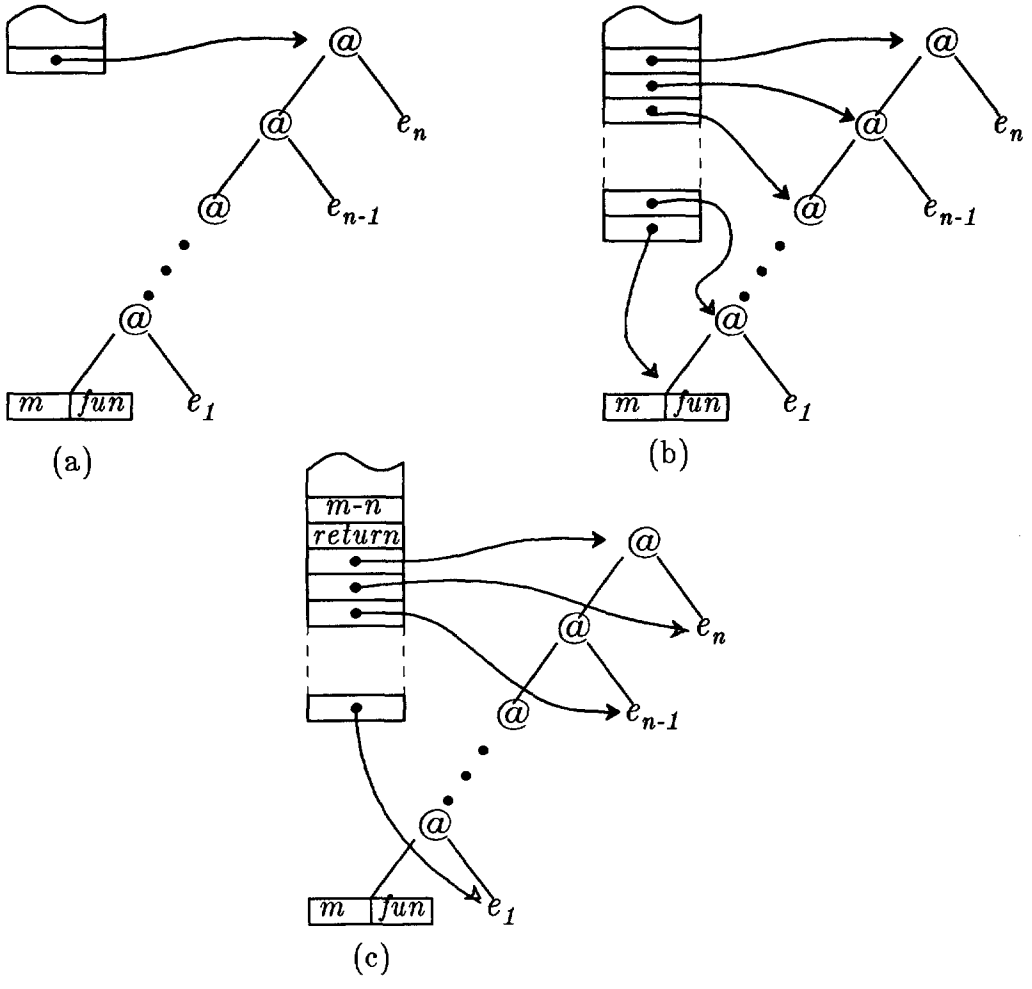


Figure 5 -- snapshots of the EVAL sequence

EVAL determines whether the P-stack contains sufficiently many arguments to proceed with reduction of the application, and if so, reconfigures the P-stack to contain pointers to the root of the redex and to each of the arguments of the application (Figure 5(c)). Underneath the redex root pointer in the stack are stored (i) the return address, and (ii) the excess argument count, which is the difference between the number of arguments stacked for this application and the number expected by the function. EVAL then invokes the function body referred to by the descriptor. In case insufficient arguments have been stacked, the expression represents a partial application and is not evaluated further. EVAL is the most complex instruction of the abstract G-machine, and dynamically expands into 65 hardware instructions to evaluate a function application to a single argument, plus 18 instructions per additional argument.

The RETURN operation is complicated by the need to determine whether the returned value represents a function whose arguments are present in the P-stack. This requires checking the excess argument count that was saved in the P-stack at the point of call. If this count is zero, which is the most common case, the return address is simply retrieved from the P-stack and loaded into the program counter. When the excess argument count is non-zero, the returned value represents a function that must be applied to the arguments stacked in the environment of the previous call, so RETURN reenters the EVAL sequence.

3. G-memory architecture

The G-machine imposes particularly stringent requirements upon the graph memory. Our simulations have shown allocation rates to be as high as 267,000 nodes/sec. Rates in the neighborhood of 100,000 nodes/sec. are typical for symbolic processing applications. Any of the stop-and-collect memory reclamation schemes would add significant overhead to a computation by graph reduction. To minimize overhead, and to avoid interruptions of service during garbage collection intervals, the G-memory has been designed to use concurrent memory reclamation based upon reference counting.

Concurrent mark-sweep collection is possible [DLM78] but analyses and simulation have shown that it does not perform well under conditions of heavy memory utilization [HiC84].

Copying collectors can incur even greater overhead under heavy memory load. Furthermore, concurrent copying would require that the G-processor have the capability to fault upon executing a memory reference that reads a forwarded pointer, then reissue the memory reference instruction on the location referred to by the forwarding pointer. Synchronization would also be needed to enforce mutual exclusion between the G-processor and a concurrent, copying collector when both attempt to access the same node. This seems unattractive.

Ideally, the G-memory should manage its own reclamation without help from or interference with the G-processor. Dynamically allocated, list-structure memory should be an architectural primitive of symbolic processing systems. In the design of the G-memory subsystem, we have tried to come as close as possible to this ideal. The memory architecture defines a "smart" controller that automatically maintains reference counts and logs memory allocation instructions. The reference counts and allocation log are used by a concurrent housekeeping processor that performs the actual restructuring of reclaimed memory nodes.

3.1. Memory reclamation by reference counting

Maintaining a dynamic count of references to each allocated node of a changing data structure is an old idea for determining when nodes are no longer needed. It has never been made practical for use at the level of individual graph nodes because of the performance required. If reference-count management were attempted by in-line instruction execution by the G-processor, the cost of reference counting would dominate all other aspects of computation. Not only must reference-count management not be a programmed task of the G-processor, it cannot even be a programmed task of a co-processor of comparable speed without degrading G-memory performance intolerably. Managing reference counts at the level of granularity of individual graph nodes requires specialized hardware in order to achieve acceptable performance.

With hardware support, reference counting is quite feasible, however. Reference count fields are maintained in a bank of G-memory that is physically separate from the data accessed by the G-processor. Thus, reference counting produces almost no memory access contention with the G-processor[1]. Reference count management is the responsibility of a hardware component that monitors requests to the G-memory from the G-processor. This is the only custom hardware component used in the design of the G-memory. All other components are off-the-shelf.

Only ALLOC and WRITE requests from the G-processor affect reference counts. The reference count manager interprets the tags of the datum in a WRITE. If the tags indicate that the value being written is a pointer, then the reference count indexed by that pointer is incremented.

Since a WRITE request also destroys the previous contents of the addressed cell, it is necessary to determine whether the previous content represents a pointer, and if so, to decrement the reference count indexed by that pointer. This is a complex requirement. In fact, most of the WRITE operations requested by the G-processor are for the purpose of initializing the values of G-memory cells, not for changing them. Changes only occur when the WRITE occurs as part of an UPDATE operation. The overhead of handling reference count decrements can be mitigated by identifying exactly those WRITE operations that cause decrements.

For this purpose, we have mandated some cooperation between the G-processor and the reference count manager. The instruction sequence implementing an UPDATE operation includes a special pseudo instruction that precedes the WRITE to each cell of the graph node that is to be overwritten. This pseudo instruction is called TRASH. It has no effect, insofar as the state of the G-processor is concerned, but it is interpreted by the reference count manager of the G-memory as an order for an indirect reference count decrement operation. That is, the reference count indexed by the contents of the node addressed by the TRASH instruction is to be decremented. TRASH is followed in the code sequence for UPDATE by a WRITE to the same address. To guard against the possibility that a reference count decrement occurs before an increment of the same count during an UPDATE operation, the reference count manager delays the decrement operation until after processing a following WRITE.

---

[1] The only source of contention that might be attributed to reference count maintenance occurs during the actual collection phase. When a node being collected contains pointer fields, the reference counts of the nodes pointed to must in turn be decremented. The collecting processor must read the fields of a node being collected in order to find any pointer values contained there. This results in some read activity to the G-memory.

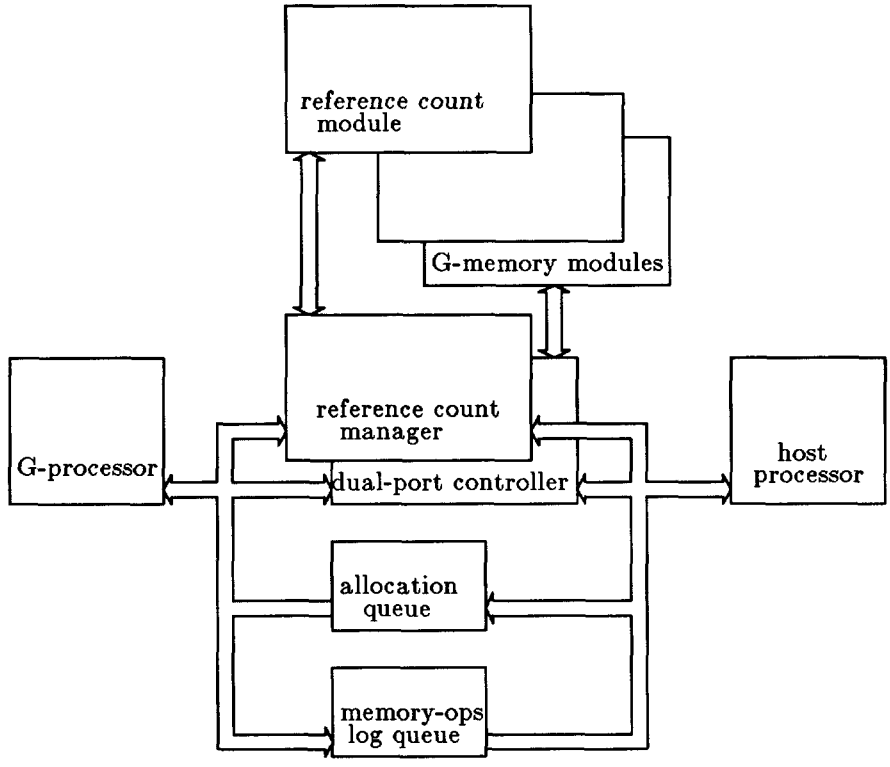


Figure 6 -- Block diagram of the G-memory architecture
The host processor performs actual collection and node allocation transactions on the memory.

3.2. Stack allocation with persistence

Hardware supported reference counts can account for references that are explicitly manifested as pointers in the expression graph. The processor state may hold additional references, however. Internal references can be created or deleted in the P-stack of the G-processor by operations that execute in a single clock cycle. Since these operations require only 1/5 to 1/3 the time needed to complete a memory operation, it is unreasonable to expect that reference count management implemented in conjunction with the G-memory could keep pace. Furthermore, to count stack references would require additional memory operations to signal each COPY, POP or MOVE operation on the P-stack. This seems impractical.

Under certain assumptions, however, it is unnecessary to account for references from the P-stack. The contents of the P-stack are logically segmented into stack frames. These correspond to the outstanding function body activations at any point in a computation, analogous to nested procedure calls in an imperative language implementation. Suppose a node n has been allocated during the execution of a function call, then the stack frame corresponding to that node must be presumed to contain an unknown number of references to n. Stack frames more deeply nested cannot contain any reference to n, since only the topmost stack frame can be actively modified. Thus, if any reference to n persists at the time that the function call returns, it is either

(A) a reference that was explicitly written into G-memory in constructing or updating an expression graph, or

(B) a reference contained in a residual part of the stack frame that was catenated onto the surrounding stack frame during the function return sequence.

Any reference satisfying condition (A) will be counted by the reference count manager of the G-memory. References occurring *only* by condition (B) must be prevented. We require of compilers for the G-machine that they must ensure that the following *reference-count safety* condition holds:

> No reference to any node allocated during execution of a function call is ever included in the residual stack frame upon return from that call.

The reference count safety condition can be satisfied by observing a computation rule:

RS: The value returned by a function call must either be a basic value returned in the P-stack, or must be represented as an update of the root of the redex graph evaluated by the function call.

Since there must exist a reference to the root of the redex graph at the point of the function call, the compilation rule *RS* ensures the reference count safety condition. Rule *RS* can be considered as a functional programming analog of the familiar rule of "call-by-reference" that allows a procedure to return its results in storage locations passed at the point of the call. Enforcing rule *RS* is neither difficult nor does it prevent the use of optimized compilation strategies. It embodies the paradigm of graph reduction.

When the reference count safety condition is observed, the references contained in the P-stack at the point of return from a function call are never more than those that existed in the P-stack at the point of call. If collection of nodes allocated during a function call were deferred until the call has returned, then intermediate (and possibly uncounted) references that may exist during execution of the function call would be of no consequence. Based upon this observation, we implement an allocation and collection strategy called *stack allocation with persistence.*

During execution of a function call, the nodes allocated by that call are not collected, but their addresses must be kept track of. Following return from a function call, the reference counts of all nodes that were allocated during execution of the call are examined. Those whose reference counts are zero are collected. The reference count of any node pointed to by a collected node is also decremented and that node is examined for collectability in the usual manner. Any node not collected during this sweep is marked as *persistent.* A persistent node whose reference count is later found to be zero may be immediately collected.

The only price to be paid for this strategy of deferring collection until after a function call has returned is that some nodes allocated as temporary storage cells will not be collected as soon as they might have been. Thus, the overall use of an address space will not be as efficient as it might have been if immediate collection were practiced.

In the G-machine implementation, actual collection is performed by the host processor, using dual-port access to the G-memory. The deferred collection scheme has been shown to perform well in simulation studies. Assuming a 16 Mhz, M68000-family processor as the host, the collector has no difficulty in keeping up with the demands of the G-processor [Ran86].

3.3. Allocation

ALLOC is a primitive instruction of the G-processor. It is expected to complete in a single cycle most of the time, although it cannot be assumed to be a synchronous instruction. To support allocation, the G-memory includes a hardware FIFO queue holding node pointers pre-allocated from the free storage pool by the host processor. Unless this queue is empty, the ALLOC instruction completes execution immediately, returning the node pointer from the front of the queue. Each issue of an ALLOC instruction to the G-memory is logged in the memory-ops log queue. The value of the pointer returned in response to an ALLOC is not logged, however.

In order to base a collection strategy on the reference count safety condition, it is necessary to be able to identify those nodes that were allocated during execution of each function call. In principle, this could be done by logging the addresses that were returned in response to ALLOC requests by the processor, but this is unattractive because of the storage required to maintain an explicit address log.

Suppose it were possible to allocate addresses sequentially from a free storage pool. Then it would only be necessary to log the first and the last node address that was allocated during each call. If the G-processor were induced to cooperate by issuing memory signal instructions START and FINISH as part of the code sequence to implement EVAL or CALL, and RETurn operations, respectively, then the G-memory controller could log the requisite node addresses automatically.

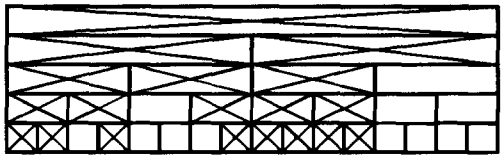
The more difficult problem is how to ensure sequential issue of addresses in allocation. If a linked list were used to represent the free storage pool, then this list would have to be maintained in address-sorted order. The algorithmic complexity of inserting the addresses of reclaimed nodes in order to maintain a sorted list precludes the use of this technique.

3.3.1. Modified buddy-system allocation

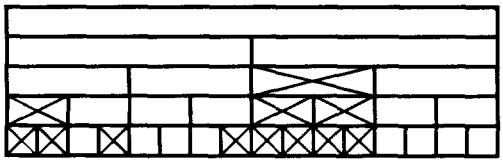
The buddy-system strategy [Knu68] was developed to allow easy allocation and aggregation of variably sized blocks of storage. In buddy system allocation, the available address space is logically partitioned into blocks, each of a size that is a power of two times the size of the smallest allocatable unit, and each block is aligned on an address that is a multiple of the block size. A block of size 2^N, $N>1$, contains nested within it a pair of blocks of size 2^{N-1}. This is a pair of "buddies". Because blocks are aligned on addresses that are multiples of the block size, it is easy to identify the buddy of any block. The address of the buddy of a block with address A and size 2^N is $\lfloor A+2^N \rfloor_{2^{N+1}}$. We use the notation $\lfloor x \rfloor_y$ to denote the greatest multiple of y that is less than or equal to x, and $[x]_y$ to denote x modulo y.

Associated with each block, although not necessarily in contiguous storage, is a tag bit that indicates whether or not that block is free. Assume the tag value is 0 if the block is free, and 1 if it is not. A block is free only if every block nested within it is free. To allocate a free block of size 2^N, the tags corresponding to size 2^N are searched in order for the first that is zero-valued. When a block is selected for allocation, its tag is set, and so also are the tags of each of its ancestor blocks, since each of them now contains a nested block that is allocated.

When a block of storage is reclaimed, its tag bit is cleared, and the tag of its ancestor is set to equal the tag of its buddy. This is the boolean disjunct of the tags of



(1) Blocks allocated



(2) Blocks fully occupied

Figure 7 -- buddy system allocation

the (now free) block and its buddy. If this marking changes the value of the ancestor's tag, then the ancestor has just been freed, and the tag of its ancestor must be reset, etc., etc. Buddy system allocation allows sequential allocation of free blocks, and cheap re-aggregation of buddy pairs.

Our need is a little different, however. Suppose for the time being that we are interested only in the sequential allocation of free blocks of unit size. Let us place a different interpretation upon the tag bits. A block of unit size will be tagged as free or allocated as before, but a block of size 2^N, $N>1$, will be tagged as allocated only if it contains no free unit block. Thus a block remains marked as free until all of its unit blocks are allocated. Now, when a block is freed, instead of marking its ancestor with the tag of its buddy, its ancestor's tag is zeroed since the ancestor now contains a free block. Zeroing of tags propagates upward in the ancestor tree until it reaches a level at which it produces no change.

It might appear as if sequential allocation in the modified buddy system should proceed from the top of the ancestor tree, always selecting the leftmost subordinate block whose tag shows it to be free. This strategy would indeed select the lowest addressed unit block that is free, but would require a number of tag inspections equal to $\log_2 N_{\max}$, which is undesirable and usually unnecessary.

A more efficient way to secure sequential allocation of unit blocks is to begin the search from the last address previously allocated. If A_{last} is an even address, then try its buddy. If this fails, or if A_{last} is odd, then move up a level in the ancestor tree. In general, if at level N and $[A_{last}/N]_{2^{N+1}} = 0$, then try the level N buddy. If this test fails, or the buddy is not free, then move up to level $N+1$ and try again. This will locate the lowest addressed block that contains a free unit block with address greater than A_{last}, if such exists.

When such a block is found then the search proceeds downward, selecting the left-most buddy that is marked as free, just as does the search from the top level. However, the upward search doesn't very often reach the top level, and so the average number of comparisons is greatly reduced. If the address space were randomly occupied with a uniform distribution of allocated blocks, and the occupancy ratio were 50 per cent, then the average number of levels searched would be slightly less than 1.5. The search is economical enough to be practical for real-time allocation by a host processor.

This describes in outline the strategy used to allocate and reclaim storage for the G-machine. When the G-processor signals the memory that a new function call is beginning by issuing a START pseudo instruction, the reference count manager enters the START into the memory-ops log queue. The host, upon interpreting the sequence found in this queue, will note the beginning of a new address allocation interval to be associated with the newly active function call. All ALLOC's that are issued are also logged. When the function returns, a FINISH pseudo instruction is logged and the host processor can terminate the allocation interval for the call. Note that the host need not analyze the pseudo-ops in real time, insofar as the G-processor is concerned. It must only keep up with the average rate of issue of the pseudo-ops.

An allocation interval that has been FINISHed can be collected. The host sweeps the addresses of the unit blocks, inspecting the allocation tag, the reference count and the persistence tag of each node. Any node with zero reference count can safely be reclaimed. Any node not reclaimed is marked as persistent at this point. When reclamation of an interval has completed, that interval can be reused as an allocation interval for a later function call. As time progresses, allocation intervals tend to become fragmented. However, these intervals are themselves allocated by the buddy system and re-aggregation is relatively cheap.

The buddy system also allows for allocation of non-unit blocks to provide storage for vector nodes. This requires keeping an extra set of allocation tags for blocks of size greater than one. These tags are managed according to the original buddy system tag discipline.

One drawback of reference counting as a basis for memory recycling is that a cyclic graph will never be collected, even if it becomes unreachable from the roots of the active expression graph. No satisfactory, practical algorithm for concurrent collection of general cyclic graphs is known. Algorithms that will allow collection of restricted cyclic graphs concurrently and in real time are possible. Another practical possibility is to invoke a stop and collect garbage collector at infrequent intervals.

3.4. How to fetch tagged data from an untagged memory

One problem that tagged architectures have is that the memory structure is incompatible with that of standard systems architectures at the word level. General-purpose architectures have become standardized on 32-bit words. Backplane busses are often built to accommodate this word size. General purpose microprocessors are standardized on 32-bit data paths. A tagged architecture clashes with standardization.

Two approaches to resolve the clash between the tagged architecture of the G-machine and the untagged 32-bit data paths of general purpose computer systems come to mind.

(1) Accommodate the tags within a 32-bit word by making the internal data path of the G-machine equal to 32-t, where t is the number of tag bits required. Memory fetches by the G-processor would conform to the standard word size, and the tags and data would be separated internally. This is the tack taken by the designers of the Xenologic X-1 processor [DDP85], an application accelerator for Prolog. This scheme has the disadvantage that other standard parts that might someday be used in the data path of the G-machine, such as a floating-point coprocessor, may require a 32-bit data path, resulting in an internal data path width incompatibility.

(2) Let the internal data path of the G-machine be 32 bits and accommodate the tags in memory with an auxiliary byte not necessarily contiguous with the data word with which the tags are associated. This solution might appear to require multiple data fetches by the G-machine in order to aggregate tags and data into a tagged word. Note, however, that aggregation could be done by a data cache. Upon a cache miss, fetches of both the data word and the correspondingly addressed tags byte are initiated by the cache controller. When both arrive, they are assembled into a single tagged data word of width 32+t bits within the cache. The data path width between the cache and the G-processor is the full 32+t bits.

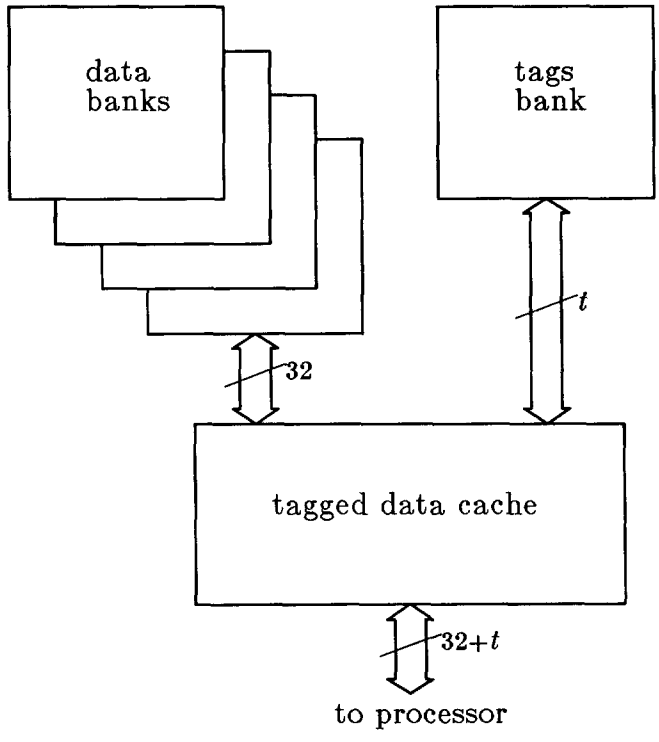


Figure 8 -- Data aggregating cache for tagged memories

Because it allows standard data path widths to be maintained both within and without the G-processor, we favor the latter solution.

4. A RISC G-machine

As an alternative to the instruction fetch and translation mechanism originally designed for the G-processor, the design is simplified by making it a true RISC, dispens-

ing with the instruction translation phase entirely[2]. One difficulty with abandoning the design of a microcoded machine is that there can be an impressive give-away in code volume if every complex instruction of the abstract G-machine is replaced in-line by its static expansion sequence. Take the EVAL combinator, for instance. The static sequence is implementing EVAL is 58 instructions long. When compilation follows a lazy evaluation rule, EVAL occurs frequently, and in-line expansion of the static sequence of this instruction alone could multiply the code volume by several times. An alternative is to realize the few complex instructions whose static expansions are large by subroutines. The subroutine call is a relatively short instruction sequence. It does represent introduced overhead, however.

There is a better alternative than to use a conventional implementation of subroutines. Expansion of the complex operations of an abstract machine into RISC code sequences is a rather specialized activity. Complex operations are never nested within RISC sequences, so the subroutine mechanism does not have to be recursive. There are no parameters. (Actually, some G-code operations do take a single parameter, but these operations translate into very short instruction sequences, typically one or two RISC instructions). The IFU can easily be designed to perform such simple subroutine calls itself, with no introduced overhead.

4.1.1. Fetch-time macro expansion

The 32-bit RISC instruction format of the G-machine has a seven-bit opcode field accompanied by a single tag bit, called the *macro tag*. When an instruction has the macro tag bit set, it is either the first instruction of a subroutine sequence, or is an instruction in the sequence from which a return must be made. The low-order 10 bits of a macro-tagged instruction are preempted for a subroutine continuation address. (Except for jump and conditional jump instructions, and literal data, the low-order 10 bits of the instruction format are unused.) In compiled code, only the first instruction of the subroutine sequence representing a complex G-code operation is emitted in line. The remainder of the sequence is loaded with a subroutine library into the lowest addressed page of control memory.

The IFU of the RISC G-machine will respond to the macro tag bit by making a transition between two states, as indicated in Figure 9. When the IFU in its normal state encounters an instruction with the macro tag set, it saves the active program counter value in a register and loads the 10-bit address field from the instruction, padding the high order bits with zeros. Fetching continues from the addressed subroutine sequence until another instruction is encountered whose macro bit is set. That causes the IFU to return from the subroutine call, reloading the return address from the save register. Neither saving nor restoring the program counter inhibits issue of the selected address for the next instruction fetch, and so no delay is incurred. This mechanism has the effect of removing the microprogram store to the regular control memory, and makes the IFU perform the function of a simple microsequence controller. We can enjoy the code space economy of complex instructions, but without surrendering the performance advantages of a RISC!

Furthermore, the strategy works well with an instruction cache. Since the subroutines that implement G-code operations are small, they exhibit very good locality and

[2] This idea has been promoted by Bill Hostmann, who not only suggested it, but argues it strongly.

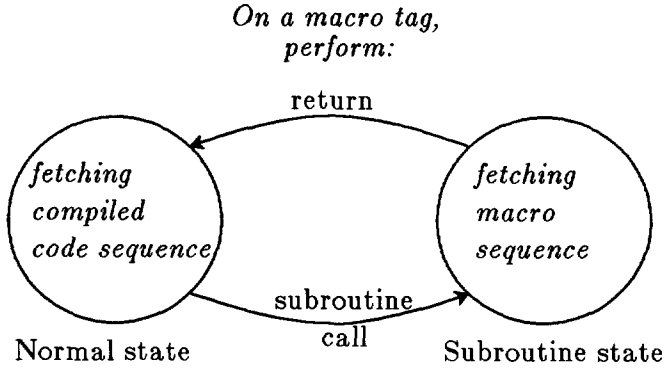


Figure 9 -- States of the instruction fetch unit.
Transitions occur whenever a fetched
instruction has the macro tag bit set.

cache misses should not occur once the cache is loaded with the subroutine code. Non-locality is induced by switching back and forth between the normal control sequence and the subroutine sequences. A two-way, set associative cache would accommodate this addressing pattern very nicely. With subroutines effectively locked into an instruction cache, its performance should approximate that of a microstore.

5. Performance Estimation

Performance of the G-machine implementation has been evaluated by executing code compiled from a number of functional language (LML) source programs.

5.1. Macrosimulation

The macrosimulator (Macrosim) is an instrumented interpreter for G-code, the instructions of the abstract architecture. Macrosim is coded in Modula-II, and runs approximately as fast as a Lisp interpreter, exclusive of the extra code inserted for instrumentation purposes. It includes a garbage collector, and is therefore quite satisfactory for simulating the execution of good-sized test programs.

Even though it does not simulate execution of each RISC instruction individually, Macrosim can provide data on RISC instruction execution. It uses a table that summarizes the number of instructions that would be executed by the G-processor in the expansion of each G-code operation. When the RISC sequence is data dependent, as is the case for the G-codes EVAL and RETURN, Macrosim's table gives the number of microinstructions that would be executed and the average execution time corresponding to each data-dependent option. Thus Macrosim furnishes not only accurate counts of G-code operations by species, but also the time attributed to them and the total number of RISC instructions that would be executed by the actual G-processor.

Since the garbage collection implemented in Macrosim does not simulate the reference counted garbage collection of the G-machine design, no time has been attributed to garbage collection in the statistics reported here.

5.2. G-code translation

In order to obtain a comparison with the performance of an implementation on a conventional computer, the G-code output by our compiler has been expanded into native machine code for a DEC VAX-11, an architecture with which there is wide familiarity. Furthermore, the code generator has been instrumented so that in the VAX code sequence emitted for each G-code instruction, there is included code to increment two counters. One counts the number of executions of that G-code instruction species, and another counts the number of VAX instructions executed in its expansion (exclusive of instrumentation instructions). We thus obtain the number of VAX instructions executed when the abstract G-machine architecture is implemented by compiling G-code into VAX code.

Of course, there are some basic differences between the VAX implementation and the G-machine design. Since the VAX is not a tagged-memory architecture, The tags required by the G-machine are implemented with an extra 16-bit word extending the storage allocated for each graph node. Also, the EVAL instruction is implemented as a parameterless, assembler-coded procedure. However, the G-code sequences interpreted by Macrosim and those translated into VAX code are identical.

VAX code generation is attribute-driven and reasonably sophisticated. It takes advantage of opportunities to use auto-increment addressing modes. Operands are held in registers when possible. In the examples we have run, the contents of the G-machine V-stack are always held in registers, never in memory.

5.2.1. A Comparative Architecture Measure

Since we can measure the simulated time require for the G-machine implementation to execute test programs and measure the number of VAX instructions executed to run the same test programs, we have a way to directly compare the designed G-machine implementation to that of a VAX. The measure is

$$\textbf{equivalent VAX Mips} = \frac{\text{number of VAX instructions executed}}{\text{time for G—processor execution (microseconds)}}$$

This measure is more meaningful than raw benchmark timings because it is not contaminated by variables that are uncontrollable or hard to predict, such as time spent in performing operating system services, cache performance, or different garbage collection algorithms. It could be made more universal if the technology-dependent performance assumptions made in the design of the G-processor could be factored out. Primarily this means the designed clock frequency. Unfortunately, not all of the technology assumptions can be expected to scale uniformly. A faster processor does not necessarily mean that a faster G-memory will be available to accompany it.

Nevertheless, this relative architectural performance figure is more meaningful than figures such as the number of function evaluations per second, which is wildly unstable across a mix of different test programs, or the number of graph reductions per second, which depends upon compilation strategy at least as much as upon machine architecture and implementation.

6. Simulation results

The G-machine design has been evaluated by simulated execution of a variety of test programs, using several different computation rules realized by variants of a

functional language compiler.

As a technology assumption, the simulator assumes the clock cycle time of the implementation to be 100 nanoseconds. When times are quoted, they are derived from the number of cycles executed. Average execution times for the RISC sequences of each G-code species were obtained by a detailed microsimulation of the G-processor design. The microsimulation is not described here, but it accurately accounts for the internal concurrency of the data path pipeline.

The G-memory is also assumed to be synchronous (not essential to the design but convenient for simulation) and to require three cycles to complete a read operation. The processor does not wait for write operations to complete, unless followed closely by another memory operation. Caching of the G-memory was not assumed for these simulations.

Caching of the control memory was assumed, but no allowance is made for cache misses. Nor was allowance made for virtual memory paging time in either the G-memory or the control memory.

6.1. Computation rules

Several different computation rules have been followed in compiling code from functional language source programs. The source language is LML [Joh84], a purely functional language with lazy evaluation rules. However, it is obviously not necessary to adhere to a lazy evaluation rule in compiling programs, and we have created several versions of our compiler in order to compare the performance of different computation rules.

Lazy evaluation, is the rule of the original source language. It means that functions evaluate their arguments only by need, and that the constructors of all compound data types are lazy as well. N-tuples are compiled as pairs, nested to the right, and each pair construction is lazy. Values of defined constants (the definitions appearing in let expressions) are also evaluated only by need. Nested abstractions are transformed into global function declarations by lambda-lifting [Joh85] rather than by m.f.e. abstraction [Hug82] and so do not achieve "full-laziness". No attempt is made by the compiler to identify textually separate occurrences of equivalent subexpressions.

Strict evaluation means that both functions and data type constructors evaluate their arguments. Strict evaluation still fails to achieve the full economy of call-by-value, however. Values are uniformly represented by nodes in the data graph, and when arguments are bound in a function call, they are communicated by passing pointers to the graph nodes that represent their values, even if these values are "unboxed".

In **call-by-value** computation, basic values are passed directly rather than indirectly. In the G-machine, this means that basic values are passed in the P-stack. In the presently operational version of our compiler, only arithmetic and conditional expressions occurring as operands are actually passed by value. Applicative expressions are passed by reference, because the result of a function application produces a pointer to a graph node that represents the value. The G-machine distinguishes between basic values and pointers in the P-stack by testing the "is-pointer" tag that accompanies each element of the stack.

6.2. Statistics

Statistics are given for each program executed by Macrosim, subject to the assumed invariants stated above. The statistics presented are:

- total execution time, in seconds (cycles $\times 10^{-7}$)
- RISC instruction dispatch rate, (instructions/cycle)
- microcode expansion ratio (RISC instructions/G-code executed)
- rate of G-memory allocations, ($\times 10^{-5}$)
- relative performance, in equivalent VAX Mips
- total number of EVAL's executed
- G-code execution profile, by instruction group

The RISC instruction dispatch rate is a measure of the internal efficiency of the processor. At a rate of 1.0 the processor would execute one instruction per clock cycle. It is less than one because some instructions, such as READ from G-memory or an ALU operation, do not execute in a single cycle. When the instruction following one of these multi-cycle operations requires a resource in use by the multi-cycle instruction, dispatch of the following instruction is inhibited until the needed resource is free.

The G-memory allocation rate has been presented to quantify the requirement imposed upon the G-memory controller. The total number of EVAL instructions executed has been included in the statistics to help compare alternate computation rules.

The G-code execution profile indicates how the processor is spending its time, and in particular, helps to determine where effort should be directed to obtain further performance improvements. The G-code groupings used in the profile are:

Group	Activity	G-codes
CALL	function call and return	EVAL, CALL, RET, RET_INT
ALLOC	graph node creation	ALLOC, MK_PR, MK_APP, MK_INT
UPDATE	node update	UPDATE, UPDATE_PR, UPDATE_V
ALU	arithmetic operations	ADD, SUB, MUL, shifts, etc.
READ	read G-memory	FST, SND, GET_FST, GET_SND
STACK	pure stack operations	COPY, MOVE, POP, ROT, NROT
JMP	control jumps	JMP, J_ZERO, J_NOT_ZERO, J_IF_PTR, etc.
LIT	literal data	PUSH_CONST, PUSHGLOBAL, GET_BYTE

6.3. Test results

The first example is a strange function (the Takeuchi function) concocted to produce an extremely high incidence of recursive function calls. Source code for each of the example programs is given in the Appendix.

The profile shows that the time attributed to G-code operations in the CALL group is affected drastically by the computation model. Under lazy evaluation, the EVAL operation is used to invoke all function calls. This dominates all other computational operations, in terms of total time consumed. It is because EVAL represents data-dependent control; it must interpret a graph by reading and testing every node on its spine. Under a strict evaluation rule, EVAL is replaced by an explicit CALL which takes little time. RETurns are actually more costly than CALLs under this rule.

Tak			
	Lazy	*Strict*	*by-Value*
time	1.691	0.542	0.491
dispatch rate	0.76	0.67	0.65
expansion ratio	7.23	2.98	2.62
allocation rate	1.13	1.76	0.97
equivalent VAX Mips	3.23	3.43	
number of EVAL's	190828	1	1
G-code profile:			
CALL	.643	.264	.291
ALLOC	.155	.130	.030
UPDATE	.053	.166	.184
ALU	.039	.123	.136
READ	.053	.165	.182
STACK	.043	.094	.113
JMP	.008	.026	.029
LIT	.010	.019	.021

Note also that the dynamic instruction expansion ratio (RISC/G-code) is also dominated by the expansion of EVAL operations in lazy evaluation. The expansion ratio is much more modest when EVAL is removed.

The RISC dispatch rate is adversely affected by removing the EVAL operations. The EVAL sequence of RISC instructions has been tightly hand coded to allow time-overlapped execution of asynchronous instructions (reading G-memory and ALU operations) with synchronous instructions that do not contend for the same resources. It makes better use of opportunities to use the RISC delayed jump than do the sequences generated by our compiler. It also involves more data traffic internal to the G-processor than do typical compiled code sequences. For all of these reasons, the EVAL sequence achieves better instruction dispatch efficiency than is typical for compiler-generated code sequences.

Call by value made only a little difference in the preceding example, in which most argument expressions were function applications rather than explicit arithmetic expressions. Functions return their values in the expression graph. In the next example, call by value effects a substantial improvement. The program is a linear-time Fibonacci sequence algorithm that uses a pair of accumulator arguments to hold the two previous values in the sequence, and a third argument to count down the number of sequence elements left to be generated. The function is tail recursive.

Linfib shows the most dramatic speedup from lazy to strict evaluation, nearly four times. This is entirely attributable to having suspended the arguments in repeated calls to the function under the lazy evaluation rule, even though these are tail calls. Notice that when the the computation rule is strict evaluation, the G-memory allocation rate has become 267,000 nodes/second. These allocations are all for graph nodes in which to pass an integer-valued argument (by reference) in a tail-recursive function call. The call-by-value model eliminates all graph node allocations in this example.

Linfib			
	Lazy	*Strict*	*by-Value*
time	.00292	.00074	.00069
dispatch rate	0.79	0.66	0.58
expansion ratio	5.93	2.23	1.43
allocation rate	1.01	2.67	0.0
equivalent VAX Mips	3.61	4.43	
number of EVAL's	300	1	1
G-code profile:			
CALL	.649	.018	.019
ALLOC	.139	.313	.000
UPDATE	.001	.003	.003
ALU	.051	.202	.303
READ	.069	.274	.367
STACK	.061	.108	.159
JMP	.010	.040	.072
LIT	.016	.030	.048

The last example is one that requires lazy evaluation, or at least lazy evaluation of the list constructor. It uses Peter Quarendon's algorithm [Hen80] to computing prime numbers by the sieve of Erastothenes. This algorithm is formulated in terms of streams, or infinite lists, which are evaluated incrementally by lazy evaluation. The computation is cut off by a result counting function at 250 primes.

Primes	
	Lazy
time	0.798
dispatch rate	0.72
expansion ratio	4.84
allocation rate	0.88
equivalent VAX Mips	3.86
number of EVAL's	104984
G-code profile:	
CALL	.537
ALLOC	.120
UPDATE	.052
ALU	.086
READ	.100
STACK	.077
JMP	.013
LIT	.011

7. Conclusions

Simulation of the hardware G-machine demonstrates the performance improvement achievable through architecture. The relative performance, of the G-processor,

expressed in terms of a well-known architecture for general-purpose computation, is roughly that of a 3.6 Mips VAX. The design of the G-processor does not rely on any particular device technology, although there are a few components that may be difficult to realize without a custom VLSI implementation. The technology assumptions made in the design have been conservative. Commercial, silicon VLSI processors of much greater design complexity have achieved basic clock rates well in excess of twice that for which we have designed. Further improvement in performance might also be obtained by caching G-memory accesses.

The G-processor achieves its performance through a few basic mechanisms and design strategies:

a) A RISC architecture, most of whose instructions execute synchronously and can be dispatched on nearly every clock cycle. The measured instruction dispatch efficiency (0.58 to 0.79) indicates the effectiveness of the pipelined data path.

b) The context for a function call is maintained by the processor in the top segment of the P-stack, which is held in a register file within the processor. The overhead of a context switch is minimal, and consists of saving (or restoring) a return address and an excess argument count in the P-stack.

c) Data are tagged to distinguish pointers from basic values and applications from reduced data structures. Tags are fetched from memory along with data and can be tested in the following cycle. Pointer tags are saved in the P-stack. This allows basic values as well as pointers to reside in the P-stack and supports true call-by-value optimizations, superimposed upon a graph-reduction model of computation. The addition of another tag will also support efficient computation of procedures compiled from logic programs.

d) Concurrent management of the G-memory supports a high rate of graph node allocations without interrupting the activity of the G-processor.

The design is notable for its versatility. It supports several models of computation, ranging from lazy evaluation of recursive functions through iteration with call-by-value. Several mechanisms are available to invoke function calls, providing efficient evaluation of function applications whether suspended or not, and whether or not the compiler can identify the function being applied.

8. Acknowledgements

This research would not have been possible without the dedicated efforts of Linda Rankin, Shyue-Ling Kuo, Siroos Farahmand-nia, Boris Agapiev and Raman Tenneti.

References

[Aug86] Augustsson, L., "The revised G-machine," PMG Memo 45, Dept. of Computer Science, Chalmers University of Technology, Gothenburg, 1986.

[Car84] Cardelli, L., "Compiling a functional language," *Proc. of 1984 ACM Conf. on Lisp and Functional Programming*, 1984, pp. 208-217.

[DLM78] Dijkstra, E. W., Lamport, L., Martin, A. J., Scholten, C. S. and Steffens, E. F. M., "On-the-fly garbage collection: an exercise in cooperation," *Comm. ACM*, vol. 21, 11 (1978), pp. 966-975.

[DDP85] Dobry, T. P., Despain, A. M. and Patt, Y. N., "Performance studies of a Prolog machine architecture," *Proc. of 12th Internat. Conf. on Computer Architecture*,

1985.

[Hen80] Henderson, P., *Functional Programming*, Prentice-Hall International, London, 1980.

[HiC84] Hickey, T. and Cohen, J., "Performance analysis of on-the-fly garbage collection," *Comm. ACM*, vol. 27, 11 (1984), pp. 1143-1154.

[Hug82] Hughes, R. J. M., "Super-combinators," *Proc. of 1982 ACM Conf. on Lisp and Functional Programming*, Pittsburgh, 1982, pp. 1-10.

[Hug85] Hughes, J., "Lazy memo-functions," in *Functional Programming Languages and Computer Architecture*, vol. 201, J. Jouannaud (ed.), Springer-Verlag, Nancy, 1985, pp. 129-146.

[Joh84] Johnsson, T., "Efficient compilation of lazy evaluation," *Proc. of 1984 ACM/SIGPLAN Notices Conf. on Compiler Construction*, Montreal, 1984.

[Joh85] Johnsson, T., "Lambda lifting: transforming programs to recursive equations," in *Functional Programming Languages and Computer Architecture*, vol. 201, J. Jouannaud (ed.), Springer-Verlag, Nancy, 1985, pp. 190-203.

[Kie85] Kieburtz, R. B., "The G-machine: a fast, graph-reduction evaluator," *Proc. of IFIP Conf. on Functional Prog. Lang. and Computer Arch.*, Nancy, 1985.

[Kie87] Kieburtz, R. B., "Performance evaluation of a G-machine implementation," in *Graph Reduction*, R. B. Keller and J. H. Fasel (ed.), Springer-Verlag, LNCS series, 1987.

[Knu68] Knuth, D. E., in *The Art of Computer Programming, Vol 1 -- Fundamental Algorithms*, Addison-Wesley, Reading, Mass., 1968.

[Lin85] Lindstrom, G., "Functional programming and the logical variable," *Proc. Twelfth ACM Sympos. on Principles of Programming Languages*, New Orleans, 1985, pp. 266-280.

[PaS82] Patterson, D. A. and Sequin, C. H., "A VLSI RISC," *Computer*, vol. 15, 9 (1982), pp. 8-18.

[Ran86] Rankin, L. J., *A dual-ported real memory architecture for the G-machine*, M.S. thesis, Oregon Graduate Center, Beaverton, Oregon, 1986.

[Sch86] Scheevel, M., "NORMA: a graph reduction processor," *Proc. of 1986 ACM Conference on Lisp and Functional Programming*, Cambridge, Mass., 1986, pp. 212-219.

[Tur79] Turner, D., "A New Implementation Technique for Applicative Languages," *Software - Practice and Experience*, vol. 9(1979), pp. 31-49.

Appendix -- Test Programs

```
*****************************
Tak
*****************************
letrec
   tak x y z =
         if ~(y < x) then z
         else   tak (tak (x-1) y z)
                    (tak (y-1) z x)
                    (tak (z-1) x y)
in
   tak 18 12 6

*****************************
Linfib
*****************************
letrec
   fib x y n =
         if n=0 then y
         else fib y (x+y) (n-1)
in fib 0 1 100

*****************************
Primes
*****************************
   letrec
      from x = x . from (x+1)
   and
      counthd lis n =
         if n = 0 then []
         else hd l. counthd (tl l) (n-1)
   and
      filter p seq =
         case seq in
            (a.rest) :
                if a%p ~= 0 then a.filter p rest
                else filter p rest
         end
   in
      sieve seq =
         case seq in
            (p.rest) :
                p.sieve (filter p rest)
         end
   in counthd (sieve (from 2)) 250
```