The G-machine: A fast, graph-reduction evaluator

Richard B. Kieburtz
Oregon Graduate Center
Beaverton, Oregon USA

OGC Technical Report CS/E-85-002
January 29, 1985

Abstract

The G-machine is an abstract architecture for evaluating functional- language programs by programmed graph reduction. Unlike combinator reduction, in which control is derived dynamically from the expression graph itself, in programmed graph reduction control is specified by a sequence of instructions derived from an applicative expression by compilation.

The G-machine architecture was defined by Thomas Johnsson and Lennart Augustsson (Gothenburg) as the evaluation model for a compiler for a dialect of ML with lazy evaluation rules. This paper describes a sequential evaluator based upon that abstract architecture. It discusses performance issues affecting reduction architectures, then describes the organization of a hardware design to address these issues. The interplay between compilation strategies and the computational engine is exploited in this design.

Principal features of the design are (i) hardware support for graph traversal, (ii) a vertically microcoded, pipelined internal architecture, (iii) an instruction fetch and translation unit with very low latency, and (iv) a new memory architecture, one specifically suited to graph reduction and which can be extended to very large memories.

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

1. Reduction systems

A reduction architecture evaluates an expression by transforming it through a series of intermediate forms until it cannot be further transformed, under a set of admissible rewrite rules. The expression is then said to be in normal form, which is the value attributed to the original expression.

To be deemed useful for evaluating functional-language programs, a reduction system must be consistent with the mathematical semantics of applicative expressions. It should also have the Church-Rosser property, which assures the uniqueness of a normal form independent of the particular reduction sequence by which it is produced. These, however, are the only constraints placed upon reduction systems, and a great variety of elegant and imaginative schemes are possible.

Church's lambda-calculus and Curry's combinatory calculus are the best known examples of reduction systems, and have served as abstract models for experimental computing engines [Ber75] [Sch85, Cla80, SCN84]. There are other systems as well. A string-reduction system implementing the semantics of Backus' metalanguage FFP has been used as the basis for an architecture allowing massive parallelism [Mag79]. String reduction is essentially like combinator reduction except that the value of an expression is not shared among multiple references; the expression is multiply evaluated.

Architectures based upon β-reduction (lambda-calculus), combinator reduction, or string reduction all have in common the property that *control* -- the selection of the next reduction step -- is dynamically derived from the current expression form at each stage in a reduction sequence. We call systems with this property *pure reduction* systems.

As an alternative to pure reduction systems, we may also consider *programmed* reduction. In a programmed reduction system, the steps of computation are still transformations of an expression form by application of reduction rules, just as in a pure reduction system. However, control is derived from the original expression form by static analysis -- compilation of a program. This derived control is manifested as a stream of instructions just as in a conventional von

Neumann computer. Programmed reduction systems are not so elegant as pure reduction systems, but offer the advantage that we can make use of technology developed over the last 35 years to implement von Neumann computer architectures.

1.1. Graph reduction

In a β-reduction step, a bound variable is eliminated by substitution of a corresponding argument expression found in an application. The argument expression replaces each occurrence of the variable. In the GMD reduction machine [Ber75] β-reduction has been implemented by recopying the matrix of a redex, during which (possibly multiple) copies of the argument expression are substituted for the variable being eliminated. This is a costly operation, whose asymptotic complexity is proportional to the square of the length of a redex.

Multiple copies of an argument expression are required only if expressions are represented as strings. Using a slightly more complex representation of expressions as graphs allows multiple references to a common subexpression to be realized by arcs of a graph incident upon a common target. This form of representation allows us to make fuller use of the architectural primitive of addressable memory. *Graph reduction* then refers to a reduction process in which expressions are represented as graphs, rather than as strings, and there may be multiple references to a shared subgraph. This sharing is extremely important to performance of a sequential evaluator, because it eliminates redundant re-evaluations of copies of a common expression. It can, however, inhibit fully parallel evaluation to some degree.

1.2. Combinator reduction

The combinator reduction architecture proposed by Turner [Tur79] is a pure reduction system based upon Curry's combinatory calculus, and employing graph reduction. Prior to Turner's experiments, the combinatory calculus had not been seriously studied as a model for practical computation. Among those in computing who knew of the model, the extreme simplicity of the combinators was thought to make their use hopelessly impractical, akin to computing with a Tur-

ing machine. Turner found that if the combinators S, K, and I, sufficient to form a primitive basis, were augmented by a few additional dependent combinators that much better performance was obtained from an evaluator when applied to expressions occuring in typical programs. As an expression undergoes reduction, the intermediate expressions generated do not display the combinatorial growth in size that is typical when using only the S,K and I combinators. This has been referred to as a "self-optimizing" property of combinator reduction, and seems to occur as the set of judiciously chosen dependent combinators reaches a critical size.

Following this lead, Hughes [Hug82] explored the idea that one could abandon the restriction to a fixed number of combinators, creating instead customized combinators derived from the expression to be reduced. The representation of an applicative expression using customized combinators may be much more compact and its evaluation can require fewer reduction steps. Of course, the set of reduction rules must be expanded and tailored to the customized combinators. This idea, which Hughes called super-combinators, led to the concept of programmed reduction architectures.

1.3. Programmed graph reduction

In a graph-reduction architecture, the reducer walks through the graph of an applicative expression to locate a redex, chosen according to the computation rule (normal-order or applicative-order) being followed. As it walks the graph, it caches a trail of unreduced application nodes leading to the redex. We show this cache as a stack of pointers in the diagram of Fig. 1a. In a pure combinator reducer, the function symbol f at the lower left corner of the graph is one of a fixed set of combinators. The machine interprets this combinator to rewrite the left corner redex of the graph, as shown in Fig. 1b.

In a programmed graph-reduction architecture, the symbol f can be any defined function symbol. To reduce the application of that function, the reducer executes a program compiled from the definition of the function. The architecture does not restrict the arity of the function to some previously restricted number of arguments; if argument values are needed to compute the

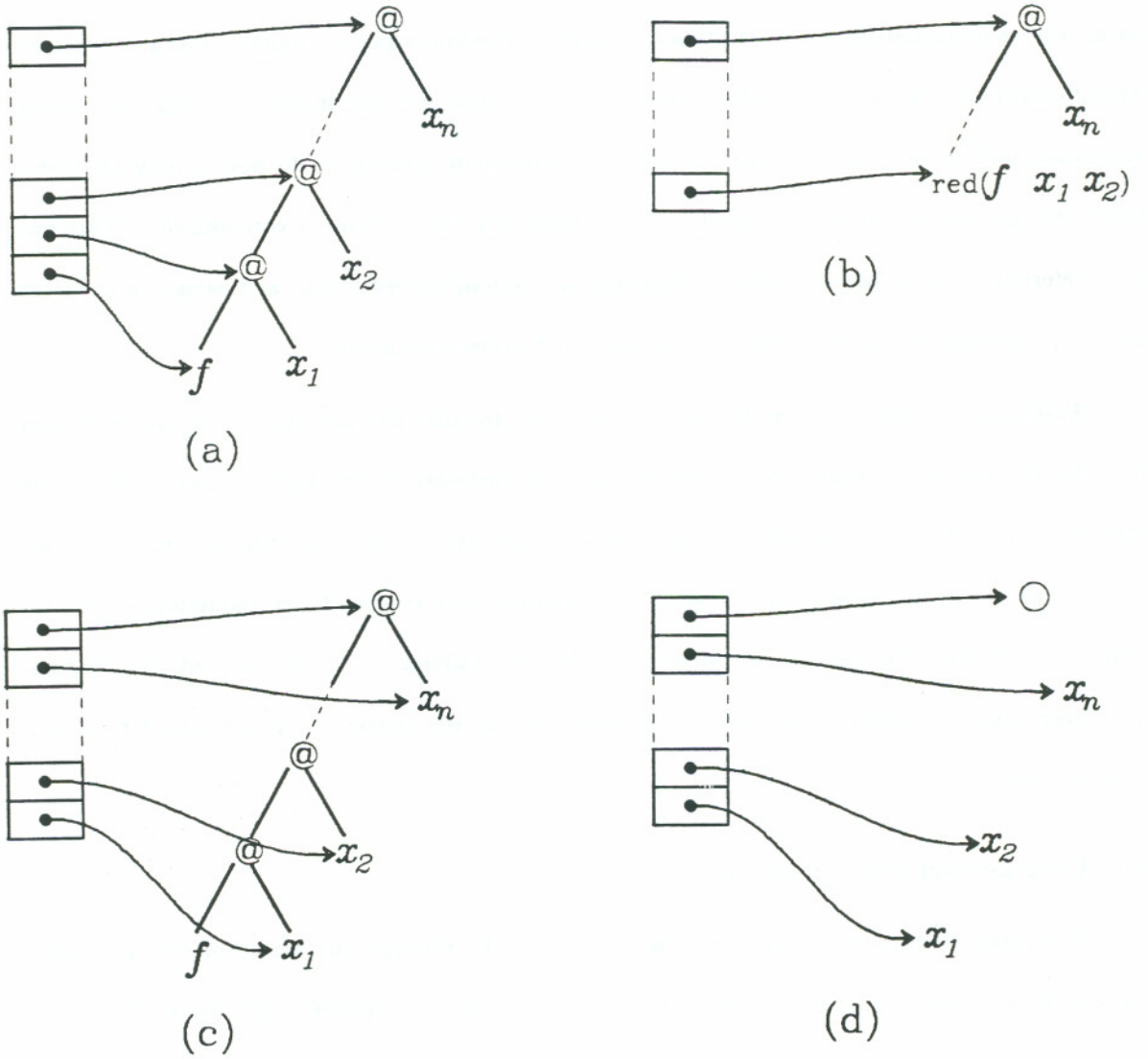

Figure 1 -- illustrating possible traversal stacks

application, the program will direct the reducer to access the needed arguments from the stack of application pointers.

Realizing that this is the use to be made of the application pointers, we can change the configuration of the traversal stack to correspond to Fig. 1c. Here the stack contains pointers directly to the argument expressions. In addition, it maintains a pointer to the principal application that is being reduced, for it is the node that will be replaced to complete the reduction.

Furthermore, in case the graph we have been considering is the direct translation of an expression from the source program text, and this expression is allowed to be evaluated as soon as it is constructed, then the compiler can avoid the steps of constructing the graph explicitly. Graphs for the argument expressions $x_1,...x_n$ must be constructed, but then the compiled code can simply configure the stack to point at the appropriate subgraphs, as shown in Fig. 1d. This represents a considerable economy in computation.

The G-machine described in this paper is a design for a programmed, graph-reduction machine, based upon an abstract architecture defined by Thomas Johnsson and Lennart Augustsson [Joh84, Aug84]. The abstract architecture was used to define an evaluation model for a functional-language compiler which compiles code for a conventional computer. We have refined it slightly with a view toward hardware implementation.

1.4. Non-reduction evaluators

In spite of the fact that the lambda-calculus has been the conceptual model for most applicative languages, until recently evaluators have not used a reduction model. Evaluators have been influenced more strongly by conventional computer architectures than by the language model that they implement. This influence has not been restricted to the idea of using programmed control, but extends to the mode of representation and the use of addressable memory.

In conventional evaluators, such as the SECD machine [Lan63] or a LISP interpreter, *values* are distinguished from *expressions*. Values are given a direct representation but expressions are not. This implies that it is rather cumbersome to deal with unevaluated expressions as data, and has dictated that most evaluators perform applicative-order evaluation (almost) exclusively. A function value is represented by a *closure* object[1], which is a pair consisting of a program text and an environment. An environment is a mapping of identifiers to their values, and the environment component in a closure must bind all of the free (i.e. non-local) identifiers in a function definition.

[1]We shall comment only upon models that correctly implement the lambda calculus.

There are several technical schemes for representing an environment object. In each, reference to the value of a non-local variable involves at least one additional level of address indexing or indirect reference, beyond that required to access a local variable. This by itself does not necessarily have a negative impact upon performance, since hardware indexing and memory caching technologies can mask the overhead. However, the representation of an environment is a data structure that must be constructed in memory. If an environment is not static, then its representation must be built during execution of a programmed evaluation. It has been widely conjectured that the construction of environment representations for closure objects is one of the principal sources of computational overhead in evaluating functional language programs with conventional evaluators.

The primary advantage in performance that graph-reduction appears to have over conventional evaluators is that it is unnecessary to provide explicit data structures to represent environments. All bindings are made uniformly by the process of rewriting a redex of the graph at each reduction step.

2. Architectural issues affecting performance

Any abstract evaluation model can be mapped onto a conventional computing system by programming. New computer architectures are justified if they offer improved performance, increased reliability or if they significantly simplify programming. The primary motivation for the G-machine architecture is that we believe it can significantly improve performance in evaluating applicative expressions.

The dominant factor limiting the performance of any computer system is the overhead of moving data. The metric of distance used to analyze systems is the time required to access data that reside on one functional unit when they are needed by another. Indirect access via a pointer stored in memory may double the cost of access to a datum by requiring two memory fetches instead of one. Cache memories can reduce average access cost by dynamically establishing address neighborhoods. Pipelined operation reduces cost by providing a bounded degree of

parallel movement of data.

The largest overheads in conventional applicative-language evaluators can be attributed to five factors:

1) Indirect reference to variables via data structures that implement environment mappings;

2) Construction of closure frames, with attendant movement of data between processor and memory;

3) Saving and restoring contexts when evaluating function applications;

4) Dynamic memory management.

5) Extensive use of list structures, which implies sequential access paths to data.

A graph-reduction architecture alters the balance of importance of these four factors, and thus provides opportunities to address performance issues with somewhat different mechanisms.

1') Non-local variables are eliminated during compilation of a functional-language program for evaluation by graph-reduction. Local variables are eliminated at the onset of evaluation of a function application, by binding the corresponding argument expressions in place of local variables. In an expression graph, subexpressions are referred to directly by arcs of the graph itself, rather than indirectly through an environment. Since the graph is transformed by update-in-place during evaluation, it is never necessary to employ a run-time data structure to represent an environment.

2') In performing graph reduction, closures are not explicitly constructed, although when normal-order (lazy) evaluation is performed, *suspensions* (suspended applications) must be constructed.

3') The G-machine makes use of an internal register set, and therefore machine state must be preserved while evaluating nested function applications. The machine state consists of its register contents and a control pointer. However, state-saving can be supported largely by hardware, almost eliminating the penalty for a "context switch" when a nested application is

evaluated.

4') In graph reduction, dynamic memory management is required. Graphs derived from recursive definitions can be cyclic, and therefore simple reference-counting is not an adequate memory management strategy. Furthermore, memory allocations may be more frequent that with conventional (applicative-order) evaluation since expressions, as well as values, are directly represented as data. We propose a new, incremental collection strategy based upon a modified reference counting scheme. It can be supported by a parallel processor and should not impose any significant in-line burden upon the reduction processor.

5') The use of arrays in preference to list structures would allow evaluation to benefit from the indexed-address computation hardware available in conventional computers. However, the implementation of a hybrid memory model, combining "flat" addressible segments with list-structure memory, has seemed overly complex to us. We have not attempted to implement arrays.

On the other hand, in a programmed machine, software and hardware can cooperate to program the use of machine registers so that many list-structure constructions can be avoided entirely. The registers of the G-machine are organized as a pair of indexed stacks, so that compilation of code to use the registers effectively is quite simple.

3. The G-machine

The G-machine is a programmable graph-reduction engine that can perform either applicative-order or normal-order evaluations. It employs a tagged, dynamically allocatable list-structure memory (G) in which to store an expression graph undergoing reduction, and a byte-addressable control store (C). The processor consists of an instruction fetch and translation unit (IFTU) that fetches G-code from the control store, translates it to microcode, and delivers a stream of micro-instructions to a processor control unit (PCU). A separate data and address bus (G-bus) connects the G-memory with a stack (P) that holds graph pointers, an arithmetic-logical

unit (ALU) which also has an associated register stack (V), and a small number of special registers. The top segment of the P-stack is supported by hardware registers in the G-machine processor, and some fixed number of cells adjacent to the stack top can be directly accessed by stack operation instructions. The remainder of the P-stack overflows into fast memory.

G-code [Joh84] is a predominately zero-address machine code. When instructions do have operands they are either control addresses in control-transfer instructions, indices for bit-shift instructions, literal data, pointers to constants stored in G-memory, or indices relative to the top of the P- or V-stack. Operation codes are represented in one byte. The mean instruction length gotten by averaging over the code compiled for 35 small programs is 2.18 bytes. Although the mean instruction length is a simplistic measure, it indicates that the code efficiency of the G-machine is probably somewhat better than that of most current generation, general-purpose von-Neumann architectures.

3.1. Organization

The G-machine processor will operate as a slave to a host processor which shares access to its control memory and to the graph memory (see Fig. 2). The host processor will be responsible for sending the G-processor an initialization signal, for loading the control and graph memories, and for signalling the G-processor to begin evaluation. During an evaluation, the G-processor can signal its host, interrupting it to request service. To make a service request the G-processor deposits a the address of a node in G-memory onto the G-bus, from which it will be read by the host processor. Details of the service request are communicated as a graph in G-memory, whose root is pointed to by the address deposited on the G-bus when the request is signalled.

This arrangement allows the G-processor to run free of interrupts. Upon initialization and following a signal of a service request, the G-processor enters a wait state, awaiting a continuation signal from its host. The host is to be realized by a conventional microprocessor. It will supply all necessary operating system services, as well as memory-management functions not directly supported by specialized hardware. Access to the G-bus is arbitrated by fixed priorities that give the

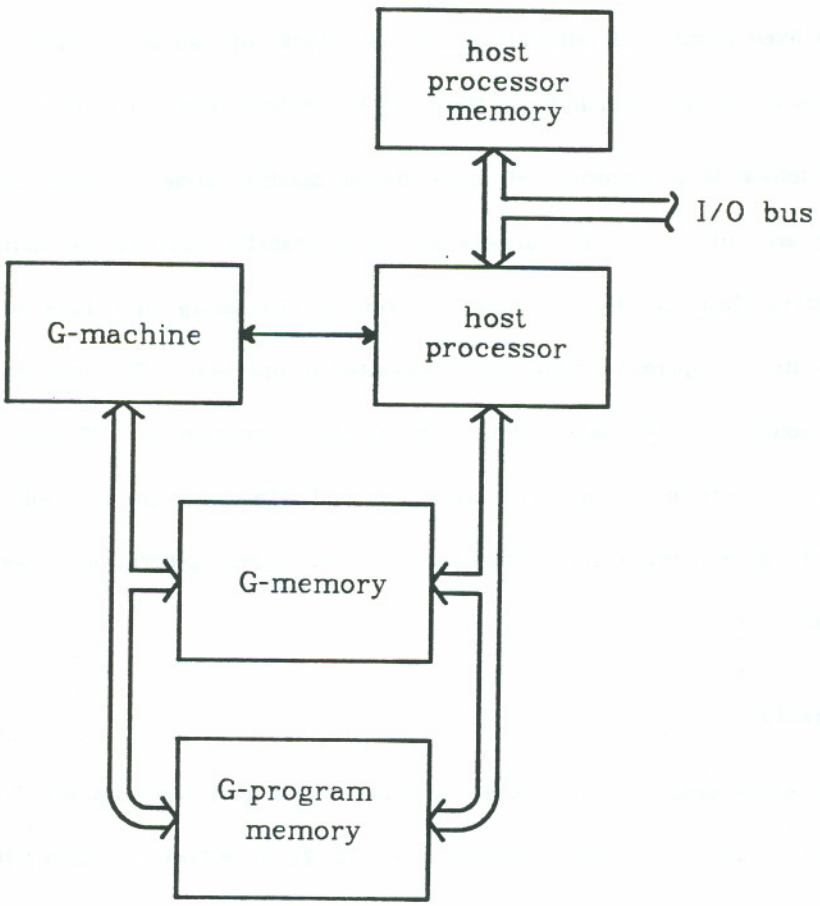

Figure 2 -- Block diagram of the G-machine and host processor

G-processor preference over the host.

The G-machine is designed to provide hardware support for the following aspects of evaluation:

Graph traversal, and access to arguments during evaluation of an applicative expression. The P-stack is loaded during traversal of an application graph, and subsequently holds pointers to the arguments needed during evaluation of the application. The current design of the G-processor holds the topmost 24 cells of the P-stack in a register queue within the processor. Any of these 24 cells can be brought to the stack top by an instruction executable in a single machine cycle, a

fraction of the time needed to fetch a datum from the G-memory.

Functional simulation of the G-machine shows that stack-manipulation instructions account for 20-25 per cent of all G-code issued. These instructions are:

PUSH		-- moves a 32-bit datum from the G-bus to the top of the P-stack
POP		-- removes the top of the P-stack onto the G-bus
COPY	*index*	-- copies the contents of the indexed stack cell to the top of the P-stack
MOVE	*index*	-- removes the cell at the top of the stack and overwrites the indexed cell
ROT	*index*	-- removes the indexed cell from the interior of the stack and places it at the top

Instruction fetch. The IFTU and PCU are designed to function as a buffered pipeline with high throughput and extremely short latency. There is no aspect of the design of these units that is particular to graph-reduction. However, during expansion of G-code instructions into a sequence of micro-instructions considerable pre-fetching of G-code can occur, even along alternative branch paths.

Context switching. A context switch saves or restores the processor state (program counter, P-stack register contents). Since there are no non-local control transfers, contexts are saved and restored according to a LIFO discipline. Context switches can be accomplished at almost no overhead by a hardware implementation that allows the hardware register queue holding the top segment of the P-stack to overflow into fast memory. In such an implementation, the inline overhead of a context switch consists in saving (or restoring) a program counter value on the P-stack. Context switching by sequentially storing or fetching the contents of the P-stack under programmed control has been estimated from simulations to account for 8-12 per cent of total execution time. This overhead is eliminated by the automatic register-queue overflow mechanism.

Dynamic, list-structured memory. Graph-reduction places heavy demands upon dynamic memory allocation because expressions, as well as list-structured data, are represented as graphs. Dynamically allocatable memory is considered an architectural primitive, to be supported by hardware. ALLOC (allocate a node of G-memory) is a primitive instruction to G-memory, just as is READ or WRITE. Furthermore, collection of inaccessible graph nodes has been made the task of the memory manager, rather than the G-processor. The obligation of the processor is limited

to signalling the memory manager i) whenever a reference pointer is written as data into a graph node, so that a reference count can be incremented, ii) whenever a reference pointer in a node is overwritten, so that a reference count can be decremented, and iii) when a function call or return occurs. The first kind of signal accompanies writes to memory and incurs no in-line overhead. The second kind of signal is also associated with some writes, but precedes the write itself in order that write operations are not encumbered with delay. This kind of signal will require an additional memory cycle. The third kind of signal is independent of any direct memory access by the G-processor, and will consume one memory cycle. The memory management scheme is described in a later section.

These, then are the aspects of a reduction architecture that have seemed to deserve hardware-supported implementation. Several of the components, the P-stack and parts of the IFTU in particular, will benefit from VLSI realizations, for they buffer data movements by using moderately sized, dynamic storage structures with a high degree of regularity.

Beyond the aspects of programmed graph-reduction that can be effectively supported by hardware or by microprogramming, there still remains great potential for performance improvement by improving the compilation schemes. When it is safe to perform, applicative-order evaluation will be less costly than normal-order evaluation. An application graph does not have to be explicitly constructed to do applicative-order evaluation. Thus, performance can be improved by a compiler that does strictness analysis to find cases in which applications that would nominally require normal-order evaluation can be safely done in applicative order.

A compiler can further improve performance if it can discover cases in which a node of G-memory can safely be re-used, in preference to allocating a fresh node and leaving the old one to be reclaimed by the memory manager. This will save ALLOC instructions, reduce the rate of memory allocations and relieve load on the dynamic memory manager.

Additional improvement can be obtained if a compiler creates "wrapped" and "unwrapped" code for many functions. A wrapped version of the code text will obtain its arguments from the

graph in G-memory, and will deposit the results there. In many cases the result of a function is a tuple of two, three, or more values which is represented as a short list in G-memory. An unwrapped version of the same function will simply leave its results in the P- and V-stacks of the G-processor. This corresponds to the practice of passing arguments in registers in conventional multi-register machines. A wrapped version of a function can be obtained from an unwrapped version by appending to its code body a suffix to perform the list construction before returning. In evaluating an application of a recursively-defined function, the advantage of executing the code of an unwrapped version is obvious.

3.2. Processor design

The G-processor itself is organized as shown in Fig. 3. The internal data/address bus (G-bus) connects the major functional units of the processor with one another, and through a latch, to the G-memory and the host processor. The ALU implements integer addition, add-with-carry,

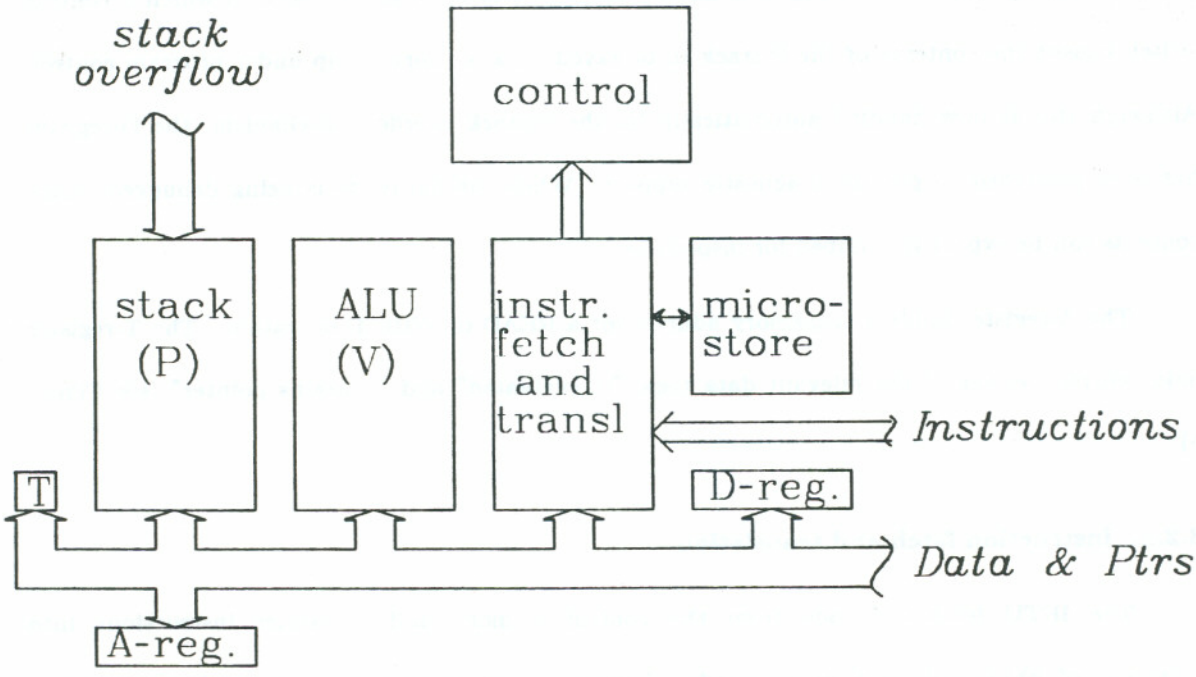

Figure 3 -- functional units of the G-processor

and complementation on signed 32-bit data. It also provides shift operations, byte insertion, and constant zero. There are condition codes for zero, negative, carry generation and overflow generation.

The processor controller (PCU) is responsible for dispatch and distribution of control signals extracted from fields of 20-bit micro-instructions. Its actual decoding function is not much more complicated than that of the Berkeley RISC architecture [PaS82]. Unlike the Berkeley RISC, however, the G-processor is not completely synchronous, and micro-instruction dispatch is subject to the availability of the resources required by the instruction. In particular, this allows the processor to await the availability of G-memory, of the next micro-instruction or literal from the G-code stream, or of completion of an ALU operation. This will accomodate alternative hardware implementation strategies, and it allows some degree of overlap in actual micro-instruction execution. For instance, an ALU operation can begin before a preceding READ or WRITE to G-memory has completed.

The D-register shown in Fig. 3 is an anachronism from an earlier design in which a context switch caused the contents of the P-stack to be saved on a memory dump under program control. Although this is now handled automatically by the P-stack overflow mechanism, the D-register has been preserved to provide diagnostic support. While the hardware is being debugged, stack contents can be explicitly dumped for inspection.

The A-register holds a G-memory address for a READ or WRITE operation. The T-register holds current values of the relevant data tags, "is_evaluated" and "contains pointer" (see Figure 5).

3.2.1. Instruction fetch and translation

The IFTU fetches G-code from the control memory and translates instructions into sequences of 20-bit wide vertical microcode. Several G-code instructions, such as the stack manipulation operations, translate directly into micro-instructions, whereas others, such as EVAL (evaluate a graph) translate into rather complex sequences which themselves contain control flow

instructions.

G-code can contain literal data, G-memory addresses, and jump instructions specifying addresses in control memory. These data are all removed by the IFTU. Literals and G-memory address constants are routed to a literals queue, from whence they can subsequently be moved to the G-bus. Jump addresses are interpreted directly by the IFTU to initialize its code buffers. Thus unconditional jumps are never translated into microcode at all, they simply reinitialize a code buffer into which G-code is fetched [Wil83].

Conditional jump and case-switch instructions are also partially interpreted by the IFTU. A conditional jump specifies a possible alternative code stream to the one currently being fetched. The IFTU maintains multiple instruction buffers (four in the current design) and a new buffer is enabled and given the target address of the conditional jump from which to begin fetching a code stream. The IFTU uses nondeterministic prediction of the path to be taken at the jump, and attempts to fill all enabled code buffers by multiplexed fetches. Case-switch instructions (which will occur quite frequently in code compiled from LML programs) may specify wider than two-way branch alternatives. The multiple code-buffers scheme will accomodate up to four-way case switches.

This scheme's effectiveness will be limited by the available bandwidth to the control memory. Since on the average, each G-code instruction expands into a sequence of several micro-instructions, there should be excess memory bandwidth over that required to support a single stream of G-code instructions. Even when microcode is being consumed at full speed by the G-processor, we expect to be able to fill additional instruction buffers.

Only a single level of nondeterministic jump prediction is supported. When a second conditional jump instruction is encountered in translating a G-code stream, translation stops until execution of the already-translated microcode stream resolves the previous jump.

Conditional jump instructions are translated into micro-instructions that specify the index of one of the enabled code buffers. (See Fig. 4.) When a conditional jump is taken, the IFTU must

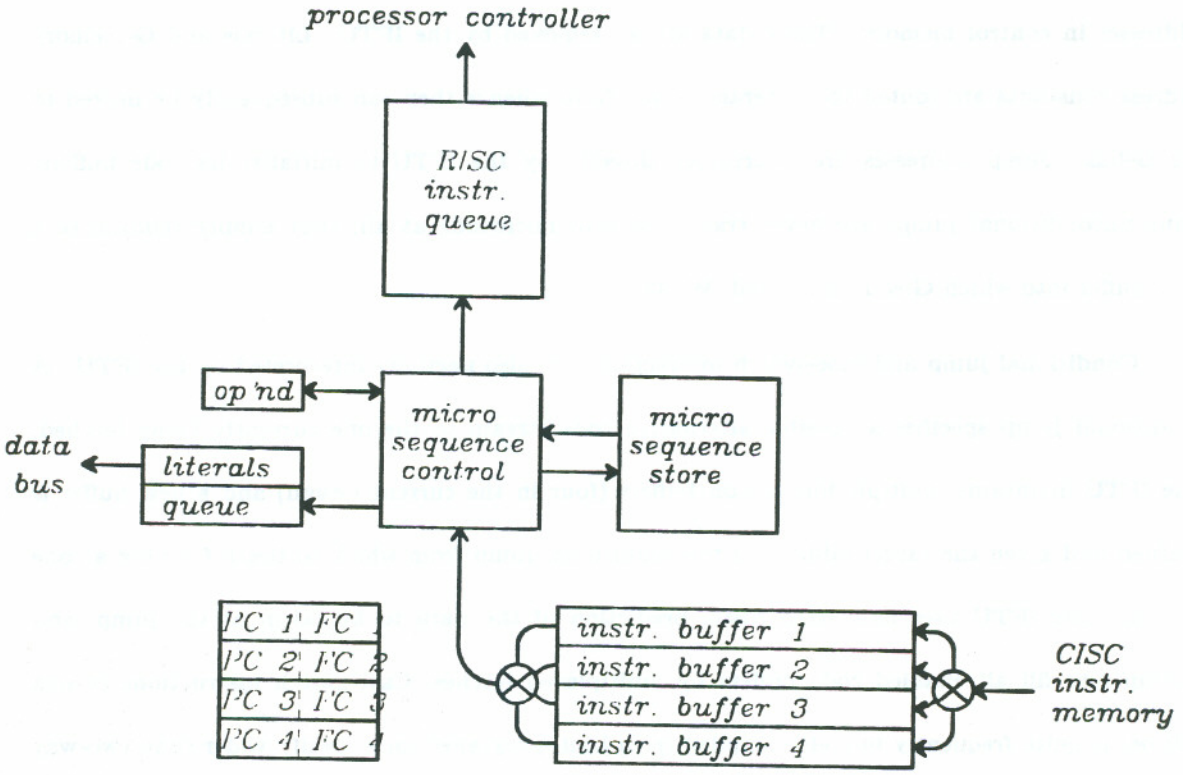

Figure 4 -- instruction fetch and translation unit

flush the queue of translated but not yet executed micro-instructions, for they were produced on the assumption that the jump would not be taken. It must abort translation of the G-code instruction currently in the translation unit, restart translation from the indexed code buffer, and disable all other code buffers. Most of these operations can be done simultaneously, in a single internal clock cycle. Restarting the translation process will require at least two additional cycles before the first micro-instruction can be produced. Fetch of new G-code instructions into the single code buffer that remains enabled after a jump has been taken will be accelerated by the absence of competition.

When a conditional jump is not taken, the action required of the IFTU is much simpler; it needs only to disable all but the currently active code buffer. No interruption of the microcode stream occurs.

Although this activity sounds complicated, it is in fact quite straightforward to support with hardware. It can have a considerable advantage if programs exhibit loops with conditional exit, as is the case when tail-recursive function definitions are compiled. The translation unit expands the code loop as if it were straight-line code, punctuated occasionally by conditional jump instructions. The only taken jump will be the one that finally terminates execution of the loop.

3.3. G-memory

A node of G-memory consists of a pair of 32-bit data cells, plus six tag bits and 18 bits of reference-count fields used in memory management. The layout of a node is shown in Figure 5. The G-memory controller performs most functions of dynamic memory management automatically, with very limited cooperation required of the G-processor. The controller is implemented partly by special hardware, and partly by software running on the host processor.

An address selects a node, and the low-order bit of a node address further selects the first or second data cell. When a cell is fetched or written, certain of the node tags are also fetched or written.

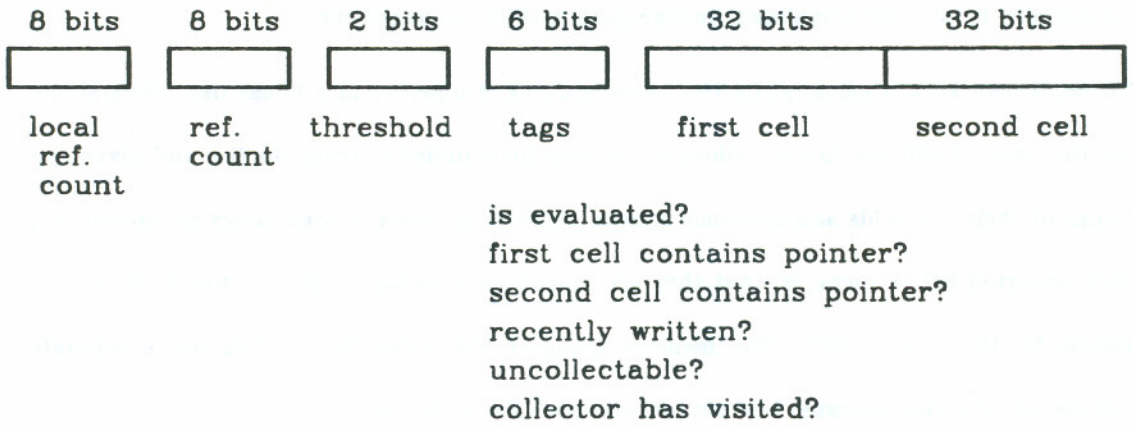

Figure 5 -- a node of G-memory

Graph-reduction requires a tagged memory architecture. A graph component may be shared, and can be reduced by the first reference that demands its value. Thus any reference to a graph component may either find it unevaluated or find that it has been evaluated by a previous reference. To allow this, each node of G-memory carries a tag bit, "is_evaluated" to tell whether its evaluation has occurred.

When a tagged memory architecture is being contemplated, it is useful to consider other tags that might be useful. We find it useful to allocate a pair of tag bits to designate the contents of the first and second data cells as representing either basic values or pointers. This allows more compact data representations to be used in the presence of generic functions.

The "recently-written" bit is automatically set each time the reference count of the node is modified. This bit can be tested or cleared by the memory manager. This provides the explicit synchronization required between the G-processor and the concurrent garbage collector.

3.4. Memory management

The basic strategy of G-memory is to provide a storage structure that behaves as a dynamically allocatable, list-structure memory. The basic operations on such a memory are to allocate a new node and to read or write to a designated cell of an addressed node. The G-processor which uses this memory is (almost) completely unconcerned with its management.

To implement such a memory, its free nodes are linked together in a linear list. Several elements of the head of this list are pre-allocated, having been unlinked from the list and given initial settings of their tag fields and reference counts. Pointers to pre-allocated nodes are stored in a hardware-supported FIFO queue, so that they are immediately available in response to an allocation request by the G-processor. The memory manager maintains for its own use a separate record of the set of nodes allocated but not yet examined for collection.

Reference counts are maintained by the memory manager whenever a node of G-memory is written. Reference counts only need to be decremented when a node is overwritten by the

UPDATE of an application by its value, or when an ancestor in the graph is collected. Only the former action is initiated by the G-processor, and it is preceded by a TRASH-node memory reference. This is the only explicit in-line overhead of the reference counting scheme. Each WRITE instruction in which the datum is a pointer (indicated by the is-pointer bit) results in incrementation of the reference count of the node pointed to.

Collection is triggered when the G-processor signals return from the code of a function application. At that time, the collector removes from the record of nodes previously allocated, the set of nodes allocated since execution of the function call began, and at its leisure, examines each such node for collectability. Any node whose reference count is zero is immediately collectable. As collection proceeds, nodes pointed to by a collected node have their reference counts decremented and are added to the list of eligibles. Any node whose reference count has ever overflowed is forever uncollectable. Other nodes with non-zero reference counts are subjected to a local graph traversal to determine whether the reference count can be accounted for solely by pointers from within the subgraph, forming cycles. If this is true of every node in the subgraph, and no such node has had its reference count modified (by action of the G-processor) during the collector's examination of the subgraph, then the subgraph is collectable.

This is a somewhat simplified summary of a modified reference-count scheme invented by Ashoke Deb [Deb84]. Its advantages are that it is incremental, imposes no additional overhead on the management of acyclic graphs, requires only the simplest of synchronization for concurrent implementation, maintains comparable locality of reference to that of the G-processor, and does not suffer performance degradation when the ratio of occupied to free memory nodes is high. These are considerable advantages. In addition, Deb's simulations show the computational cost of this collection scheme to be considerably lower than that of conventional mark-sweep collection, even when memory occupancy is quite low.

4. Present state of the project

As of this writing, the architecture of the G-machine is complete and an early version has undergone functional simulation. The results of this simulation [Sar84] showed that stack manipulation instructions occur more frequently than any other type, and provided some statistics on the use of memory and the dynamic occurrence of jump instructions. These results reinforced our decision to provide direct hardware support for the P-stack, and a fast, low-latency, pipelined IFTU. They indicated that context switching by unloading of the processor's stacks under programmed control would be a large source of overhead, and led to the decision to implement automatic overflow management of the P-stack.

A micro-architecture has also been fully designed and an initial edition of the microcode written. A full simulation is currently being constructed. We expect to obtain significant performance data from this simulation.

A VLSI implementation of the P-stack has also been designed and simulated, and is currently in fabrication. Work continues on detailed design of the G-memory and memory manager.

Acknowledgements

The author wishes to thank Thomas Johnsson and Lennart Augustsson for many stimulating discussions on functional language implementation; Shreekant Thakker for pointing out some pitfalls to be avoided in planning a hardware implementation, and Ananda Sarangi, Linda Rankin, Mark Foster, Richard Vireday and Shyue-Ling Kuo for their contributions to design and simulation of the G-machine.

References

[Aug84] Augustsson, L., A compiler for Lazy ML, *Proc. of 1984 ACM Conf. on Lisp and Funct. Prog.*, August, 1984, pp. 218-227.

[Ber75] Berkling, K., Reduction languages for reduction machines, *Proc. IEEE Int. Sympos. on Computer Arch.*, pp. 133-140, Jan. 1975.

[Cla80] Clarke, T.J.W., Gladstone, P.J.S., MacLean, C.D. and Norman, A.C., SKIM - the S, K, I reduction machine, *Proc. of 1980 ACM Lisp Conf.*, pp. 128-135, August, 1980.

[Deb84] Deb, A., An efficient garbage collector for graph machines, Oregon Graduate Center, Tech. Rept. No. CS/E-84-003, August, 1984.

[Hug82] Hughes, J., Supercombinators: a new implementation method for applicative languages, *Proc. of 1982 ACM Conf. on Lisp and Funct. Prog.*, August, 1982, pp. 1-10.

[Joh84] Johnsson, T., Efficient compilation of lazy evaluation, *Proc. of 1984 ACM SIGPLAN Conf. on Compiler Constr.*, June, 1984

[LMO84] Lampson, B.W., McDaniel, G. and Ornstein, S.M., An instruction fetch unit for a high-performance personal computer, *IEEE Trans. on Computers C-33*, No. 8 (Aug. 1984), pp. 712-730.

[Lan63] Landin, P.J., The mechanical evaluation of expressions, *Comp. J. 6*, (1963-4) p. 308.

[Mag79] Mago, G.A., A network of microprocessors to execute reduction languages - Parts I and II, *Int. J. of Comp. & Inf. Sci. 8*, No. 5 (Oct. 1979), pp. 349-385, No. 6 (Dec. 1979), pp. 435-471.

[PaS82] Patterson, D.A. and Sequin, C., A VLSI RISC, *Computer 15*, No. 9 (Sept. 1982), pp. 8-21.

[Sar84] Sarangi, A.G., Simulation and performance evaluation of a graph reduction machine architecture, M.S. thesis, Oregon Graduate Center, July, 1984.

[Sch85] Scheevel, M., NORMA, a normal-order combinator reduction machine, colloquium presented at Oregon Graduate Center, January, 1985.

[SCN84] Stoye, W.R., Clarke, T.J.W. and Norman, A.C., Some practical methods for rapid combinator reduction, *Proc. of 1984 ACM Conf. on Lisp and Functional Prog.*, pp. 159-166, August, 1984.

[Tur79] Turner, D.A., New implementation techniques for applicative languages, *Software - Prac. & Exper. 9*, No. 1 (Jan. 1979), pp. 31-49.

[Wil83] Wilkes, M.V., Keeping jump instructions out of the pipeline of a RISC-like computer, *Computer Arch. News 11*, No. 5 (Dec. 1983), pp. 5-7.