N89-19840

ORIGINAL PAGE IS
OF POOR QUALITY

GARBAGE COLLECTION CAN BE MADE REAL-TIME AND VERIFIABLE

James H. Hino
Integrated Inference Machines, Inc.
1468 East Katella Avenue
Anaheim, CA 92805

Charles L. Ross
Integrated Inference Machines, Inc.
1468 East Katella Avenue
Anaheim, CA 92805

ABSTRACT

An efficient means of memory reclamation (also known as Garbage Collection) is essential for Machine Intelligence applications where dynamic storage allocation is desired or required. Solutions for real-time systems must introduce very small processing overhead and must also provide for the verification of the software in order to both meet the application time budgets and to verify the correctness of the software. This paper proposes Garbage Collection techniques for symbolic processing systems which may simultaneously meet both real-time requirements and verification requirements.

The proposed memory reclamation technique takes advantage of the strong points of both the earlier Mark and Sweep technique and the more recent Copy Collection approaches. At least one practical implementation of these new GC techniques has already been developed and tested on a very-high performance symbolic computing system.

Complete GC processing of all generated garbage has been demonstrated to require as little as a few milliseconds to perform. This speed enables the effective operation of the GC function as either a background task or as an actual part of the application task, itself.

INTRODUCTION

Scientists and engineers may argue over the true nature of intelligence; whether it be human or artificial.

However, there is little argument that the capture and recording of useful knowledge does require more memory and more storage space than does the relatively straight-forward representation of numerical or alpha-numeric data entries.

The practical application of new Machine Intelligence technology to today's and tomorrow's Aerospace and Defense problems has become an important strategic issue to all of us. Application software programs are, therefore, becoming larger and more complex.

We must be concerned with the high-priority problems of (1) developing effective software which can perform its tasks quickly enough to meet demanding mission requirements; of (2) developing these programs in a timely and affordable manner; and of (3) verifying the correctness and the predictability of the final operational programming.

Simultaneous solutions to all of these needs is extremely challenging. The first priority need must be meeting the anticipated mission requirements. Those requirements include the capability of combined application software / hardware processor systems to produce essential information in time to make critical decisions or to control dynamic processes.

Real-time, knowledge-based systems programs, in particular, must accept a wide variety of types of data, including both numerical information and non-numerical information.

Non-numerical or symbolic data representations can easily include data items and associated data values which can vary enormously in terms of memory storage needs. A considerable waste of available real system memory capacity can occur unless dynamic memory allocation of variable size memory blocks is supported. Several languages including C, LISP, and ADA allow for dynamic allocation and de-allocation of memory.

For C, this task is left up to the programmer to handle as a part of the creation of the application software. In LISP, the task has been assigned to the designers of the LISP environment (includes the operating system) for a particular processor. This choice has off-loaded this demanding task from each individual programmer. Thereby reducing the possibility of unanticipated program flaws from this potential error source.

In the ADA language, the assumption is made that either the application program or its operating system (or both) may perform the memory reclamation (Garbage Collection) function, since the ADA Language Reference Manual does not specify that an ADA implementation must handle it.

Early experiments to allow programmers to allocate and de-allocate storage in LISP was disasterous. It proved to be extremely difficult for the programmer to know when all necessary data items are no longer referenced by any system process, program, or other data item. Some of the more intractable problems found in some C programs may be a by-product of the reliance on the application programmer to program this function, without leaving an unsuspected trap under certain program conditions. When large, complex programs are written by many individual programmers, the risk may substantially increase.

Some of the extra power and flexibility of the LISP language adds to the creation of considerable temporary results in main memory. Much of which is quickly no longer referenced and is, therefore, no longer required. This increases the importance of solving the GC problem. If memory is not reclaimed, free memory locations will soon become unavailable and program execution will stop.

This paper describes an approach to the design of a real-time GC mechanism. The proposed approach was demonstrated in the demanding LISP environment. It should be effective for ADA, as well. The performance tests were run using an implementation of the design for a uniprocessor architecture. The results should be appropriate for single processor systems or a system consisting of several individual processors, each of which are running separate application programs which co-operate together to meet a collective series of concurrent mission information processing needs.

The described approach may or may not be directly transferrable to the design of a multi-processor system, or be optimum for such a configuration. Additional on-going research will assess such feasibility and effectiveness.

STORAGE MANAGEMENT

Both data and program statements in LISP are represented in terms of symbolic expressions (S-expressions). S-expressions often appear as lists of items enclosed in parentheses. An S-expression is either an "atom", a list of S-expressions, or a "dotted-pair" of S-expressions. An atom is either a "numeric atom" such as an integer or a floating point number, or a "literal atom" which is a string of characters beginning with a alphabetic letter and containing other letters, digits, or a few other characters.

Atoms may be put together to form more complicated S-expressions using either a dotted-pair construction or a list construction. List construction is far more common in actual usage of LISP. S-expressions can be further combined with other S-expressions to build larger ones. Table I shows examples of a few of the various types of symbolic expressions, along with the definition of a symbolic expression and a list.

A fundamental assumption of LISP is that at any point in a computation process all memory cells (containing either programs or data) are reachable through a chain of pointers from a fixed set of known cells or base registers. Garbage Collection approaches must deal with the extensive series of relationships of data and programs which can exist at any time.

MEMORY RECLAMATION APPROACHES

The three basic forms of memory reclamation are "Mark and Sweep", "Copying Collection", and "Dynamic Pools". The following sections will briefly discuss the advantages and disadvantages of each approach for implementing real-time systems. The concept of a "workspace" is used in these discussions. A workspace is the collection of programs and data for any application as well as the entire system code. At any time the workspace may contain unreferencable objects which is called "garbage".

MARK AND SWEEP

Mark and Sweep is also known as Stop and Collect. This technique requires that the processor perform successive passes through all of referenced memory. A specific data structure might be referenced several times. In the first pass, all accessable objects are marked. Then all marked objects are forwarded. The forwarding phase updates all pointers to their new locations. Finally, all marked objects are moved to their final destinations. Since objects are copied over each other, the application task may not run while the garbage collection is taking place. The collection process is activiated when there is insufficient free memory to allocate an object or when requested by the application (a forced or commanded GC procedure).

This process must be performed over all modifiable objects. The process is verifiable in time and correctness if a forced garbage collection is commanded at a predictable place in the application program. A forced garbage collection is often desireable with this type of collection since the time required to collect garbage increases with the amount of garbage in the workspace.

A drawback to the Mark and Sweep technique is that the time required for most machines can take many seconds or even many minutes. Hardware support for garbage collection functions, on even fast machines, has typically still fallen short of the requirements for real-time applications.

COPYING COLLECTION

Copying Collection is a popular form of memory reclamation used by several LISP machines. A copying collector splits memory into two parts, known as hemispaces. Accessable objects are copied from one hemispace into the other, leaving a forwarding pointer behind in its place. When all accessable objects are copied, the direction of copying reverses. The process of changing the direction of copying is called a "hemispace swap". This method of garbage collection can better approach real-time since it only copies a small amount of memory at any given time wherein the application program is stopped.

A key problem associated with copying collection is that it introduces additional uncertainty into the application processing time. Performance can be unpredictable since the actual time required is dependent upon when a hemispace swap occurs.

Another parameter that affects the variability of processing time of an application is the amount of information that is being copied between hemispaces. This quantity is a function of how much memory is being utilized versus how much free memory exists, and is not constant over time.

The final aspect of copying collection that affects the variability of processing time is that when an object is moved, all references to that object must traverse an indirect pointer to reach the desired object.

It is generally thought that a Copying Collection approach requires less overhead than a Mark and Sweep technique since the Mark and Sweep process passes through memory three times. This is not necessarily true. It depends upon the implementation, and especially the effective use of tag bits available in a tagged architecture.

One final note of significance is the amount of memory required to implement a copying collection approach. Since the available memory must be divided into two hemispaces (a "FROM" Space and a "TO" Space), it can take up to twice the amount of heap memory as other GC approaches.

DYNAMIC POOLS

Languages where commands to deallocate discarded memory is required , can use a scheme where there exist dynamic pools of allocated and available memory. This scheme works well if allocations are of a constant size. If allocations are of varying sizes, memory fragmentation exists. Fragmentation will cause compaction to be required. The time for compaction is a function of how much memory is required and how memory is fragmented. The process of searching for free memory occurs at each allocation. This makes verification of time budgets difficult, if not impossible.

A NEW HYBRID APPROACH

Integrated Inference Machines, Inc. has designed and implemented a memory reclamation technique that takes advantage of the strong points of the Mark and Sweep and Copying Collection approaches. It has been called the SCORE GC. SCORE stands for Stop-and-Collect, Optimizing, Real-time, Ephemeral garbage collector.

The garbage collector is a callable microcode routine that is invoked by a special opcode. The collector utilizes tag bits associated with a hardware-supported tagged architecture machine. Two bits of the 8-bit tag associated with each word in memory is used to support the GC function.

The SCORE garbage collector separates memory into "Static Space" and Heap Space. All objects in Static Space do not move and the memory they occupy does not need to be reclaimed. All new objects are allocated from "Free Space" into "Heap Space". Objects in Static Space need not be read-only, but when they are modified to be a pointer to an object in Heap Space, the address of this location must be saved for the garbage collection process. Figure 1 shows how memory is partitioned.

INFORMATION UNAVAILABLE

To begin the Mark Phase of a garbage collection procedure, the list of objects in Static Space is added to the "Mark Seed"(the mark seed is typically the execution stack as well as objects required to handle asynchronous events such as errors or interrupts). This is required to ensure that objects in Heap Space that are only referenced from Static Space are properly marked. All objects in Heap Space are then collected, using a modified Stop and Collect algorithm. The mark phase is also modified to stop if an attempt to mark an object in Static Space is made.

The Forwarding Phase operates, normally, on Heap Space only. This algorithm provides compaction which can make objects move. Therefore, when Heap Space is forwarded, the locations in Static Space that reference objects in Heap Space are forwarded to reflect the new location of the object in Heap Space. Finally, the Compaction Phase operates normally on Heap Space.

The setting of the boundary between Static and Heap Space, and the timing of the garbage collection process is a function of the application. The optimal partitioning places unmodified objects and objects which do not contain pointers into Static Space. All other objects are placed into Heap Space.

For real-time applications, the timing of the garbage collection is forced by the application program on a regular basis. This collection is placed at the end of one iteration of the application. This is typically where required references to temporary objects is at a minimum.

In order to meet the requirements of real-time applications, speed as well as verifiability of performance is required. SCORE memory reclamation does not occur in the background. Repeated timings of an application, as well as the time to reclaim garbage is repeatable, down to the number of machine cycles. Every run of an application with a given environment of data inputs is identical to the previous and to the next. Memory locations are not altered in ways that cannot be reproduced.

The SCORE GC also can be operated in an emphemeral mode. In this mode, garbage collection is performed when a small amount of memory is used. As objects survive collection, they move into Static Space. Intermediate spaces can be added with different rates of collection to provide additional ephemeral quality.

PERFORMANCE RESULTS

A single SCORE GC cycle has a minimum runtime of under 2 milliseconds. The SCORE collector requires a very small percentage of processing time do perform its functions. An average GC overhead of approximately five percent of the application runtime is predicted. The average number is useful since some tasks may create little or no garbage, while others will generate a great deal.

Of equal importance, the resulting GC overhead for a given application is measureable and is repeatable. Tables II, III, and IV show the results of GC tests performed by the author using a very high performance symbolic computer, the SM45000 (developed and manufactured by IIM).

Table II identifies a series of benchmark programs from the Gabriel Benchmark suite used for the evaluation. The suite contains benchmarks known to produce little or no garbage as well as ones which produce a significant amount of garbage. The benchmarks were timed for conditions of No Garbage Collection overhead at all (Table II), operation of the SCORE GC in real-time mode (Table III), and operation of the SCORE GC in the ephemeral mode (Table IV).

INFORMATION UNAVAILABLE

INFORMATION UNAVAILABLE

CONCLUSION

The power and flexibility of dynamic memory allocation and de-allocation can be made a part of real-time systems. Memory reclamation (garbage collection) technology has advanced to the point where fast, predictable memory management processing can accomodate these requirements. Verifiability of the resulting dynamic memory application software does not have to be sacrificed to an essentially background processing task which can, in turn, alter the dynamic memory states and defy repeatability.

The full performance of the technique makes significant use of two of the tag bits within the 8-bit tag field associated with each 32-bit word in memory.

The absolute GC processing times can be reduced still further by speeding up the symbolic processor, itself.