INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

Cache Performance of Chronological Garbage Collection

by

Yuping Ding

A thesis submitted to
the Faculty of Research and Graduate Studies
in partial fulfillment of the requirement for the degree of

Master of Science
in Computer Science

Department of Computer Science
School of Mathematical Sciences
Lakehead University
Thunder Bay, Ontario

October 8, 1999

ABSTRACT

This thesis presents cache performance analysis of the Chronological Garbage Collection Algorithm used in LVM system. LVM is a new Logic Virtual Machine for Prolog. It adopts one stack policy for all dynamic memory requirements and cooperates with an efficient garbage collection algorithm, the Chronological Garbage Collection, to recycle space, not as a deliberate garbage collection operation, but as a natural activity of the LVM engine to gather useful objects. This algorithm combines the advantages of the traditional copying, mark-compact, generational, and incremental garbage collection schemes.

In order to determine the improvement of cache performance under our garbage-collection algorithm, we developed a simulator to do trace-driven cache simulation. Direct-mapped cache and set-associative cache with different cache sizes, write policies, block sizes and set associativities are simulated and measured. A comparison of LVM and SICStus 3.1 for the same benchmarks was performed.

From the simulation results, we found important factors influencing the performance of the CGC algorithm. Meanwhile, the results from the cache simulator fully support the experimental results gathered from the LVM system: the cost of CGC is almost paid by the improved cache performance. Further, we found that the memory reference patterns of our benchmarks share the same properties: most writes are for allocation and most reads are to recently written objects. In addition, the results also showed that the write-miss policy can have a dramatic effect on the cache performance of the benchmarks and a write-validate policy gives the best performance. The comparison shows that when the input size of benchmarks is small, SICStus is about 3-8 times faster than LVM. This is an acceptable range of performance ratio for comparing a binary-code engine against a byte-code emulator. When we increase the input sizes, some benchmarks maintain this performance ratio, whereas others greatly narrow the performance gap and at certain breakthrough points perform better than their counterparts under SICStus.

ACKNOWLEDGMENTS

I would like to thank my supervisor, Dr. Xining Li, not only for his help during this research, but also for his support during the years I spent at Lakehead University.

I am indebted to Yin Li for accepting to be a reader on short notice, and for many suggestions to improve the quality of this thesis. Thanks also to Yifei Wang for his help during this work.

I am grateful to the National Science and Engineering Council of Canada for its financial support.

I would like also to think my wife for her support and encouragement.

Finally, I would like to thank my external examiner Dr. Yao and internal examiner Prof. Black for their comments.

List of Figures and Tables

Contents

Chapter 1. Introduction

1.1 Motivation

As the speed gap between processor and main memory chips is widening, cache performance is becoming more important in implementing programming languages and designing garbage collection algorithms. In addition, logic and functional programming languages, such as Prolog and Lisp, typically manipulate large data structures with complex inter-dependencies. Thus automatic storage reclamation is essential for practical implementations. The Logic-inference Virtual Machine (LVM) is our newly designed execution model for Prolog. It abandons the heap/stack memory architecture used in the traditional Prolog implementations. Instead it adopts a single stack policy and embeds an efficient garbage collector, Chronological Garbage Collection (CGC), as a part of its engine.

We have implemented an experimental LVM emulator that includes the CGC algorithm (about 300 lines of C-code) as a core part of the virtual machine engine. Our benchmarks show that the LVM has low run-time overhead, good virtual memory and cache performance, and very short, evenly distributed pause times. Some benchmarks even revealed that the CGC improves the program's cache performance by more than enough to pay off its own cost.

Related problems of cache performance have been widely studied by other researchers. From their measurements of four Scheme programs, Wilson et al. [1] first suggested that garbage collectors could be applied to improve the performance of caches. Zorn [2] measured the cache performance of four large Lisp programs running with a

noncompacting mark-and-sweep collector and a more traditional copying collector in various configurations, and showed that the data-cache miss ratios of the programs were improved by the collectors. Reinhold [3] measured the cache performance of five large Scheme programs, and concluded that garbage-collected programs written in a mostly-functional style should perform well with simple linear storage allocation and an infrequently-run generational compacting collector. He showed that, as long as allocation misses are not a problem, the use of large memory areas could actually be good for cache performance on direct-mapped caches, because the references tend to be spread evenly between cache blocks, thus minimizing conflict misses. Jouppi [5] classified cache architectures into four classes: fetch-on-write, write-validate, write-around and write-invalidate. Goncalves [4] studied the cache performance of a set of ML programs in SML/NJ, and reported measurements of miss ratios with varying cache sizes, block sizes, associativities and write miss policies.

This thesis seeks to verify and validate our experimental results, and to find important factors influencing the performance of the CGC algorithm.

1.2 Overview

This thesis presents the cache performance analysis of the CGC algorithm used in LVM. In order to verify and validate our experimental results, and find important factors influencing the performance of the CGC algorithm, we developed a trace-driven cache simulator. In this work, direct-mapped cache and set-associative cache are simulated and measured. To determine the extent to which the cache performance of the test programs has been improved under the CGC algorithm, we have simulated benchmarks with

different cache sizes, write policies, block sizes, and set associativities. As to cache write policies, we consider only three architectures, fetch-on-write, write-validate and write-around in our simulation.

The organization of the thesis is: Chapter 2 makes a survey of garbage collection algorithms. Section 2.1 introduces three ways of storage allocation. Section 2.2 explains what garbage collection is and why we need it. Section 2.3 presents and compares the basic algorithms. Section 2.4 discusses generational garbage scheme and its advantages. Section 2.5 summarizes garbage collection algorithms used in typical Prolog implementations.

Chapter 3 discusses the LVM system and the Chronological Garbage Collection (CGC) algorithm. After summarizing the major differences between the Warren Abstract Machine (WAM) and our LVM system in section 3.1, the representation of logic terms in the LVM system is discussed by comparing Program Sharing (PS), Structure Sharing (SS) and Structure Copying (SC) in section 3.2. Then section 3.3 sketches the memory organization of the LVM. Section 3.4 presents the CGC algorithm and analyses several of its important aspects.

Chapter 4 presents cache architectures. It begins with the principle of caches, in section 4.1. Section 4.2 describes cache architectures and introduces related terms and concepts. Section 4.3 studies important parameters in cache design. Section 4.4 focuses on the write strategy, which greatly affects the cache behavior. Finally, section 4.5 presents a table to show common cache organizations of some popular architectures.

Chapter 5 describes the design of our trace-driven cache simulator. Section 5.1 presents design parameters. Direct-mapped cache and set-associative cache are simulated.

Section 5.2 introduces the trace-driven cache simulation. Section 5.3 discusses how to modify the LVM emulator to include the cache simulator. Finally, section 5.4 shows the simulation algorithms.

Chapter 6 analyzes the experimental results revealing the cache performance of CGC algorithm. Section 6.1 introduces benchmarks and experimental environments used in this study. Section 6.2 shows the memory references of a user program with and without garbage collection, as well as that of the collector. Section 6.3 discusses a proper range for selecting a cache-limit. Section 6.4 analyzes the experimental results and finds evidence to support the conclusion that the CGC improves program cache performance by enough to pay off its own cost. Section 6.5 considers how different cache parameters affect cache performance and shows memory reference patterns of the benchmarks. Section 6.6 gives a comparison of the LVM system and the SICStus system.

Finally, Chapter 7 summarizes the main results and concludes this study.

Chapter 2. A Survey of Garbage Collection Algorithms

2.1 Storage Allocation

In the history of storage management implementation, there have been three storage management schemes: *static allocation, stack allocation and heap allocation.*

Static allocation is the simplest policy. Under this policy, all names in the program are bound to storage locations at compile-time; and thus all the bindings do not change at run-time. With stack allocation policy, an activation record or frame is pushed onto the system stack when a procedure is called, and popped when the procedure returns. Since different activations of a procedure do not share the same bindings for local variables, recursive calls become possible. Heap allocation introduces more flexibility. Unlike the last-in, first-out rule of a stack, heap allocation allows data structures to be allocated and deallocated in any order. Therefore, it is possible for activation records and dynamic data structures to outlive the procedure that created them.

2.2 Garbage Collection

In modern storage management, stack allocation takes care of dynamic memory requirements related to procedure calls and returns, while heap allocation is responsible for all other dynamic memory requirements. Garbage collection (GC) is the automatic management of dynamic storage.

A program can directly manipulate values held in processor registers, on the program stack and in global variables. Such locations holding references to heap data form the *roots* of the computation [6]. The user program should access the dynamically

allocated data through the roots directly or by following chains of pointers from the roots. An individually-allocated piece of data in the heap will be called a cell (or an object). A cell is live if its address is held in a root, or there is a pointer to it held in another live cell. More formally, define –> as the 'points-to' relation: for any cell or root M and any cell N, M –> N if and only if M holds a reference to N. The set of live cells is the *transitive referential closure* of the set of roots under this relation [6],

$$live = \{N \in Cells \mid (\exists r \in Roots.\, r \rightarrow N) \vee (\exists M \in live.\, M \rightarrow N)$$

The rule above implies that the storage mechanism's view of the liveness of the graph of objects in the heap is defined by *pointer reachability*. Dynamically allocated storage may become unreachable. Objects which are neither live nor free are called *garbage*. Algorithms for freeing up dynamic memory automatically are called garbage collection.

Why do we need garbage collection? First of all, explicit storage allocation creates unnecessary complications and subtle interactions. Explicitly deallocating a cell may render some cells inaccessible. These inaccessible cells are called *space leaks*. A pointer referring to memory that has been deallocated is called a *dangling pointer*. Failing to reclaim memory at the proper point may lead to slow memory leaks, with unreclaimed memory gradually increasing until the process terminates or swap space is exhausted. On the other hand, reclaiming memory too soon can lead to very strange behavior, because an object's space may be reused to store a completely different object while it still can be reached by existing pointers. The same memory may therefore be interpreted as two different objects simultaneously and updates to one cause unpredictable mutations of the other.

Next, garbage collection is necessary for modular programming to avoid introducing unnecessary intermodule dependencies. For example, in object oriented languages, garbage collection uncouples the problem of memory management from class interfaces, rather than dispersing it throughout the code.

Furthermore, garbage collection may be a language requirement. Programs written in Prolog or Lisp typically manipulate large data structures with complex inter-dependencies, so it is essential to collect garbage in these languages.

Some works have suggested that a considerable proportion of program development time may be spent on fixing memory management bugs. Effective garbage collection as a software tool is certainly necessary, because it can relieve the programmer of the burden of discovering memory management errors by preventing the occurrence of this type of errors.

Of course, garbage collection has it own costs in both time and space. Also, although garbage collection removes the two classic bugs of explicit storage management, dangling pointers and space leaks, it still might suffer from its own errors.

2.3 Garbage Collection Algorithms

Garbage collection automatically reclaims the space occupied by data objects that the running program can never access again. The work a garbage collector does can be divided into two phases:

1. Garbage detection: distinguish the live objects from the garbage in some way, and
2. Garbage reclamation: reclaim the garbage objects' storage, so that the running program can reuse it.

These two phases may be functionally or temporally interleaved, and the reclamation technique is strongly dependent on the garbage detection technique.

Given the basic two-phase operation of a garbage collector, many variations are possible. The first phase, distinguishing live objects from garbage, may be done in two ways: reference counting or tracing. Reference counting garbage collectors maintain counts of the number of pointers to each object. The count is used as a local approximation of determining true liveness. Tracing collectors determine liveness more directly, by actually traversing the pointers that the program could traverse to find all of the objects the program might reach. There are many varieties of tracing algorithms: mark-sweep, mark-compact, copying, and non-copying implicit algorithms, etc.

We shall need to distinguish between the garbage collector and the part of the program that does "useful" work. Following Dijkstra's terminology [7], we will call the user program *mutator* in the following discussion.

2.3.1 The Reference Counting Algorithm

In a reference counting algorithm, each cell has an additional field called the reference count. The storage manager must maintain the reference count of each cell equal to the number of pointers to that cell from roots or heap cells.

In the beginning, all cells have a reference count of zero and are placed in a pool of free cells. When a new cell is allocated from the pool, its reference count is set to one. Once a pointer is set to refer to this cell, the value of the cell's counter is increased by one. When a reference to the cell is removed, the counter is decreased by one. If the

reference count drops to zero, it means that there is no remaining pointer to this cell and it can be returned to the list of free cells to be reused.

In terms of the abstract two-phase garbage collection, adjustment and checking of reference counts implement the detection phase, and reclamation occurs when reference counts hit zero. These operations are interleaved with the natural execution of the program.

Advantages of the reference counting algorithm:

- Its memory management overheads are distributed throughout the execution of the mutator.
- It is unlikely to damage the spatial locality of the mutator.
- Its performance does not degrade with heap residency.
- Immediate reuse of cells generates fewer page faults in a virtual memory system, and possibly better cache behavior.

Some disadvantages of this algorithm:

- A simple reference counting algorithm can not reclaim cyclic structures.
- Extra space in each cell is needed to store the reference count, and there are high processing costs to update counters.
- Tight coupling to the mutator makes the collector hard to maintain.

There are some modified algorithms for overcoming or at least ameliorating each of these shortcomings. For example, Weizenbaum's algorithm [6] (non-recursive freeing) removes the uneven delay of simple recursive freeing to smooth the reclamation overhead. A deferred reference counting algorithm [8] is used to reduce the cost of pointer writes. Limited-field reference counting algorithms, such as the one-bit reference

counts algorithm [9], use a smaller count field to save space at the cost of having to handle overflow. In addition, there are several algorithms, such as Bobrow's algorithm, weak-pointer algorithm, and partial mark-sweep algorithm [6], used to deal with the problem of cyclic data structures.

2.3.2 The Mark-Sweep Algorithm

Under this scheme, cells are not reclaimed as soon as they become garbage, but remain unreachable and undetected until all available storage is exhausted. If a request is made for a new cell when all available storage is exhausted, "useful" processing is temporarily suspended and the garbage collector routine begins to work.

Mark-sweep collection is performed in two phases. The first phase is known as marking, distinguishing live objects from garbage. Based on the root set, the objects are traced and marked in some way. Any cell that is left unmarked could not be reached from roots, and hence must be garbage. The second phase is sweeping. It sweeps the heap linearly from bottom to top, returning unmarked cells to the free pool and clearing the mark-bits of active cells in preparation for the next garbage collection cycle. If the garbage collector is successful in reclaiming sufficient memory, the mutator request is satisfied and computation can be resumed. A bit associated with each cell, the mark-bit, is reserved by the garbage collector to determine whether the cell is reachable from the roots.

Comparing to reference counting, mark-sweep has three advantages:

- Cyclic structures are handled quite naturally with no special precautions needed.
- There is no overhead for pointer manipulation.

- The interface between the mutator and the collector is much simpler.

On the other hand, it has its own disadvantages:

- It is a stop-and-start algorithm: computation stops while the garbage collector runs, and the pause may be substantial.
- The complexity of this algorithm is proportional to the size of the entire heap rather than, say, just the number of live cells.
- It tends to fragment memory, scatter cells across the heap and leads to loss of locality.
- It runs become more frequently as the heap occupancy or residency of a program increases, and so the mutator's share of the processor will be reduced.

There are several ways improving the performance of this algorithm:

- An iterative marking algorithm [6] with an auxiliary marking stack can replace the recursive one to speed up the marking algorithm.
- Bitmap marking [10] uses a separate bitmap table to store mark-bits which may avoid wasting space and reduce the frequency of page faults and cache write misses in the marking phase.
- Lazy sweeping, such as the Boehm-Demers-Weiser sweeper [6] or Zorn's lazy sweeper [11], reduces garbage collection pause time by transferring the cost of the sweep phase to allocation.

2.3.3 The Mark-Compact Algorithm

Mark-compact collectors remedy the fragmentation and allocation problems of mark-sweep collectors. In general, compacting collectors have three phases:

1. Traversing and marking the reachable objects;

2. Compacting the graph by relocating cells; and

3. Updating the values of pointers that referred to moved cells.

Mark-compact algorithms can be categorized into three classes according to the relative position in which cells are left after compaction: arbitrary, linearising and sliding. In sliding algorithms, such as Lisp 2 algorithm, Table-based methods and Threaded methods [6], compacting is done by a linear scan through memory, finding live objects and "sliding" them down adjacent to the previous object. In linearising algorithms, such as Fenichel-Yochelson collector [12], cells that originally pointed to one another are moved into adjacent positions. On the other hand, in arbitrary algorithms, such as Two-Finger algorithm [6], cells are moved without regard for their original order, or whether they point to one another.

Because of the several scans over the live objects in compaction phrase, the mark-compact is undoubtedly expensive. However there are several reasons to use it:

- It can reduce the cost of allocation because the free area of the heap is continuous.
- It preserves the initial layout of data in the heap (only for sliding algorithm).
- Long-lived data are unlikely to be moved again once compacted.

2.3.4 The Copying Algorithm

Like the mark-compact algorithm, copying garbage collection does not really "collect" garbage. Rather, it moves all of the live objects into one area, and the rest of the heap is then known to be available because it contains only garbage. Unlike mark-compacting collectors who use a separate marking phase that traverses the live data, copying

collectors integrate the traversal of the data and the copying process, so most objects need only be traversed once.

A very common kind of copying garbage collector is the semi-space collector. It divides the heap equally into two spaces, one containing current data, and the other obsolete data. Copying garbage collection starts by flipping the roles of the two spaces. Then the collector traverses active data structures in the old semi-space (Fromspace) and copies each live cell into the new semi-space (Tospace) when the cell is first visited. After all active cells in Fromspace have been traced, a replica of the active data structures has been created in Tospace and the mutator is restarted. A natural and beneficial side effect of copying garbage collection is that the active data structure is compacted into the bottom of Tospace. The Cheney's non-recursive copying algorithm [13] is one of the most popular algorithms.

Advantages of this algorithm:

- Allocation costs are low because the heap is compacted.
- Work done at each collection is proportional to the amount of live data at the time of garbage collection.

Disadvantages of the copying algorithm:

- Requiring two semi-spaces, it needs double address spaces compared with non-copying collectors.
- Its locality might be worse than mark-sweep because of flipping between two semi-spaces.
- Performance degrades as the residency of a program increases.

So far, we have discussed some basic garbage collection algorithms: reference counting, mark-sweep, mark-compact and copying algorithms. There are others, such as the non-copying implicit algorithm [14], which uses two pointers to link free space and live objects in order to avoid copying objects. In the next section, we give brief comparisons of these basic algorithms.

2.3.5 Comparing the Basic Garbage Collection Algorithms

It is difficult to compare garbage collection algorithms either in principle or in practice. The following comparisons are informal and simplified.

In Table 2.1, we compare the reference counting and the tracing algorithm. The tracing algorithms include mark-sweep, mark compact, and copying algorithms.

Standards	Reference Counting	Tracing
Cyclic data structures	Need special action	No special action
Pause or not	Instant and evenly distributed	Stop and collect
Overhead on pointer update	Considerable	No
Relation between the collector and the mutator	Complex	Simple
Space overhead	Reference counter bits	Mark bits
Affected by high heap residency	No degradation	Degradation

Table 2.1. Comparing Non-tracing Algorithms and Basic Tracing Algorithms

Comparisons among the tracing algorithms (mark-sweep, mark compact, and copying algorithms) are more subtle. We compare them further in two aspects.

1. Space and locality

- Mark-sweep and mark-compact collectors require less address space than semi-space copying collectors.

- Sophisticated mark-sweep collection, using a stack and a mark bitmap, only modifies heap memory at allocation time. On the other hand, copying collectors must write forwarding address into live objects in Fromspace and update pointers in Tospace data. Therefore, mark-sweep may have better locality than copying.
- Sliding compactors preserve the initial layout of data in the heap, while copy collectors do not. So, mark-compact may also have better locality than copying.

2. Time complexity

- The complexity of copying algorithm is proportional to the size of live data. On the other hand, that of simple mark-sweep is proportional to the size of heap. However, when a lazy sweeper is used, sweeping can be done lazily by the allocator, so complexity is not so high. As to mark-compact, the two or three more scans of live data in the compacting phrase make this algorithm more expensive.
- Marking any but very small objects will be less expensive than copying them.
- Copying and mark-compact collector's cost of allocation will be less than that of mark-sweep collector.

From the discussion above, if allocation rates are very high, or the lifetimes of most objects are very short, copying collector should be a better choice.

2.3.6 Generational Garbage Scheme

The generational strategy is to segregate objects by age into two or more regions of the heap called generations. Different generations can then be collected at different frequencies, with the youngest generation being collected frequently and older generation

much less often or possibly not at all. Objects are first allocated in the youngest generation, and promoted into older generations when they survive long enough.

This scheme is based on considerable evidence supporting the weak generational hypothesis – most objects die young [6]. The insight behind the generational garbage collection is that storage reclamation can be made more efficient and less obstructive by concentrating on those objects most likely to be garbage, i.e., young objects.

Several attractive advantages of this strategy:

- By collecting only a part of the heap, pause time can be diminished.
- By avoiding repeated processing objects that remain active, the overall effort of garbage collection measured over the entire program may be reduced.
- By concentrating allocation and collection effort on a smaller region of the heap, paging and cache behavior of both the mutator and the collector can often be improved.

However, there is a big price to pay: the system must be able to distinguish older objects from younger ones. Garbage collection starts by tracing from a known root set. Unfortunately, determining the roots of a generation is more difficult than determining the roots of the entire heap. A generational collector must check whether any pointers to objects in one generation are stored in objects of other generation. Any such pointers must be treated as roots of that generation.

Restricting the size of the young generation will reduce pause time. However, a small generation will increase the rate of promotion, so tenured garbage (objects that would have died in a younger generation if the promotion rate was low) will be copied into an older generation and then die in a less frequently collected generation. We can use

multiple generations to get smaller youngest generation, or use an adaptive tenuring algorithm. Ungar-Jackson's feedback mediation [15] adapts the threshold dynamically to avoid premature promotion, and Barrett-Zorn's dynamic threatening boundary technique [16] can further reduce the amount of tenured garbage.

Generational garbage collection techniques have proven to be very successful, and are now in widespread use. In addition, to reduce the pause time of tracing garbage collection, we can adopt not only a generational scheme but also a technique called incremental technique [6][14].

2.4 Garbage Collection in Prolog

The first Prolog interpreter was designed with no real concern for memory efficiency. Memory allocated during a procedure call was not reclaimed before backtracking. The Edinburgh implementation is the first to benefit from memory management efforts. The memory space is divided into two spaces, the local stack and the global stack (heap). The local stack can be deallocated on the return of procedure calls. Hence, some memory can be reclaimed before backtracking, which is an improvement over previous systems. Despite the improvement, the heap is not garbage collected and it generally grows indefinitely as processing continues.

In an earlier paper written by Bruynooghe [17], he proposed to implement a conservative traversal of all goal statements found in the backtrack stack, which consists of choice points. A choice point holds a goal statement plus a reference to alternate clauses for that goal. Huitouze [18] studied the properties of variables, and presented a new data structure, attributed variable, combined with a memory management machine,

MALI, which encapsulated a garbage collector. Touati and Hama [19] developed a generational copying garbage collector. Their algorithm uses copying when the new generation consists of the top most heap segment. For the older generation they use a mark-compact algorithm. While Touati and Hama still wish to retain properties such as memory recovery on backtracking, Bevemyr and Lindgren [20] took a more radical approach. They tried to ease garbage collection by giving up some backtracking. Theirs is a copying algorithm, and requires the existence of two tag bits for each cell on the heap. Tarau [21] proposed that an ideal memory manager is ecological and he wants the collector to have a "self-purifying" engine that recuperates space not as deliberate "garbage-collection" but as a natural way of life.

In Bekkers et al. [22], they proposed three principles used to determine which run-time objects in logic programming are useful.

1. All accesses to useful objects come from active goal statements. This suggests a marking procedure executing a traversal of goal statements in the backtrack stack.
2. A variable reachable only from terms earlier than its binding renders this binding useless. A technique to reset the variable and discard its trail is called "early reset".
3. Some variables may become irreversibly substituted. The technique to replace an occurrence of a variable with its binding value has been named "variable shunting".

With regard to the implementation of Prolog, there are several typical problems in previous GC algorithms:

Copying GC:

A frequent claim among implementers is that backtracking is incompatible with copying collection. The reason given is that copying GC usually move objects regardless of the heap structure. Hence, instant reclaiming by backtracking becomes impossible.

Mark and Compact GC:

SICStus Prolog uses a mark-compact algorithm to collect garbage. This method preserves chronological order of heap and stack segments as required for backtracking. However, two or three more scans in the compacting phrase make it expensive.

Generational GC:

The two generations are delimited by choice points. As usual, when references are created from an old generation to the new one, these references have to be considered as roots. By chance, a Prolog run-time system records such reference creations in the trail, and the collector simply has to scan the trail to find them. One drawback of this scheme is that the generation delimiter may be left in such a state that old segment is empty (or almost empty). For example, deterministic programs would never be in a position to benefit from the advantage of generation collection. One solution is to create artificial choice points.

In this chapter, we have argued that garbage collection is essential for fully modular programming to allow reusable code and to eliminate a large class of extremely dangerous coding errors. Recent advances in garbage collection technology make automatic storage reclamation affordable for high-performance systems. Even relatively simple garbage collectors' performance is often competitive with conventional explicit storage management. Furthermore, generational techniques reduce the basic costs and disruptiveness of collection by exploiting the empirically observed tendency of objects to die young. Incremental techniques may even make garbage collection relatively attractive for real-time systems.

Chapter 3. The LVM and CGC

3.1. WAM vs. LVM

The Warren Abstract Machine (WAM) [23] is an efficient Prolog execution model consisting of a set of high-level instructions and a memory architecture for handling control and unification. It has been accepted as the standard basis for implementing Prolog for more than ten years.

The WAM simulates the conventional procedure call to control Prolog program execution. It adopts Structure Copying (SC) [24] to represent Prolog terms and defines a set of operations to deal with special cases of the general unification. Parameter passing in a procedure call consists of two phases: the put-phase and the get-phase. During the put-phase the arguments of the caller are loaded into argument registers; and during the get-phase the values in the argument registers are unified with the arguments of the head of the callee.

The LVM is a new Prolog execution model. The difference between the WAM and LVM is that the LVM blends a new method – Program Sharing (PS) [25] with SC to represent and handle logic terms. Generally, a Prolog program is translated into LVM code specifying the control and unification. The control instructions are similar to the WAM's counterpart, however, unification instructions are defined and implemented in a totally different way. All terms are compiled into and handled as executable instructions. Unification is purely pair-wise instruction driven. Objects stored in execution environment are no longer tagged data but rather directly-executable unification instructions or code-segment entries.

Another major difference is that LVM adopts a single stack scheme to manipulate memory, while traditional Prolog implementations are based on stack/heap memory architecture. In the stack/heap architecture, the stack is responsible for dynamic memory requirements related to procedure call and return. The heap takes care of all other dynamical memory needs. For example, WAM stores execution environment and choice-points in the stack and saves all dynamically created data objects in the heap. In LVM, we explore a single stack paradigm for all dynamical memory allocations and embed an efficient garbage collection algorithm, the Chronological Garbage Collection (CGC), to reclaim useless memory cells.

3.2. Term Representation in the LVM

Prolog is a dynamic typing language in the sense that variables may hold dynamically created data objects of any type. Creation and manipulation of dynamic data objects cost both time and space. Therefore, the performance and memory utilization of a Prolog system are greatly influenced by how logic terms are represented.

3.2.1 Structure Copying vs. Structure Sharing

For more than twenty years, two very different methods, Structure Sharing (SS) and Structure Copying (SC), have been used to implement term unification in various Prolog systems.

The fundamental distinction of SS and SC is their way of representing structures. SS represents a structure instance by a two-pointer molecule with one pointer to the structure skeleton and the other pointer to a variable binding environment. On the other

hand, SC makes a concrete copy of a structure in the heap for each instance. SS was used widely in earlier Prolog implementations. However, SC has been accepted as the de facto standard in modern Prolog implementations.

The following is a simple Prolog program. The execution of the query will create two structure instances carried by variables A and B, and unify A and B afterwards. Figure 3.1 [25] illustrates term representation of SC before and after A=B.

```
?- p(A), q(B), A=B.
p(t(X, q(X, Y, a), Y)).
q(t(r(Z), W, f(a,b,c))).
```

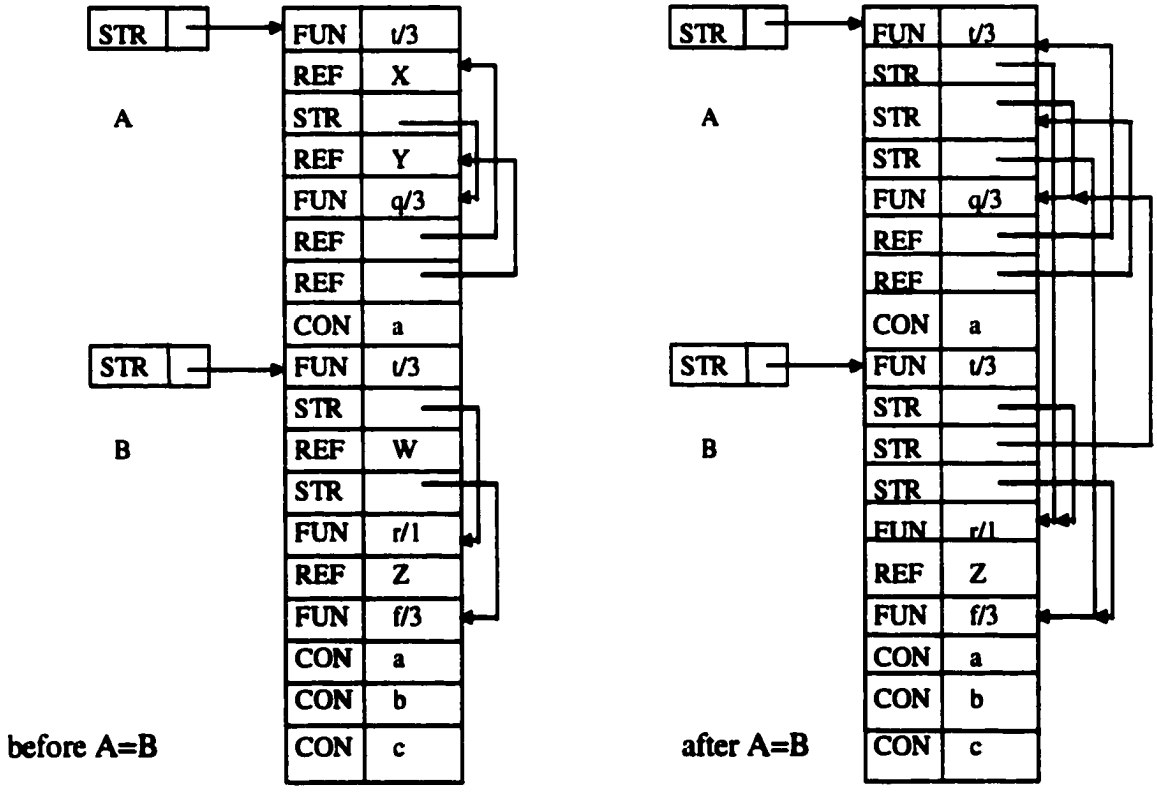


Figure 3.1. Term Representation of Structure Copying

In Figure 3.1, two structure instances are copied into heap as flattened records. Each record starts with a main *functor* followed by an array of cells identifying its

arguments. Multiple occurrences of a shared variable are equated by pointers to a self-referential cell representing a single occurrence. For example, the second occurrence of variable *X* in *q/3* (5th cell) is pointed to the self-referential cell (2nd cell). Nested structure, such *q/3* and *r/1*, are represented by tagged data (*STR*) with pointers to their corresponding records. When the structure is unified with a free variable, SC changes the variable's tag *REF* to *STR*, and adds a pointer to the record. For example, the 2nd cell *REF X* is changed to *STR* with a pointer to *FUN r/1*.

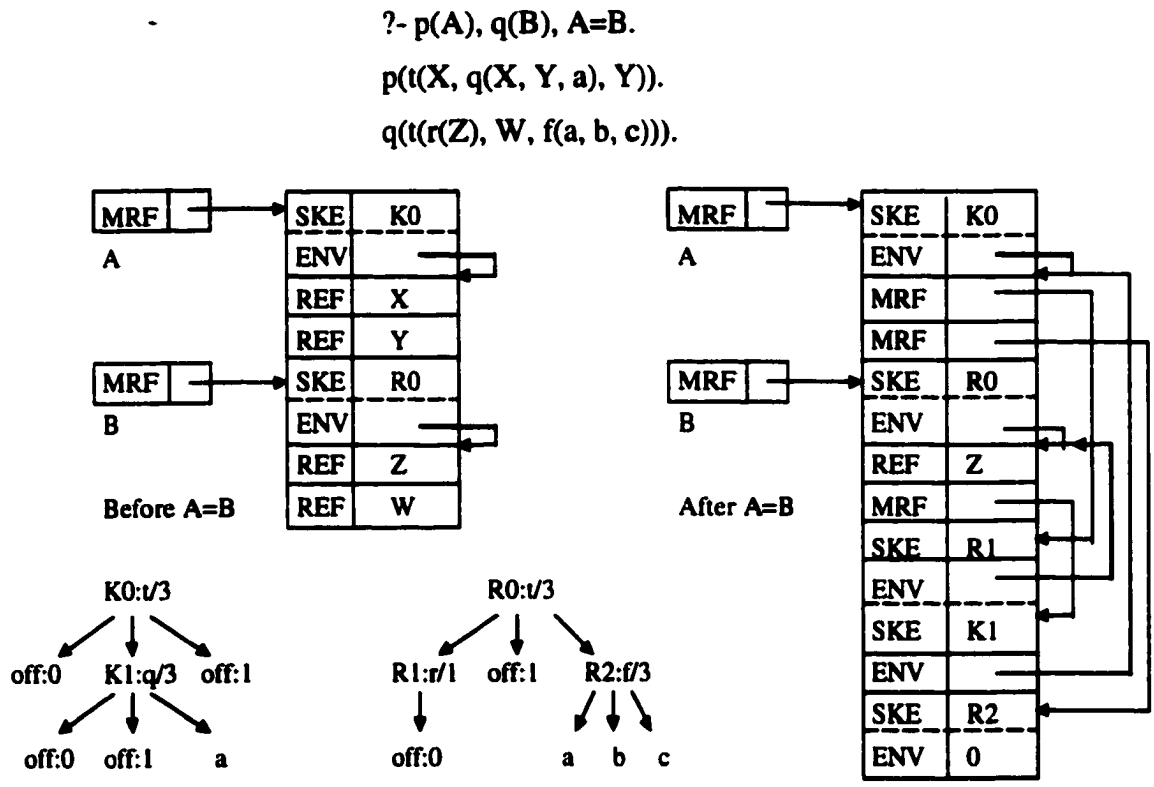


Figure 3.2. Term Representation of Structure Sharing

Figure 3.2 gives the SS term representation adopted by MProlog. A molecule is represented in two successive machine words delimited by a dotted line. One word pointes to the structure skeleton, and the other to a variable binding environment. When a structure becomes the binding of a variable, a molecule may be created on the heap. For

example, variable *X* should be bound with structure *r(Z)*. Then a molecule with skeleton pointer to *R1* and an environment pointer to *Z* is created on the heap. The symbol *off:i* shown in Figure 3.2 indicates the offset of the *ith* variable in its environment. In SS, variable indices in the same environment are calculated against their fixed frame base.

Comparison of SC with SS is not simple, and no quick answer can be given as to which is better. In the SC system, copying a complex structure consumes not only time, but also space because superfluous memory cells must be allocated to constants and pointers to share variables and nested structures. On the other hand, the molecule in SS system consists of two components, which may increase the heap space or result in address accessing limitation. Because SS takes advantage of the fact that different instances of the same term could share a single prototype and differ only in their variable bindings, it is faster to create terms in a SS system. However, since SS does not show nested structures in the heap, it has to create extra heap cells to unify the variables and nested structures. On the other hand, SC does not need extra heap cells when unifying. Therefore, it is slower to unify terms in a SS system than in a SC system.

3.2.2 Program Sharing

Recently, Li [25] proposed a new Prolog term representation method – Program Sharing (PS). The idea of PS originated in SS. It tries to extract static information from a structure during compilation, which static information could be shared by all dynamic instances of the structure, if care is taken to let them have different environments for holding variables. The major difference between them is how PS handles the information held by a molecule in SS. More importantly, the shared static information can be seen as

executable bytecodes for the structure unification in the PS system, but not the only structure skeletons in the SS system.

The reason for using two-cell molecules in SS is that we do need a skeleton pointer and an environment pointer for a dynamic structure instance; such an instance might be carried by a variable to an arbitrary place, and delayed execution of the structure code needs to access variables or nested structure instances which were created in some environment different from the current one. How does PS handle these two pieces of information without using a molecule? A so-called code stub mechanism is introduced to solve this two-pointer problem. When a procedure is called, the heap frame is allocated to hold not only variables but also the code entries of the structures (including nested structures). The heap cell holding a structure code entry is called the code stub.

To compare the PS with SC and SS, we use the same Prolog program as before. Figure 3.3 shows the term representation and executable code in the PS system.

As shown in Figure 3.3, when procedure *p/1* is called by goal *p(A)*, four stack cells are allocated as an integral frame to hold stack variables and stubs occurred in *p/1*. At the same time, the address of stub *PC0* plus an opcode *DSI* is assigned to variable *A*. The first two cells are initialized by stubs (*PC0* and *PC1*), and the next two cells are unbound variables (*X* and *Y*). A stub serves two purposes: its address is the environment base for accessing nested components; and its content, which is *PC0* in this case, gives the structure code entry. A similar action occurs for the call of *q(B)*. The last goal of *A=B* thus involves four basic pair-wise operations: a functor matching and three assignments. The assignments are:

- an opcode *DSI* with pointer to stub *QC1* is assigned to *X*;

- an opcode *DSI* with pointer to stub *PC1* is assigned to *Y*; and
- an opcode *DCI* with code entry *QC2* to *Y*.

The symbol *Vi* in Figure 3.3 is variable offset too, but not the exact same as *off:i* in SS. In SS, *off:0* refers to the same variable *X* in both skeletons of *K0* and *K1* in Figure 3.2, while in Figure3.3, the *V2* in *PC0* segment refers to *X* and the *V2* in *PC1* segment refers to *Y*. This is because in PS the offset of variables and stubs is computed according to their own segment bases.

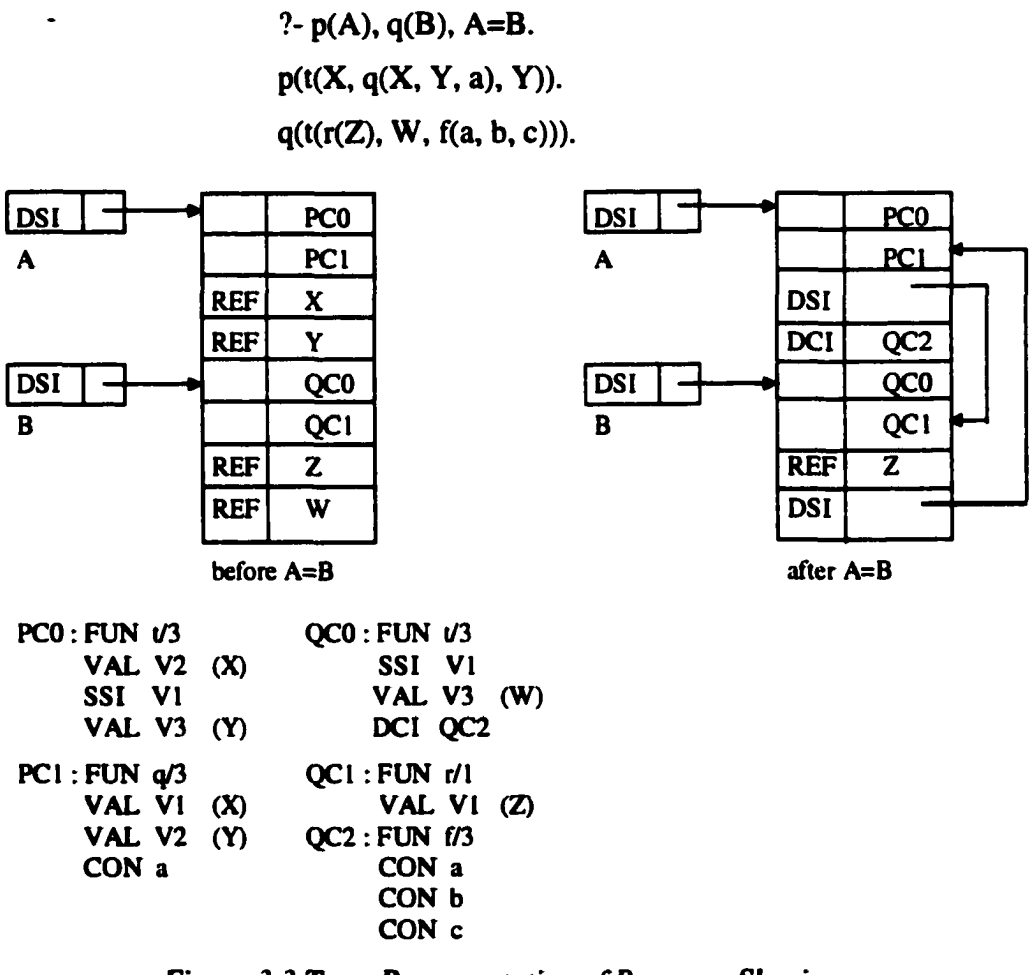


Figure 3.3 Term Representation of Program Sharing

Because the stubs of all structures are allocated in the stack, we do not need extra cells during unification. In addition, PS does not flatten whole terms in the stack as SC does, so its stack consumption is not so high as in SC. Therefore, when handling complex shared structures, PS has better space performance than SC, and better time performance than SS.

Although PS can efficiently handle structures, it is not convenient for manipulating lists. This is the first reason we combine SC and PS to handle lists in the LVM system. When there are shared variables in the lists, the initializations equate them. The second reason that we combine SC with PS in LVM comes from considerations of garbage collection. It is impossible to compact the heap properly in a sharing-based system such as SS and PS, because a dynamic shared instance is accessed by offsets from a certain heap base, which will be damaged by compacting.

In order to take advantage of SC and solve the above mentioned garbage collection problem, LVM is designed as follows:

- It uses a single stack to replace the traditional environment stack and heap;
- It adopts a hybrid of PS and SC to represent Prolog terms; and
- It cooperates with CGC that carries out "PS to SC" transformation during garbage collection.

Meanwhile, two supplementary stacks, the trail and the pushdown list, remain unchanged.

3.2.3 Shared Instance and Copied Instance

The LVM defines two types of structure instance: shared instance and copied instance.

A *shared instance* needs to consult a changing environment. It carries an environment pointer to a stub of code entry. By accessing the stub we have the entry to the code segment that defines the necessary instructions for structure unification, and by the stub address we get the environment which will be consulted for visiting nested components. A shared instance may be a static one that is represented in the code area by *SSI* bycode, or a dynamic one created in the stack with *DSI* tag.

A *copied instance* does not involve offset-addressable variables. If there are shared variables in a copied instance, these variables are equated in the same way as in SC. A copied instance represents a concrete copy of a structure dynamically created on the LVM stack.

During compilation, lists and static ground structures are processed as copied instances. Since a static ground structure is completely environment-independent, a single copy of its LVM code can be freely accessed during execution. Other compound terms will be handled as shared structures. If the compiler incorporates mode analysis, further optimizations can be applied.

3.3 Memory Organization

LVM divides the main memory into segments of various sizes. The memory architecture is similar to the WAM memory configuration. Figure 3.4 gives the layout of

memory architecture. A major difference from the WAM memory layout is that LVM uses one stack for holding dynamic objects as well as control information.

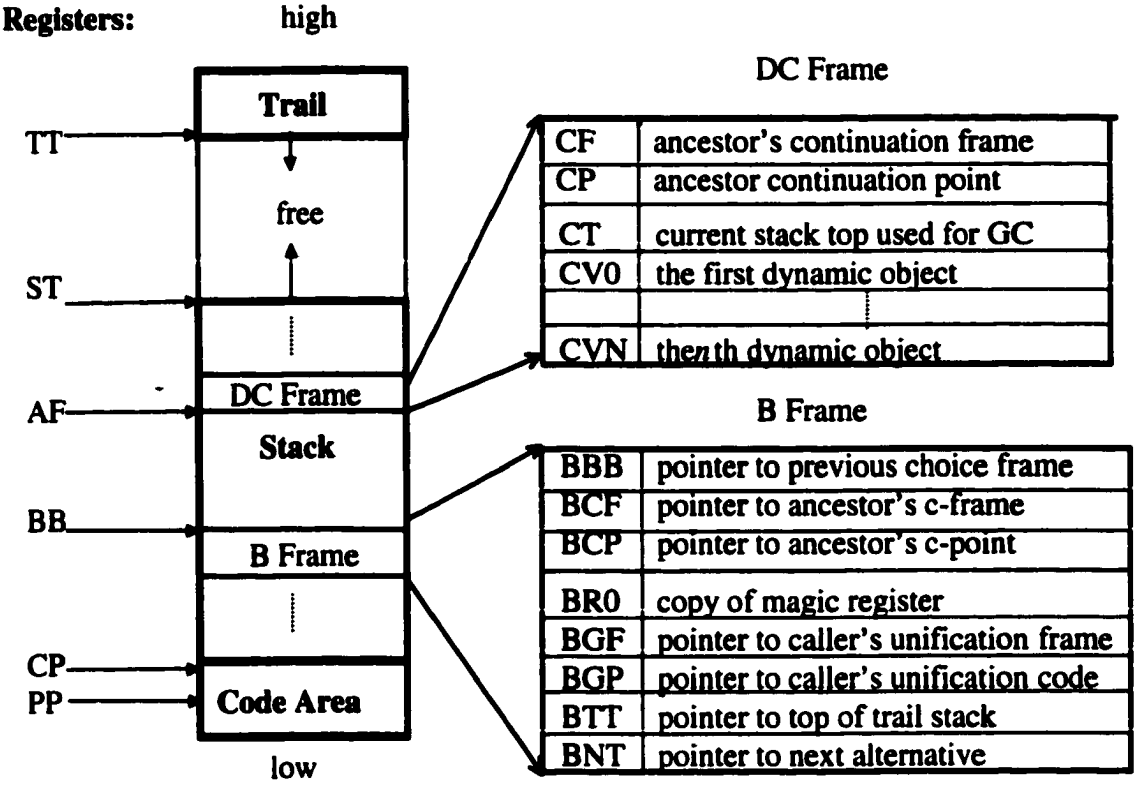


Figure 3.4. The LVM Memory Architecture

LVM does not classify local variable and global variables. All variables, code stubs, and copied instances are called *dynamic objects*. For a given clause, the total number of its dynamic objects is completely determined during compilation. When a procedure is called, an integral stack frame is allocated for the matching clause. Procedure invocation and backtracking are implemented using different chains of information frames allocated in the stack. There are three types of stack frames: *D-frame, DC-frame and B-frame*. A *D-frame* is used for facts and chain-call clauses. It has only cells for dynamic objects without control information. A *DC-frame* is allocated when a matching clause has more than one goal in its body. It consists of three cells for control

information and cells for dynamic objects, such as variables, copied instances and stubs. Control information includes:

- continuation point *CP*,
- continuation frame pointer *CF*, and
- current stack top *CT*.

A B-frame is allocated when a choice point is met. It contains a fixed number of cells representing the execution state.

Stack allocation is simple and straightforward, and it does not even check stack overflow. The whole stack is divided by a stack top register *ST*: the space above *ST* is the free area and below is the occupied area. The LVM does not deallocate stack frames, nor does it do last call optimization. This paradigm assumes that we have an infinite memory to use. This is clearly beneficial for the LVM implementation. There is no need to verify binding direction of two variables. There are no unsafe variables. The trail operation can also be simplified by comparing the variable address with the latest choice pointer only (WAM requires one more condition to discard local variables). Furthermore, passing arguments from a caller to a callee can be done from stack to stack and therefore eliminates the bottleneck of soft-registers introduced by WAM.

3.4 Chronological Garbage Collection

Apparently, the assumption of infinite memory is not feasible in the real world. Garbage collection is mandatory for LVM to be a practical system.

Memory can never be unlimited. Because application programs have grown enormously in size and complexity, especially programs written in Prolog or Lisp that

typically manipulate large data structures with complex inter-dependencies, automatic storage reclamation is essential for practical implementations. Moreover, the single stack paradigm used in LVM makes the memory consumption faster. Therefore, a high-efficiency, relatively frequently-run garbage collector is the key to the LVM model.

Based on the study of garbage collection algorithms discussed in Chapter 2, we developed a new algorithm: Chronological Garbage Collection. It is a tracing collector, combining the advantages of several garbage collection algorithms. Like copying collector, CGC traverses from a small set of roots and copies live objects onto a free space. From mark-compact, CGC borrows the idea that at the end of collection, the stack will be compacted into two continuous areas: one for active objects and the other for free cells. Based on the weak generational hypothesis, CGC introduces a concept of chronological generation – a dynamical and "natural" way to divide generations. Finally, CGC controls the frequency of collector invocations by a factor of cache size, and collects garbage incrementally with a trivial pause time.

3.4.1 C-line and C-reachable

In LVM, a single stack is used to contain the invocation frames. New frames being allocated are piled on top of what already exist on the stack. Frame allocation follows the chronological order of procedure calls. When a procedure returns, it leaves its frame undeallocated. A possible snapshot of the stack layout is given in Figure 3.5, where *AF* indicates the current active *DC-frame and BB* points to the latest *B-frame*.

The current active *DC-frame* is called *c-frame*, in which *CT* contains the base-address of stack top saved when this frame was allocated. This base-address is called *c-*

line. However, another possible stack layout is with the c-frame below the latest choice frame. If this occurs, the c-line is determined by *BB* (then we do not need to worry about any effect on backtracking and trailing).

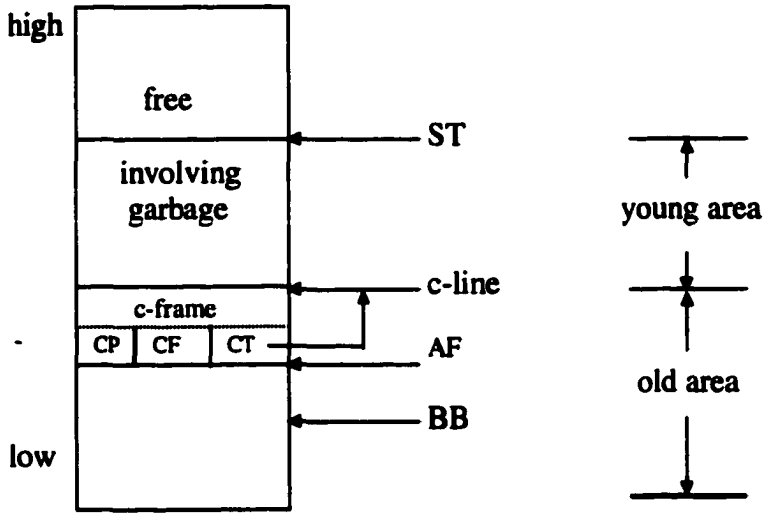


Figure 3.5 A Possible Snapshot of the Stack Layout

The most important address in both layouts is the *c-line*. It is a divider of two generations. The area above the *c-line* is the young generation, where stack frames were used by terminated procedures. The area below is the old generation, where stack frames are either still active or being frozen by choice points. A *c-variable* is a variable that resides in the *c-frame*. A location that can be reached by accessing a c-variable (through dereferencing and recursively visiting every argument if the variable is bound to a compound term) is called *c-reachable*. An old-young reference is a reference that resides at a location in the old generation and points to a location in the young generation.

Based on the above definitions and discussion, we introduce two properties:

- Property 1: Any old-young reference must be *c-reachable*.

The proof directly comes from the nature of the LVM single stack architecture and the way of implementing Prolog procedure calls. First of all, there are no globally scoped

variables in Prolog, i.e., all variables are local to their clauses. Secondly, invocation frames are strictly allocated in order of procedure calls. Third, execution of a procedure finds all its needs from the associated frame without any further memory requirement. With this set up, the only way to pass an old reference to a young procedure is through the parameter passing mechanism, namely, variables in the *c-frame*. In other words, the set of *c-variables* is the only bridge connecting the old generation to the young generation.

- Property 2: Space above the *c-line* (young area) can be safely reused if all *c-reachable* objects have been collected.

From Property 1, we know that any object above the *c-line* is not *c-reachable* is garbage, because there is no legal sequence of program actions that would allow an old reference to reach that object. Thus, if we collect concrete copies by traversing all *c-variables* and store these copies in a temporary free area, then the young area can be safely reused.

Now, let us take an overview of the CGC algorithm. At some stage of execution of a mutator, the collector is invoked. Two dynamic generations are divided by the *c-line*. The collection algorithm consists of two phases. The collection phase traverses from an initial root set (the set of *c-variables*), and creates copies of all *c-reachable* objects onto a free space (the space above ST). During collection, new roots might be added to the root set. This phase stops when the root set becomes empty. Then the compact phase will move copied objects back to the young area and the rest of the space in young area will be returned to the free pool.

3.4.2 The 'cgc' Instruction and the Root Set

The idea of CGC looks quite simple. However, several questions remain to be answered. When and where do we invoke CGC? How do we specify the initial root set? Do we actually need to collect all *c-reachable* objects? To answer these questions, let us first discuss some compilation issues and a special LVM instruction *cgc*.

Most garbage collections require cooperation from the compiler. For example, object formats must be recognizable by the GC algorithm. Since CGC is tightly coupled to the LVM execution engine, two jobs must be carried out by the LVM compiler: inserting *cgc* instructions into proper code positions and generating the initial root set associated with each *cgc* instruction. To invoke CGC, there must exist some garbage. Clearly, it is impossible to do quantitative analysis of garbage during compilation. However, it is possible to have a qualitative estimation. By examining different types of clause definitions, it is easy to find that only a call to a deterministic, recursively defined procedure may generate a linear (or higher degree) amount of garbage, i.e., invocation frames accumulated by executing that procedure.

Hence, for a rule of the form p:-g_1, g_2, ... g_n, the LVM compiler inserts *cgc* instructions by the following strategies:

Let flag = false. Scan the goal list g_1, ...g_n.

- If g_i is the last goal and flag = = true, inserts a cgc instruction before g_i;
- If g_i matches a nondeterministic rule and flag = = true, inserts a cgc instruction before g_i, and flag = false;
- If g_i matches a deterministic, recursively defined rule, flag = true;

The reason for placing a *cgc* in front of a nondeterministic goal is straightforward: garbage should be collected before the stack is frozen by a choice point. Taking away nondeterministic cases, there exists a resemblance between *cgc* instructions and WAM deallocation instructions in that they are placed before the last call. However, if we invoke the collector every time a *cgc* instruction is executed, the cost of running such an aggressive collector may be significant. Certainly, increasing the size of the region being collected can always reduce collection frequency. The most important measurements to determine the invocation of CGC are generation-gap and cache-limit. Generation-gap is defined as the distance from the youngest generation to some old generation. In our current implementation, (ST – AF–>CF) is used as an approximate estimation. Cache-limit is a machine-dependent constant. From our experiments, half to 2/3 of the size of data cache would be a proper range to select. These measurements serve two purposes. On one hand, we want to control collection frequency so that the collector is not invoked unless there is a reasonable amount of accumulated garbage. On the other hand, we want to collect useful objects more frequently than ordinary generational copying collectors do so that most working objects are kept in the cache. As it turns out, only a simple comparison is required by the *cgc* operation: if the generation-gap is greater than the cache-limit, it triggers the collector, otherwise, it does nothing.

Next, the LVM compiler ought to generate the initial root sets. An initial root set defines a list of offsets of variables that should be collected. The safest way to specify the initial root set is to include all variables occurred in a rule that contains *cgc* operations. If so, some of the collection efforts might be wasted because not all the collected objects are useful in the remaining computation in most cases. Thus, the LVM compiler attaches an

initial root set to each *cgc* instruction. It includes all variables in the clause head and all initialized/instantiated variables in the remaining goals. Clearly optimization can be applied to further reduce the size of initial root set. For example, incorporated with mode analysis, head variables used as pure selectors could be excluded. Again after an arithmetic operation "N1 is N-1", N1 commonly occurs in the last goal. In this case, N1 need not to be collected.

3.4.3 The CGC Algorithm

Here is an outline of the CGC algorithm.

1. Create the root stack; push all initial roots onto the stack.
2. For each root *r*, we search for binding *r'*:
   - if *r'* is atomic, *r'*<– *r* and continue, where <– is assignment operation;
   - if *r'* is an unbound (self-referential),

     in old area, continue;

     in young area, create a copy.
   - if *r'* is a copied instance:

     in code area (this instance must be statically ground), continue;

     in old area, scan the instance and collect new roots;

     in young area, copy the instance.
   - if *r'* is a shared instance:

     in old area, scan the instance and collect new roots;

     in young area, copy the instance.
3. Re-address the old roots.

4. Move the useful objects back to the space above *c-line* and change *ST*.

Figure 3.6 shows the memory layout during garbage collection. The root trail is used to re-address the old roots. The shade area is the initial target area of copying.

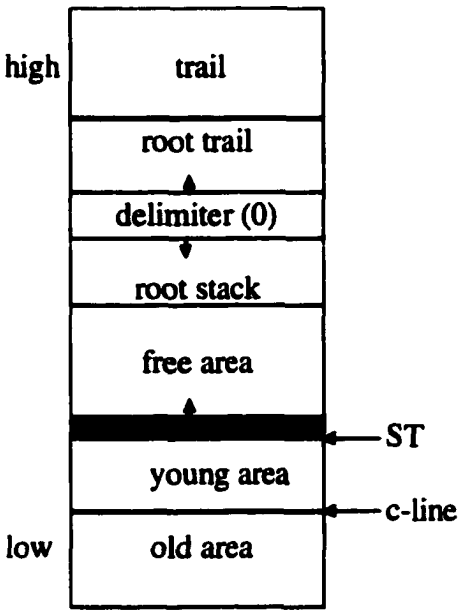


Figure 3.6 Memory Layout during Garbage Collection

CGC uses a modified version of the Cheney Scan [13] to implement the transitive closure algorithm. During collection, all shared instances will be transformed to copied instances. As soon as a young instance has been copied, a forward pointer is set which will prevent creation of duplicate copies. CGC does not need early reset because the LVM compiler [27] has already taken uninitialized variables from the initial root set. Furthermore, CGC implements a cheap variable shunting by simply dropping all intermediate references above the *c-line*.

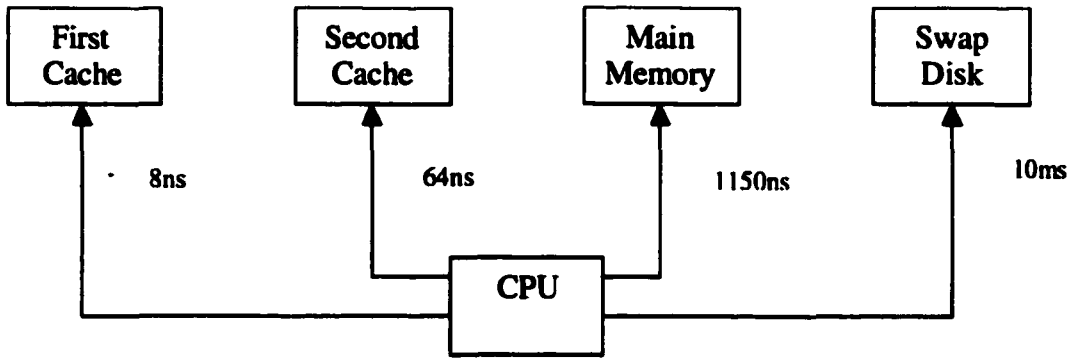
Chapter 4. Cache Architectures

4.1 The principle of caches

In the past decade, peek processor speeds and memory sizes have increased by nearly three orders of magnitude. However, a trend "portends more difficulty in archieving much higher application performance in the coming years – the disparity between speed increases in processors (60% per year) and in DRAM memories (7% per year). This trend, coupled with physically-distributed memory architectures, is leading to very nonuniform memory access time, with latencies ranging from a couple of processor cycles for data in cache to hundreds of thousands of cycles." [28]

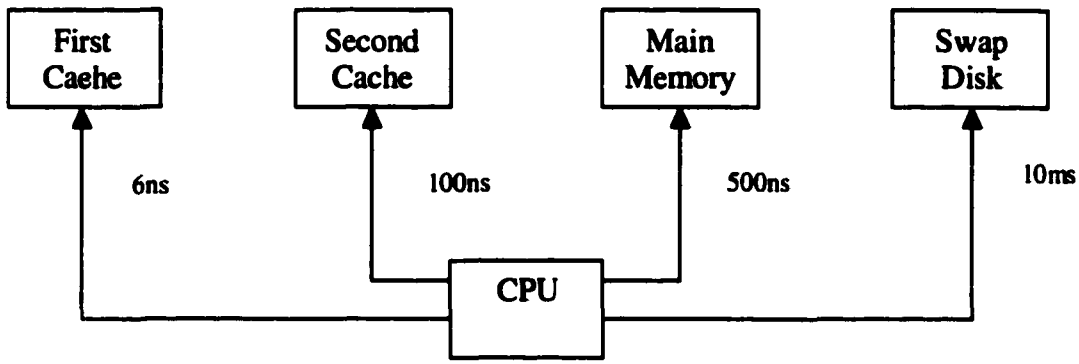
Although faster memory chips are available, it is not economic to use only them for main memory. Instead, inserting a small cache of fast SRAM memory between the CPU and the main memory (DRAM), to reduce the average speed of memory access, is more acceptable. This idea is similar to virtual memory in that an active portion of low-speed memory is stored in duplicate in a higher-speed cache memory.

The cache contains a subset of the words in the address space of a program, which is defined as the set of memory words that can be addressed by a program. Generally, the memory hierarchy of a computer system is organized so that the higher levels contain a subset of the words of the lower levels. The cache is the highest level of the memory hierarchy, closest to the CPU. The virtual memory (or swap disk) is the lowest level, while the main memory is between the cache and the virtual memory. Figure 4.1 shows typical cache-memory architectures – MIPS R4400 and HP NetServer 5/166. It shows ratios of access-time for various levels of memory hierarchy.



1 : 8 :144 : 1,250,000

MIPS R4400



1 : 16 : 83 :1,670,000

HP NetServer 5/166

Figure 4.1 Typical Cache-Memory Architecture

When the CPU generates a memory request, it is first presented to the cache. If the cache cannot respond, the request is passed on to the lower level memory. Caches are effective in improving performance only if programs have good locality properties, that is, either the same locations are used many times (temporal locality) or adjacent locations are used within a short time interval (spatial locality).

If the mutator accesses a memory block that is held in the cache, *i.e.*, *a cache hit* occurs, the datum is immediately available. If not, *i.e.*, a *cache miss* occurs and the processor may have to stall for several clock cycles until the block is retrieved from lower-level memory. The ratio of missed references to the total memory references is

called *cache miss rate*. The time to fetch a word from the cache is called *hit time*. The cost of cache misses, *cache penalty*, is architecture-dependent, varying also between read misses and write misses.

With cache penalty becoming higher, caches are increasingly important in implementing programming languages and designing garbage collection algorithms. So the cache performance of the Chronological Garbage Collection and its improvement (or degradation) of the cache performance of mutators are important factors in evaluating this algorithm.

In our experiments, benchmarks show that this algorithm not only safely collects garbage, but also improves the program's cache performance by more than enough to pay its own overhead. In order to determine the improvement of cache performance and find the best configuration for this algorithm, we developed an emulator to do the trace-driven cache simulation.

4.2 Cache Architectures

A cache consists of an array of fixed-size blocks (or lines) that can keep the contents of one memory block. An auxiliary array of tags contains a directory of memory blocks stored in each cache block. Apart from the tag, the directory contains control bits that keep status information on each cache block. Valid bit indicates whether the block contains valid data, and dirty bit indicates whether its contents have been modified. Each block can be further divided into several sub-blocks. A sub-block is the smallest part of a cache with which a valid bit is associated.

The architecture of *n-way* set associative cache without sub-blocks is shown in Figure 4.2. There are *s* sets in the cache and *n* blocks are grouped into each set. A cache is called *n-way set associative* (or *n-way* associative) if each set contains *n* blocks. *A direct-mapped* cache is a particular case of an *n-way* associative cache where *n=1*, while a *fully associative* cache has only one set containing all the cache blocks. In a *fully associative* cache, a memory block can be placed in any block in the cache, and all entries in the tag array must be searched in parallel to find where the desired block is. In a *direct-mapped* cache, a memory block can be found in exactly one cache block. Some mathematical functions of the memory block address (usually the modulus number of cache blocks) are used to compute the entry that can contain a block. In short, a memory block can be located in exactly one set, but anywhere within a set depending on the replacement policy, which we will discuss later.

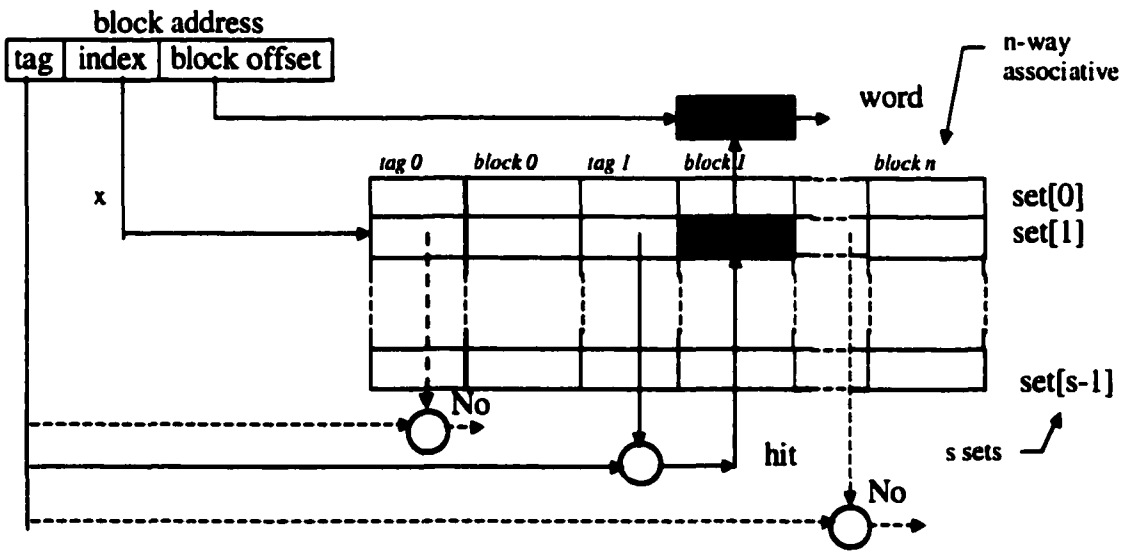


Figure 4.2. Architecture of N-way Associative Cache

Figure 4.2 shows searching method for a memory reference as well. A memory address may be divided into three parts, the tag, the index, and the block offset. The high-order bits are compared with the cache block's tag to ensure that it does indeed store the

contents of the memory block referenced. The index is used to select the set. Note that this is a many-to-one mapping: different blocks in main memory will map to the same set in the cache. The block offset selects a word within a block.

When the mutator issues a memory read reference, the memory location must be mapped to a set according to its index segment. Then the tag array and valid bits in this set are searched and checked to find whether the block that contains the address is present in the set. If the block is in the set, the word is selected from this cache block by the offset segment. Otherwise, the missing block is copied from main memory to the cache. The ways by which write references are handled depends on the write policies, which will be discussed later.

4.3 Important Parameters in Cache Design

The goal of cache design is to balance cost and performance. Cache design involves selecting a number of different parameters, such as cache size, block size, and set-associativity. Usually, there are trade-offs to be made in these selections.

- ***Split/unified caches***

  A cache is unified if the same cache is used for both instructions and data. Having separate caches for instructions and data allows instruction fetching, data reads and writes to be processed in parallel. In our study, we only discuss split caches and concentrate on data caches.

- ***Cache size***

  The size of the cache varies between implementations. Typical cache sizes range from a few kilobytes to a few megabytes. Large caches have lower miss rates, but may

have higher access time. In general, small on-chip caches are common on current processors, while larger caches are common for second-level or single-level off-chip caches.

- ***Block size***

  Block sizes typically range between 4 and 256 bytes. Large blocks may make better use of spatial locality and thus reduce miss rates, because it is more likely that the following references will address the same block. However, if the block size becomes too large in comparison with overall cache size, and thus the number of blocks in the cache becomes too small, cache miss rates may again rise. Large blocks can also have higher miss penalties, because there are more words to be transferred from memory to cache. Therefore, there is a performance trade-off to be made between reducing miss ratios by increasing block size, and increasing penalty when a miss does occur.

- ***Associativity***

  Most caches today are direct-mapped, *i.e.*, each block of main memory is mapped to a single position in the caches. Although direct-mapped caches are simpler to build and faster to search, they may be more prone to conflicts as frequently used blocks of memory map to the same line in the cache.

  Usually, increasing associativity tends to decrease miss rates, but the need to compare many tags in parallel can increase the hit time and parallel hardware, and thus cause a negative impact on performance and cost. Therefore, direct-mapped caches and low degree (from two- to eight-way) associative caches are the most common ones.

- ***Fetch size***

The fetch size determines how many blocks to fetch on a miss. Usually only one block is fetched. Some policies may fetch more than one block or only part of a block, and have different fetch sizes for read and write misses. When less than one full block can be fetched, the fetched unit is called a sub-block, and there is usually one valid (and possibly a dirty) bit associated with each sub-block.

- ***Replacement policy***

For set-associative caches, a memory block can be located anywhere within a set if without further limit. The replacement policy determines which block in a set is replaced when a new block must be fetched into the cache. The least recently used (LRU) policy usually approximates optimal replacement and is the most common replacement policy used in set-associative caches. Other possible replacement policies are random and FIFO.

- ***Write buffers***

A write buffer can store a word or a dirty block temporarily. Writes that miss the cache can be sent to a write buffer and do not need to cause the processor to stall. The processor must stall if the buffer is full. Usually the depth of the write buffer, which is the number of blocks it can hold, varies between four and eight.

It can happen that the write buffer contains the most recent copy of a block, and a word in that block is read, causing a cache miss. Then there are two alternatives: flush the buffer back to the cache before the block can be read, or fetch the block from the buffer directly. The first choice may increase the miss penalty, while the second one may increase the complexity of the buffer hardware.

4.4 Write Strategy

Write issues are in many ways more complicated than read issues, because writes require additional work beyond that for a cache hit, *e.g.*, writing the data back to the memory system. Write strategies can be classed as: write hit policies and write miss policies.

4.4.1 Write Hit Policies

When a write hit occurs, there are two policies to be chosen: *write-through* and *write-back.*

- In a write-through cache, the data are written to the block both in the cache and in the lower level (either a further level of cache or the main memory). A write buffer can be used in a write-through cache to improve its performance by avoiding stalling the CPU.
- In a write-back cache, data are written to the cache, and they eventually go to the next level when removed from the cache because of the replacement caused by a cache miss. On a miss it is not necessary to write a block to the next level if it has not been altered. So, a dirty bit associated with each cache block indicates whether the block has been modified or not.

Write-back caches reduce the write traffic since multiple writes to a single block require only the last write to be transferred to the lower level, but might cause a delay in the fetch of a new block. On the other hand, for a write-through cache, misses do not

cause a block to be displaced so no delay occurs. Note that read misses do not cause data to be written back to the next level when the block is replaced in the cache.

Both high performance write-back and write-through caches require some additional support hardware and complexity: write-through caches require a write buffer, while write-back caches require a dirty bit on every cache block. Since the reduction in write traffic provided by a write-back cache increases as its size increases, for very large on-chip caches write-back caches become more attractive compared to write-through caches with write buffer.

4.4.2 Write Miss Policies

Unlike read misses, where the usual action is to fetch the block containing the addressed word, there are many alternative actions on write misses. For example, it is possible to fetch a block, just as on a read miss, or to bypass the cache and place the word directly in memory.

We study two important write miss policies: *write-allocate/no-write-allocate* and *fetch-on-write/no-fetch-on-write*.

- **Write-allocate vs. no-write-allocate**

Writes that miss in the cache may or may not have a block allocated there. On a write-allocate cache, a block in the cache is allocated and placed for the word being written, and the word is written to the next level, i.e., the main memory, as well. On the other hand, on a no-write-allocate (write-around) cache, the word is written only to the main memory and no block is allocated in the cache. So, for a no-write-allocate cache,

when reads occur to recently written data, they must wait for the data to be fetched from the main memory.

- **Fetch-on-write vs. no-fetch-on-write**

These two choices determine whether writes that miss in the cache fetch the block being written. If the block size is one word, there is no need to fetch, because the word is going to be overwritten anyway. However, if the block size is greater than one word, fetch-on-write and no-fetch-on-write will make difference. If the decision is no-fetch-on-write, then the words remaining in the block must be invalidated. If the decision is the fetch-on-write, the block containing the word being written should be fetched from the main memory first. Then all the words in the block will be valid after write.

4.4.3 Fetch-on-write, Write-validate and Write-around Caches

There are four kinds of caches by meaningful combinations of these write policies. In our work, three of them are considered*: fetch-on-write, write-validate* and *write-around caches.*

- **Fetch-on-write:** A cache is fetch-on-write if it uses write-allocate and fetch-on-write policies.
- **Write-validate:** A cache is write-validate if it uses write-allocate and no-fetch-on-write policies.
- **Write-around:** A cache is write-around if it uses no-write-allocate and no-fetch-on-write policies.

Fetch-on-write and write-validate caches can be combined with either write-through or write-back policy. However, write-around caches can only be combined meaningfully with write-through.

Fetch-on-write caches must fetch a block from the memory before writing it, and thus may cause a pipeline stall. Write-validate caches do not cause a stall, since they do not fetch the block at all. Like write-validate cache, write-around caches fetch nothing from memory, but must wait for data to be fetched back when reads occur to recently written data. To avoid processor stalls, write-around caches usually write the word into a write buffer. Unless the buffer is full, it should not cause any delay. Therefore, write-validate and write-around caches eliminate write misses, because their penalties are eliminated.

However, write-validate and write-around caches may increase read misses. On write-validate caches, an extra read miss occurs only if an invalidated word in the block not fetched on the write miss is read before being written. On write-around caches, an extra read miss occurs if word (the word written and all the other invalidated words) in the block that was not fetched on the write miss is read before being written. Write-around caches tend to add more read misses than write-validate caches, because the word written is more likely to be read again soon than the other words in the block.

In general, write-validate caches perform better than write-around caches, and write-around caches perform better than fetch-on-write caches.

4.5 Cache Architecture of Current Machines

The following table [6] shows the common cache organizations of some popular architectures:

Architecture	Split cache ?	Write hit policy	Write miss policy	Asso-ciativity	Block size (words)	Cache size (kilobytes)
DEC DS3100	yes	through	allocate	1	4	64
DEC DS5000/200	yes	through	allocate	1	16	64
DEC DS3000/500	yes		no-alloc/alloc	1	32	8/512
DEC Alpha21164	yes/no		no-allocate	1/3	32/64	8/96
MIPS R4400	yes	Back		1	16/32	16
MIPS R5000	yes	through/back		2	32	32
HP 9000	yes	back	allocate	1	32	64-2K
SPARCStation 2	no	through	no-allocate	1	32	64
UltraSPARC	yes	through	no-allocate	1	32	16
PowerPC 604	yes	back		4	32	16
PowerPC 620	yes	through/back		8	64	32
Intel Pentinum Pro	yes/no	back		2/4	32	8/256
Intel Pentinum	yes	back		2	32	8
IBM RS6000	yes		allocate	2/4	32	8/64

Table 4.1. Common Cache Organizations

Chapter 5. Simulators and Simulation Algorithms

In the last chapter, we discussed cache architecture and parameters at the conceptual level. In this chapter, we present values of these parameters. Then, we treat simulators and simulation algorithms in greater detail.

5.1 Cache Parameters

To cover typical cache implementations, large ranges of cache parameters are considered.

First of all, not only direct-mapped but also set associative caches are simulated. Direct-mapped caches are simplest to implement, and have faster access times than other types. These are most common in current high-performance computers. Therefore, they are the main subjects considered. Since set-associative caches tend to reduce miss rates by reducing conflict misses, they will be studied as well. However, as shown in Table 4.1, only the lower associativities are implemented in current computer systems. So, only associativities of two, four and eight will be covered by our simulations. In the next chapter, we will see that higher associativities (greater than four) would not reduce cache misses much.

A wide range of cache sizes is studied, from 8KB to 1MB. This covers typical sizes for single-level off-chip caches (32KB – 64KB) and for second- or third-level caches in multi-level systems (1MB) as currently used.

The cache block size ranges from 16 bytes to 256 bytes by powers of two. Main memory will be discussed in terms of memory blocks, assumed to be the same size as the cache blocks. The fetch size may be equal to or less than the block size.

Although multi-level caches are becoming common, only single level caching is considered in our study. The results reported here are expected to extend to two- and even three-level caches. Since accurate analysis of multi-level cache performance requires a more sophisticated memory system simulator, it is left for later.

Three kinds of caches are discussed according to the different write miss policies:

- fetch-on-write caches,
- write-validate caches and
- write-around caches.

Fetch-on-write caches are the most common ones. Write-validate caches can yield significant performance improvement, while the performance of write-around caches lies between those of fetch-on-write and write-validate caches.

The write-hit policies, write-through and write-back, are discussed briefly. Fetch-on-write and write-validate caches can be used with either write through or write back policies, while write-around can only be used meaningfully with write through.

Here only data cache performance is considered. Instruction cache performance is left for later.

5.2 Cache Simulation

Studies of cache performance traditionally use computer simulation, analytical models, or a combination of both. Our method of examining the cache performance of CGC algorithm is trace-driven simulation.

Trace-driven simulation uses one or more address traces and a cache simulator. A trace is the log of a dynamic series of memory references, recorded during the execution

of a program [29]. The information recorded for each reference may include its address, its type such as instruction fetch, data read or data write, its length, and other information. A trace generally contains extremely lengthy address references sequences. A simulator is a program which accepts a trace and parameters describing one or more caches, then simulates the behavior of those caches in response to the trace, and computes performance metrics (such as miss ratio) for each cache.

There are several ways to obtain traces, the fastest being through special hardware attached to an operational machine [30]. The special hardware monitors memory requests and logs each individual reference on tape or disk. Obviously, this method is expensive. So far, the most popular method for generating a trace for studying cache performance is the machine simulator. It is a program that simulates the instruction execution of a computer under study. The input of the simulator is a typical workload. As each instruction is executed, the simulator writes a sequence of address references generated during the simulation to an external file. In our study, the machine to be analyzed is the LVM emulator embedded with the CGC algorithm, which is a "virtual machine" implemented completely in software. Thus, in our case, the machine simulator is almost ready. What we need to do is to modify the emulator to record the references.

To avoid occupying a large amount of disk space, our cache simulations are done "on the fly", which means that there is no explicit generation of address traces. Instead, calls to the cache simulator are inserted directly into the source code of the LVM emulator. The address and type of each reference are carried to the cache simulator as parameters of the call. Here we have mentioned two simulators. One is the cache

simulator, which simulates the behavior of caches and evaluates metrics. The other is the machine simulator, which is a modified LVM emulator.

5.3 The Modified Emulator

First of all, let us discuss the machine simulator. Since we only study data-cache performance, we focus on data accesses to memory. The inserted function call to the cache simulator is as follows:

```
int Cache_Simulation(int owner, int type, unsigned address);
```

There are three parameters defined in this function: 'owner' is a flag to indicate who performs this memory access: the garbage collector is indicated by 1 and the mutator is indicated by 0; 'type' indicates the reference's type, 0 for read and 1 for write; and finally, 'address' stores the address of the reference.

According to their function, instructions in LVM can generally be classified as:

- control instructions,
- initialization instructions,
- load/store instructions,
- arithmetic instructions,
- branching instructions,
- dispatching instructions,
- built-in instructions,
- unification instruction pairs, and
- CGC collection related instructions.

Not all of these types of instruction access data in the memory directly, since some of them are handled through registers only. Passing a call to the cache simulator in the proper position of code requires understanding the behavior of LVM instructions clearly. Unfortunately, there are many instructions tending to access memory, it is impossible to analyze all of them in this thesis. Therefore, several typical instructions are discussed to show how the interface between the LVM emulator and the cache simulator is designed.

Let us take ALC, a very basic control instruction, as our first example.

Mnemonic:	ALC n
Operation:	allocate a c-frame with n cells
Bytecode:	alc od1
Function:	allocate a n-cell frame from the stack, link this new frame with the ancestor's frame to form a call-chain

The following is the segment of LVM implementation (without the cache simulator part):

```
case alc: af = st;
          st = st + OPD1;
          *af = cf;
          *(af + 1) = cp;
          *(af + 2) = st;
          cf = af;
          NEXT(2);
```

where `af` is the current active frame register pointing to the base address of the current active frame, `st` is the stack top register pointing to the top of the stack, and `OPD1` is a macro that gets the value of n in the ALC instruction. The first two lines of code allocate n cells from the stack top to form the current active frame. The next three assignments store three important values in the first three cells of this frame. The cell at the address

af holds the last active frame base address linking this new frame with the ancestor's frame to form a call-chain. The next cell *(af + 1)* stores the continuation point *(cp)*. The third one *(af + 2)* is the new top of the stack. From this description, we can see that there are three memory cell writes, so a call to the cache simulator is inserted following each write. The modified code is as follows:

```
case alc:  af = st;
           st = st + OPD1;
           *af = cf;
           Cache_Simulation(0,1,(unsigned)af);
           *(af + 1) = cp;
           Cache_Simulation(0,1,(unsigned)(af+1));
           *(af + 2) = st;
           Cache_Simulation(0,1,(unsigned)(af+2));
           cf = af;
           NEXT(2);
```

Besides the mutator's instructions, CGC makes a lot of memory data references as well. GC-compact is straightforward to analyze.

As we mentioned in the previous chapter, GC compact is a process that moves a temporary copy of collected results from the free area to the area just above the c-line. The following is a segment of source code for moving the collected data.

```
for(i = 0; i < ep; i++ ){
      hp[i] =  st[i];
}
```

where *ep* stores the total number of collected results, *hp*, the so-called c-line, divides the old and young areas, and *st* is the stack top, *i.e.* the base line of the free area. Since each movement of a cell causes a read and a write to that cell, this block movement yields

many reads and writes if the number of collected results is large. The code is modified as follows.

```
for(i = 0; i < ep; i++ ){
      hp[i] =  st[i];
      Cache_Simulation(1, 0, (unsigned)(st + i));
      Cache_Simulation(1, 1, (unsigned)(hp + i));
}
```

5.4 The Cache Simulator

So far, we have had several examples describing how to insert the calls to the LVM emulator. Now, let us discuss the algorithms of the cache simulator.

According to the cache architecture presented in chapter 4, we define a structure for caches as follows,

```
typedef struct{
      unsigned   Tag;
      int        *Valid, *Dirty;
}Cache_Line;
```

The field `Tag` indicates whether a memory block is stored in cache. The fields `Valid` and `Dirty` are valid bit and dirty bit. Since a cache block can be further divided into several sub-blocks, these two fields are defined as integer pointers allocated with several cells representing the valid bit and dirty bit for each sub-block. Although these three fields are defined in this structure, they are not used in all three cache caches. For example, `Dirty` will not be used in write-around caches.

Caches are initialized when LVM virtual machine is started up. Parameters, such as associativity, cache size and cache type, have to be provided. During the execution of

each instruction, calls to the cache simulator are made. The behavior of the caches is simulated along the execution of the program. After the execution, statistical results, such as miss ratio, are evaluated. At the same time, the results of the program are displayed. Let us examine the algorithms to simulate write_validate, write_around, and fetch_on_write caches.

```
void Write_Validate(int owner, int type, unsigned address)
{
      switch(type){
            case 0:     if(!Read(owner, address))
                        Load(owner, address, 0); /*read miss*/
                        break;
            case 1:     if(!Write(owner, address))
                        Load(owner, address, 1); /*write miss*/
                        break;
            default:    display error messages;
      }
}

void Write_Around(int owner, int type, unsigned address)
{
      switch(type){
            case 0:     if(!Read(owner, address))
                        Load(owner, address, 0);
                        break;
            case 1:     if(!Write(owner, address))
            default:    display error messages;
      }
}

void Fetch_on_Write(int owner, int type, unsigned address)
{
      switch(type){
            case 0:     if(!Read(owner, address))
                        Load_Fetch(owner, address, 0);
                        break;
            case 1:     if(!Write(owner, address))
                        Load_Fetch(owner, address, 1);
                        break;
            default:    display error messages;
      }
}
```

For write_validate caches, no matter whether read or write misses occur, the missed block will be loaded into cache. For write_around caches, however, only read misses result in loading a block. The missing blocks of write misses will be loaded into memory directly, bypassing the cache itself. For fetch_on_write caches, loading will be executed when read or write misses occur, but the loading is different from the one for write_validate caches. In general, the misses (read or write) have to occur before loading.

```
int Read(int owner, unsigned address)
{
   • Split the address of the block and search the word in the
     cache by its tag.
   • If the tag matches and the corresponding valid bit is on,
     i.e., read hit occurs, do the following,
     1. LRU update;
     2. Return 1 to indicate read hit;
   • If Valid == 0, return 0;
}

int Write(int owner, unsigned address)
{
   • Split the address of the block and search the word in the
     cache by its tag.
   • If the tag matches and the corresponding valid bit is on, do
     the following,
     1. Set the corresponding dirty bit on to indicate the
        corresponding sub block is written. This is not
        applicable for write_around caches.
     2. LRU update;
   3. Return 1 to indicate write hit;
   • If Valid == 0, return 0;
}
```

void Load(int owner, unsigned address, int inWrite)

{

- *Split the address of the block;*
- *If the tag matches, i.e., the misses occur because of the valid bit is off.*
  1. *Set the valid bit on to indicate writing the block into the corresponding sub_block in the cache;*
  2. *Set the corresponding dirty bit to "inWrite".*
- *If the tag does not match,*
  1. *Find & allocate the least recent used blockin the cache;*
  2. *Write the word to the corresponding sub block in cache, including set Tag field and turn Valid field on;*
  3. *Set the dirty bit to 'inWrite';*
  4. *Invalidate the remaining sub block within the block;*
- *Do LRU update;*

}

void Load_Fetch(int owner, unsigned address, int inWrite)

{

- *Split the address of the block;*
- *If the tag matches, i.e., the misses are caused by valid bit being 'off'.*
  1. *Set the valid bit on to indicate writing the block into the corresponding sub block in the cache;*
  2. *Set the dirty bit to "inWrite", which means dirty bit is on for writes, off for read;*
- *If the tag does not match,*
  1. *Find & allocate the least recent used block;*
  2. *Fetch and load the whole line, (not only the sub block itself), i.e., setting tag filed, setting each valid bit to on and setting each dirty bit to 'inWrite';*
  3. *Set the 'Dirty' field to 'inWrite';*
- *Do LRU update;*

}

If the cache uses write-back as write hit policy, the dirty sub block should be written to the memory. It is called "cast-out" (from the cache point of view) or "write-back" (from the main memory point of view). If the cache write hit policy is write-through, invalidating is still applicable, but there is no need to write the block back to memory because write through guarantees the consistency of memory and cache.

Chapter 6. Performance Analysis

6.1 Benchmarks

Six benchmarks tested under the LVM system are:

- Traveling Salesman Problem (*tsp*),
- DNA matching (*match*),
- Recursive integer arithmetic (*tak*),
- Quick sort and naïve reverse (*qsnv*),
- Boyer-Moore Theorem prover (*boyer*), and
- Build and query a database (*browse*).

Traveling Salesman Problem (*tsp*) and DNA matching (*match*) come from [20]. For the *tsp* program, tours of 30, 50, 70 and 100 were computed. The *match* program implements a dynamic algorithm for comparing DNA sequences. One sequence of length 32 was compared with others of length 20, 100, 500 and 1000. Another two programs are part of the Berkley Benchmark suite. Benchmark *tak* implements a recursive arithmetic computation. It was tested with input (x, 16, 8), where x was chosen from 22, 24, 26 to 28. The reason for choosing this benchmark is to test LVM performance in case no long-lived (heap) objects are involved in execution. Benchmark *qsnv* is a quick-sort followed by naïve reverse, and has been tested with lists of 500, 1000, 2000 and 5000 integers. This benchmark is particularly interesting because some collected objects may survive through many collections. Boyer-Moore Theorem Prover (*boyer*) is a prover for quantifier free logic for recursive functions over the integers and other finitely generated structures.

It is classic Lisp program since it recursively examines and constructs lists at a great rate. Inputs to *boyer* are made up by the previous tautology in conjunction with a truth value in each subsequent test. The other one is Build and Query a Database (*browse*), tested with input sizes 100, 500, 1000 and 2000.

Benchmarks were run under two different hardware environments:

- SPARC IPC workstation with an 8MB memory and a 64KB cache,
- Sun Enterprise 4000 running SunOS 5.6, with 16K D-cache, 1MB E-cache, 256MB main memory, dual sun4u CPU, and 168MHz clock.

We identify the hardware environment when we present our data.

6.2 Discussion of Memory References

During a program run, a garbage collector imposes both direct and indirect costs. Directly, the collector itself executes some instructions and causes some cache misses. The number of misses depends upon the collector's own memory reference patterns. Indirectly, there are two ways in which the collector affects the number of cache misses. Each time the collector is invoked, its memory references remove some, or possibly all, of the program's state from the cache and when the program resumes, more cache misses occur as that state is restored. On the other hand, the collector can also move data objects in memory, which improves the objects' reference locality, thereby decreasing the program's miss count. [1]

Therefore, memory references with and without garbage collection are compared in this section to show how the collector improves mutator locality. Further, the collector's memory references are presented and studied. These figures allow us to

visualize their memory reference patterns, and support the statistical results presented later in this chapter.

First of all, let us look at Figure 6.1, which shows memory references of *tsp*(30) without garbage collection. In the figure, the x-axis stands for the sequence of memory references; the y-axis represents the addresses in the stack. Data are presented every 1,024 memory references, some of which are extracted and shown because the memory references log is extremely long. Some memory references in the trail area (the "pulse" at the right of the figure) are scaled manually to fit into the figure (for example, the maximum point was 3999934, but it is scaled to 99934 in the figure).

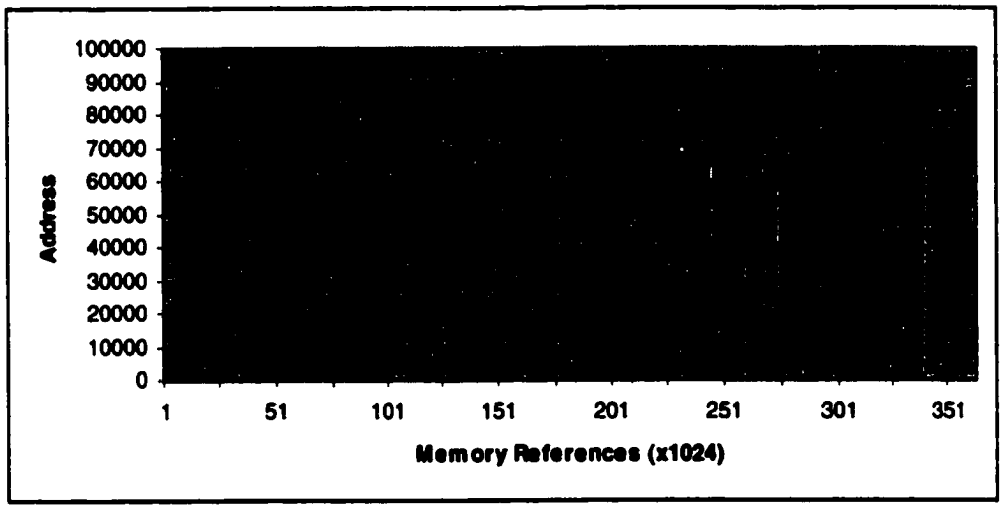


Figure 6.1 Memory References of *tsp*(30) without Garbage Collection

As we mentioned before, the LVM system adopts single stack policy. So, during execution of the program, the stack is occupied in a linearly increasing pattern if no garbage collector is employed to free up memory.

Figure 6.2 shows data of *tsp*(30) with garbage collection. The size of cache simulated is 64K and the cache-limit is 40K.

When memory requirements accumulate enough to meet the garbage collection condition, the collector begins to work and releases some space for reuse. Between two garbage collections, the stack is still occupied in a linear pattern. However, because the collector collects garbage and releases memory periodically, the program can reuse the stack space again and again. This is why we can see a lot of "saw teeth" in the figure. Therefore, with garbage collection, the program requires much less memory. The stack space is no longer occupied in a linearly increasing pattern, but in a constant pattern. This feature makes it possible for a cache to contain most working memory reference spaces, thus reducing misses and resulting in good cache performance.

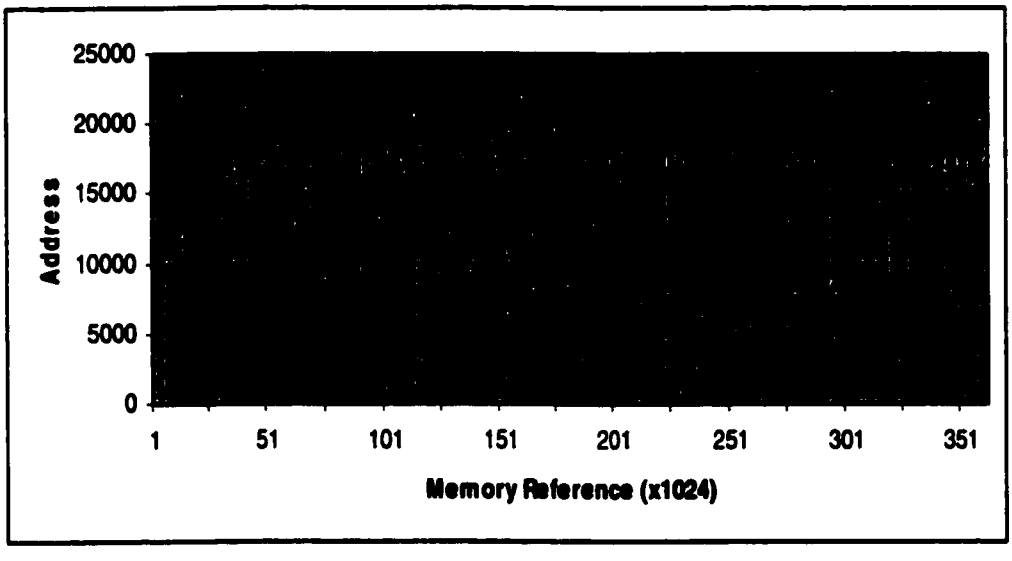


Figure 6.2 Memory References of *tsp*(30) with Garbage Collection

Theoretically, we should see some memory references to the trail area (the "pulse") during garbage collections. But since garbage collections do not issue many memory references, they are not sampled and shown in the figure. This is a good sign, because it means that the number of instructions that our collector executes is small, and thus our garbage collection cost is low.

We have shown the memory references of the *tsp*(30) mutator. Now let us look at the memory references of the collector.

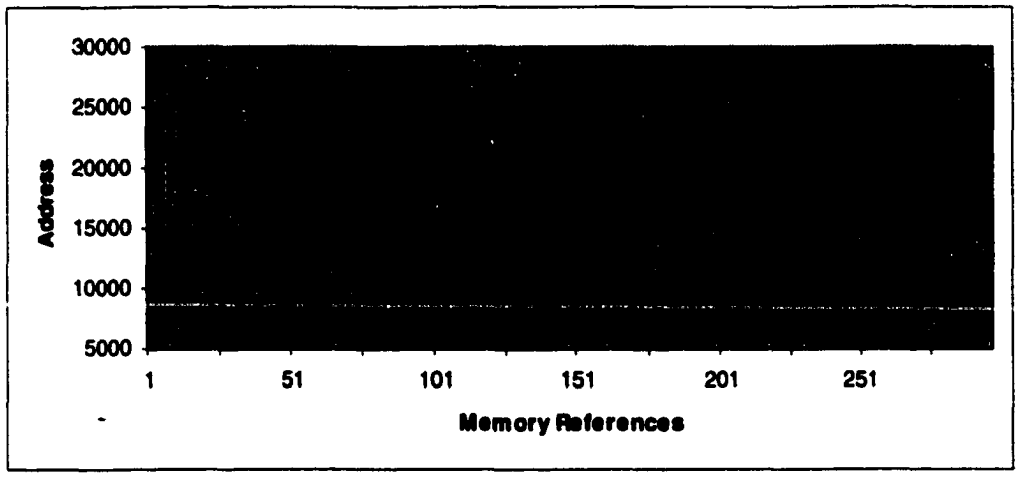


Figure 6.3 Memory References of the Collector

From Figure 6.3, we can see collector activity very clearly. At the beginning, the collector creates new roots based on the initial roots. Then it tries to trace the live objects and write them to the free area. During tracing stage, the references may cluster at the young and the old areas. Finally, the collector moves live objects down to the c-line, which is shown as waves at the right of the figure.

The collector only issues 296 memory references during this garbage collection. It is very small compare to the mutator. In addition, the references mostly focus around the c-line, the top of the young area and the trail area. They should be contained within a cache with reasonable size. Therefore, the miss ratio will not be high.

From the figures and discussion above, we see that the collector drastically decreases memory requirements of a mutator, making it possible to run a larger program in a system with a limited memory. Further, the collector improves the space locality of a

mutator, resulting in better cache performance. In addition, the collector itself issues very few references and has good space locality too.

6.3 Discussion of Cache-limit

Cache-limit controls the collection frequency, and is very important for the performance of the CGC algorithm. For a given cache, if the cache-limit is too high, the collector will be activated more frequently. This affects performance because frequent collections could add more overhead, maybe offsetting the improvement to mutators. On the other hand, if the cache-limit is too large, collections are not frequent enough to minimize cache misses. Now, we analyze how to decide a reasonable cache-limit.

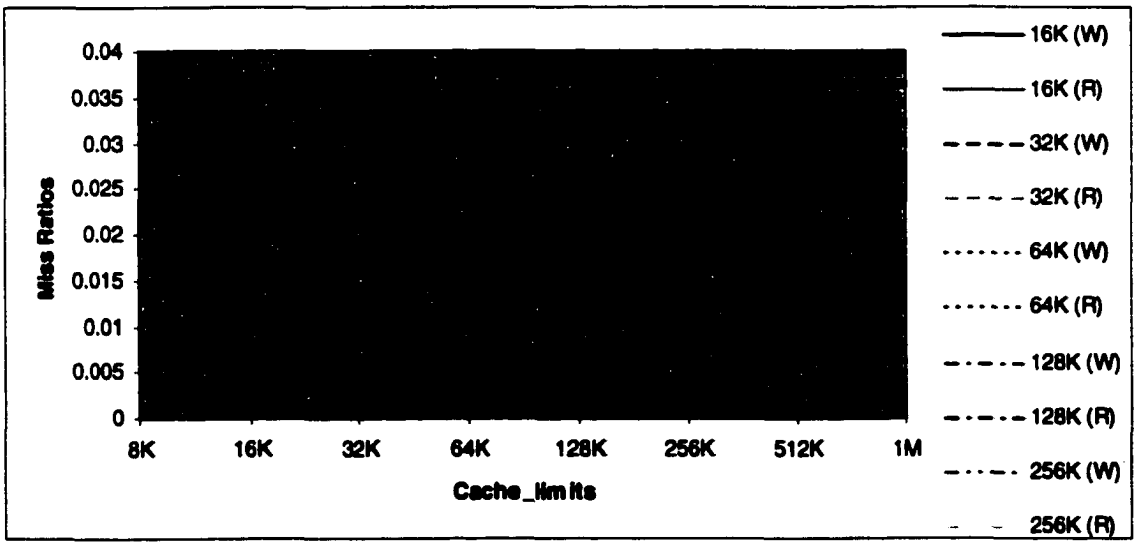


Figure 6.4 Write and Read Miss Ratios of the Mutator of tak22

Figure 6.4 and Figure 6.5 show data of the benchmark *tak*(22). The x-axis of each figure represents cache-limit, and the y-axis miss-ratios. Five direct-mapped fetch-on-write caches are considered here, with cache sizes 16K, 32K, 64K, 128K, and 256K (In the rest of this chapter, if not specified, caches are direct-mapped fetch-on-write data caches of size 64K, block size 32 bytes, no sub-block and cache-limit 40K if with garbage

collection). We separate read and write misses, mutator and collector as well. Figure 6.4 shows read and write miss ratios of the mutator, while Figure 6.5 shows those of the collector.

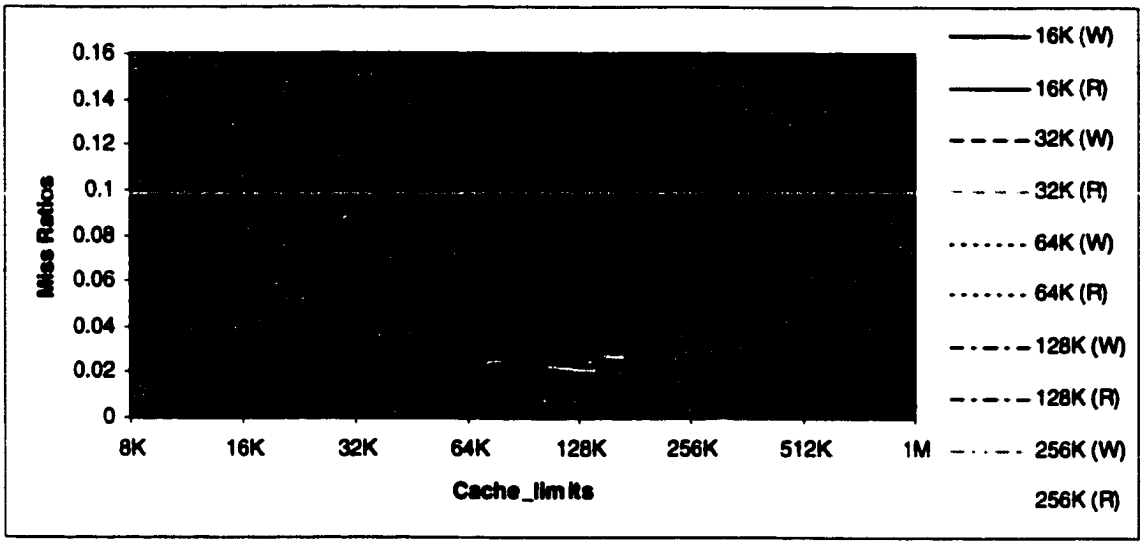


Figure 6.5 Write and Read Miss Ratios of the Collector of tak22

Figure 6.4 and Figure 6.5 show that when cache-limit is larger than the corresponding cache size, the miss rates of both mutator and collector are high and eventually have prominent plateaus. The values of the plateaus in Figure 6.4 approach the miss ratios without garbage collection. In order to keep cache misses low, we should choose cache-limit no more than the cache size. Does this imply we should choose a cache-limit as small as possible? No. If the cache-limit is too small, collector is activated more aggressively. This will increase garbage collection costs with consequent degeneration of the overall execution performance. In Figure 6.4 and Figure 6.5, when cache-limit is from ½ to 1 of the cache size, cache misses of both mutator and collector become higher and higher. During this stage, the misses are low compared to these when cache-limit is greater than cache size. At the same time, collection is not too frequent

because the cache-limit is comparably large. From our experimental results, half to 2/3 of the size of the cache would be a proper range for the cache-limit.

Appendix A gives the corresponding figures for benchmarks *qsnv*(1000), *tsp*(30) and *match*(20).

6.4 Cache Performance Analysis

Table 6.1 gives statistics of four benchmarks, run on the SPARC IPC workstation. All times are measured in seconds by the Unix timing facility that returns *usr/sys* elapsed time, and all memory related figures are in byte.

Test	cgc/size	stack	memory	time(E)	time(D)	E/D
Tsp(30)	720/12K	82K	8.36M	9.35/0.01	10.63/1.76	76%
Match(20)	600/12K	101K	8.12M	7.23/0.01	7.45/1.62	80%
Tak(22)	412/28K	272K	11.76M	22.1/0.01	22.9/3.18	84%
Qsnv(1000)	111/36K	98K	4.14M	5.50/0.10	5.50/0.10	100%

Table 6.1 Benchmark Statistics

Column *cgc/size* gives the actual counts of CGC invocations and the average size of garbage collected each time. Column *stack* gives maximum occupancy of the stack. Here we point out that the actual memory required in running each benchmark will never exceed double the corresponding stack size, because a free area of that size is more than enough to hold temporary copies. Column *memory* exhibits the summations of dynamic memory requirements, that is, maximum allocated stack sizes with CGC disabled. Note that a number in this column is the sum of total stack and heap allocations in traditional

Prolog implementations. The next two columns show execution times in gc-enabled and gc-disabled tests. The final column gives the ratios for gc-enabled and gc-disable tests.

The results in Table 6.1 are very promising. For these benchmarks, their performance with CGC is better than, or at least as good as when executed on a machine with infinite virtual memory. One reason is that the CGC algorithm is very efficient. It does not collect garbage, instead, it only collects useful objects (starting from a very small set of initial roots specified by the LVM compiler) with respect to dynamically partitioned generations. Another important reason is that the single stack paradigm incorporated with CGC improves locality. This greatly reduced overhead incurred by the gaps between cache and main memory, and between main memory and secondary virtual storage.

As we mentioned in section 6.2, the collector issues much fewer memory references than the mutator. Table 6.2 shows memory references statistics for four benchmarks (all numbers in millions). Caches simulated here are 64K data caches with 40K cache-limit.

test	gc-disabled	gc-enabled		
	mutator	mutator	collector	ratio
tsp(30)	10.93	9.679	0.239	2.4%
match(20)	5.587	5.407	0.601	2.4%
tak(22)	16.76	16.76	0.234	1.4%
qsnv(1000)	1.072	1.072	0.681	39%

Table 6.2 Memory References

From Table 6.2 we observe that benchmarks, except *qsnv*, only contribute a very small number of memory references to garbage collection, approximately proportional to

the garbage collection overhead. We also observe that CGC reduces the number of memory references in some benchmarks. This is a typical result *of variable-shunting* adopted by the CGC algorithm, *i.e.*, intermediate links in young generation are discarded during collection. However, the *qsnv* benchmark spends a big chunk of overhead on CGC. In fact, it is one of the worst cases for the CGC algorithm: successive collections repeatedly copy results surviving across all earlier collections. Does this mean that CGC is not suitable for such kind of programs? Further investigation on cache-performance gives a promising answer.

test	gc-disabled	gc-enabled		reduction
	mutator	mutator	collector	
tsp(30)	347,762	67,414	3,638	80%
match(20)	230,531	9,145	6,192	93.3%
tak(22)	371,893	143,803	602	61%
qsnv(1000)	130,372	2,991	618	97.2%

Table 6.3 Cache Misses

Table 6.3 gives the numbers of cache misses in simulating our benchmarks. The final column shows the percentage of reduction in gc-enabled and gc-disabled tests. For example, execution of *qsnv* yields 130,372 cache misses on a machine with infinite memory. However, this number was reduced to 3,609 (mutator + collector), a 97.2% reduction in cooperation with the CGC algorithm. This proves that the CGC improves the program cache performance by almost or more than enough to pay off its own cost.

Appendix B gives a complete set of benchmark statistics under the Sun Enterprise 4000 environment.

6.5 Discussion of Cache Parameters

Among all cache design parameters, cache size is the most important affecting performance. Figure 6.6 presents write and read miss ratios of fetch-on-write caches.

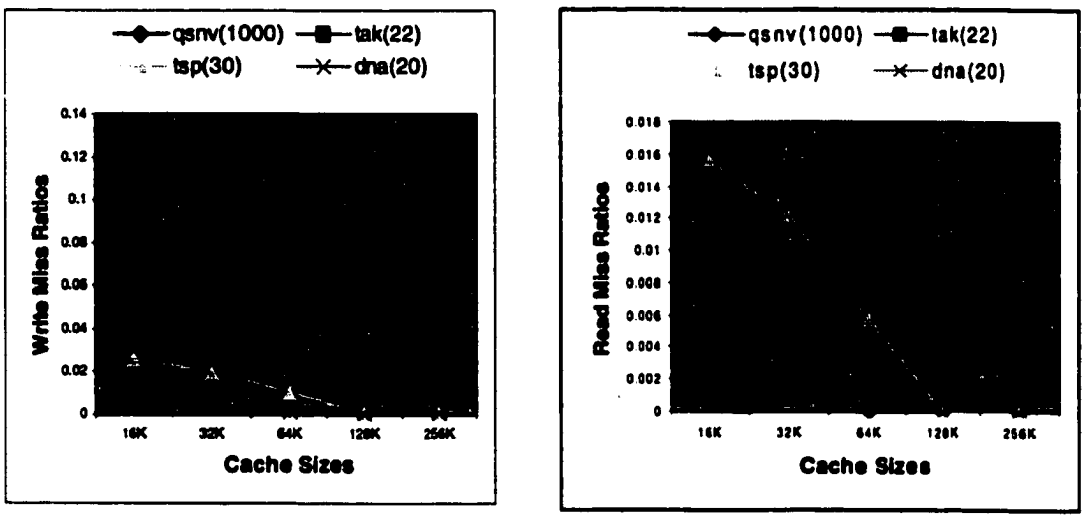


Figure 6.6 Write and Read Miss Ratios on Fetch-on-write Caches

Figure 6.6 shows that both write miss ratio and read miss ratio decrease drastically with increase in cache size. Figure 6.7 presents write and read miss ratios of *tak*(22) with and without garbage collection. It shows that the read miss ratio with or without garbage collection decreases similarly. However, write miss ratio with gc decreases much more sharply than without gc. This can be explained as follows. Most write misses in the LVM system are continuous, linear frame allocations. Without gc, each allocation must be a new address access, and result in a miss independent of cache size. On the other hand, with gc, since the memory has been reclaimed, some new allocations will reuse old spaces, resulting in cache hits. The larger the cache, the more cache hits. Therefore, a user program with our garbage collection will take better advantage of large caches.

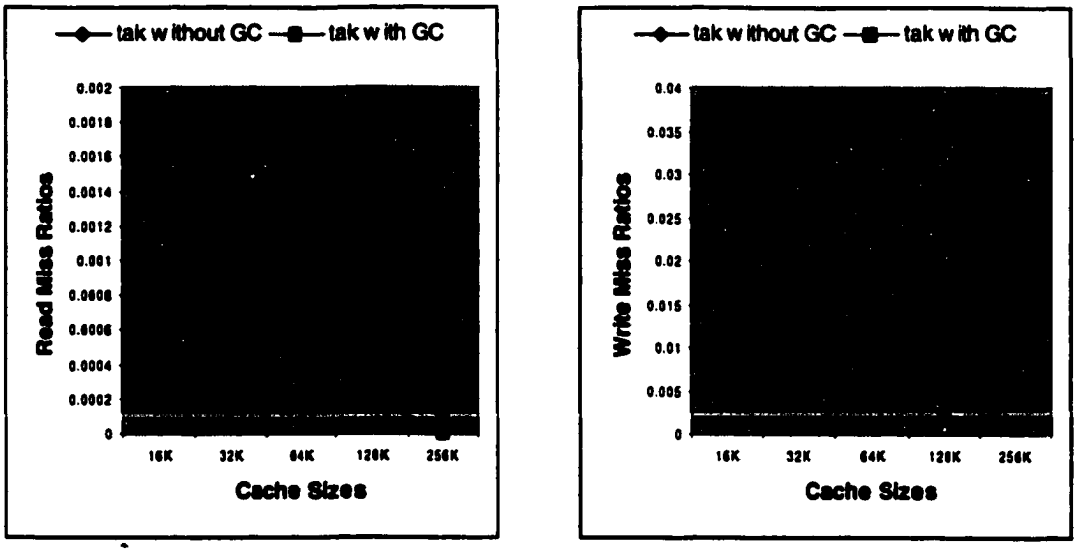


Figure 6.7 Write and Read Miss Ratios of tak with and without GC

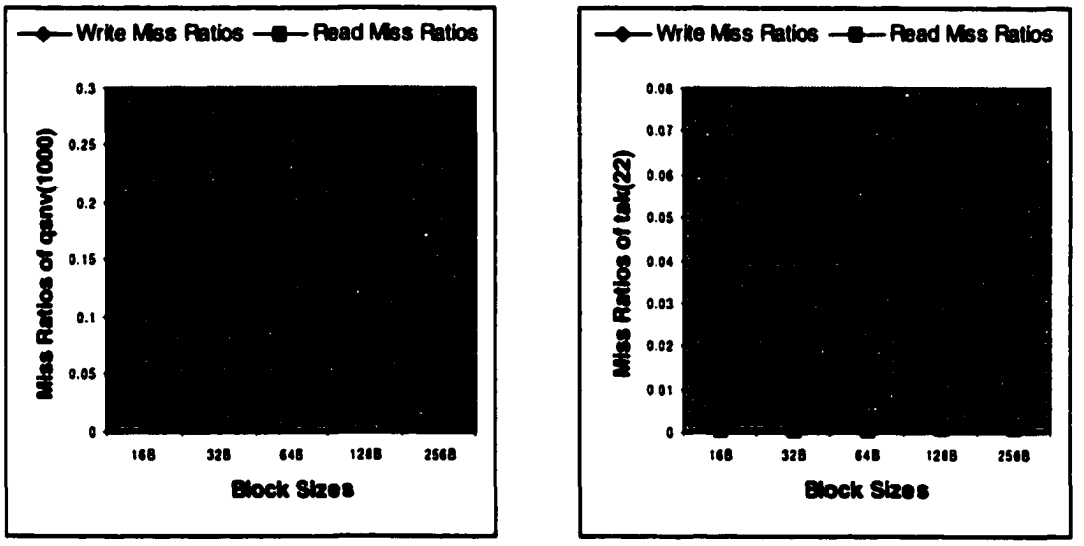


Figure 6.8 Miss Ratios on Fetch-on-write Caches with Different Block Sizes

Figure 6.8 shows miss ratio on caches (64K) with different block sizes. Large blocks make better use of spatial locality and thus reduce miss rates. They reduced write misses more because most write misses are allocation misses. However, if the block size becomes too large in comparison with overall cache size, and thus the number of blocks in the cache becomes too small, cache miss rates may again rise. This is why the read

miss ratios of *tak* rise a little in Figure 6.8. Large blocks can also have higher miss penalties, because there are more words to be transferred from memory to cache.

Cache Sizes	Fetch-on-write		Write-around	Write-validate
	read misses	total misses	read misses	read misses
4K	101700	235940	223059	117541
16K	11906	50827	49951	12751
64K	1147	21098	21034	1457
128K	2	1940	1933	3
256K	2	1940	1933	3

Table 6.4 Match(20): Miss Ratios for Write-validate, Write-around, and Fetch-on-write Caches

Table 6.4 shows misses for write-validate, write-around and fetch-on-write caches. The block sizes are 64 bytes with 2 sub-blocks, and cache-limits are 64K. The data are collected from benchmark *match*(20). Note that write-validate and write-around caches have higher read misses than fetch-on-write cache. Since they eliminate the write misses, they only have read-misses in the table. For fetch-on-write caches the *total misses* column are read misses plus write misses. Results show that misses for write-validate and write-around caches will never be larger than that total misses for fetch-on-write cache. That write-around caches add much more read misses than write-validate caches proves that a word written is more likely to be read again soon than other words in the block. These results show that write-validate caches perform much better than write-around and fetch-on-write caches, and write-around caches perform a bit better than fetch-on-write caches.

We have discussed the effect of cache size, block size and write policies. These discussions confirm that the memory reference patterns of benchmarks share the same properties: most writes are for allocation and most reads are to recently-written words. In addition, from our experiments, the impact of set associativity is not very great. The improvement can only be observed from direct-mapped to 2-way associative cache.

6.6 Comparison of LVM and SICStus 3.1

In this section, we compare LVM and SICStus 3.1 for our six benchmarks. SICStus Prolog 3.1 is an Edinburgh compatible Prolog implementation, owned and maintained by the Swedish Institute of Computer Science. It is one of the most widely used Prolog software systems.

The benchmarks were run on a Sun Enterprise 4000 environment. As the CGC algorithm is an incremental garbage collector invoked more frequently than other gc-algorithms, it is hard to estimate the actual time spent by the CGC invocations. It will be estimated by the following method.

In Appendix A, we use *P* to represent a gc-disabled test, and *P'* a gc-enabled test. Each table has these rows:

- $T(P)$ – execution time, and
- $R(P)$ – total memory references in execution.

Since P' involves garbage collection overhead, we have the following formulas:

- $T(P') = T(P_{gc}) + T(GC)$, and
- $R(P') = R(P_{gc}) + R(GC)$,

where P_{gc} represents the original program P incorporating garbage collection, and GC the CGC algorithm. As the number of memory references is roughly proportional to execution time, the time spent by the CGC can be estimated:

$$T(GC) \approx T(P') \times \frac{R(GC)}{R(P')}$$

Two execution modes of SICStus were tested, where S_f is fast-code and S_c compact-code. In each table, *gc-calls* gives the actual number of invocations of garbage collection; *garbage* gives the total number (in millions) of garbage collected by the collectors.

input-size		20	100	500	1K	10K	100K	500K	600K
S_f	*T(P')*	0.22	1.13	5.99	13.03	207	2694	32505	fail
	T(GC)	0.05	0.27	1.87	4.87	124	1881	27772	
	gc-calls	3	8	41	88	302	470	1241	
	garbage	6.08	30.45	162.26	344.44				
S_c	*T(P')*	0.63	3.03	15.48	32.26	402	4773	42462	fail
	T(GC)	0.05	0.28	1.81	50.4	124	1882	27584	
	gc-calls	3	8	41	88	302	470	1241	
	garbage	6.08	30.45	162.26	344.44				
LVM	*T(P')*	1.41	7.16	36.06	72.81	723	7240	33523	39,762
	T(GC)	0.01	0.04	0.36	1.46				
	gc-calls	42	214	1075	2150				
	garbage	6.94	35.3	177.5	354.9				

Table 6.5 DNA Matching – Comparison with SICStus 3.1

input-size		30	50	70	100	200
S_f	*T(P')*	0.18	1.24	4.49	17.91	266
	T(GC)	0.0	0.02	0.11	0.40	4.9
	gc-calls	0	1	3	10	90
	garbage	0	2.02	1012	38.32	
S_c	*T(P')*	0.56	3.92	14.48	58.59	899
	T(GC)	0.0	0.02	0.11	0.42	4.9
	gc-calls	0	1	3	10	90
	garbage	0	2.02	10.12	38.32	
LVM	*T(P')*	1.75	12.52	48.03	198	3375
	T(GC)	0.014	0.13	0.96	8.16	
	gc-calls	45	331	1178	4026	
	garbage	8.18	58.47	219.7	894.8	

Table 6.6 Travelling Salesman – Comparison with SICStus 3.1

input-size		500	1000	2000	5000
S_f	*T(P')*	0.03	0.17	0.82	5.98
	T(GC)	0.0	0.07	0.46	3.78
	gc-calls	0	1	3	12
	garbage	0	8.16	49.02	392.2
S_c	*T(P')*	0.14	0.52	2.03	14.73
	T(GC)	0.04	0.12	0.58	4.78
	gc-calls	2	4	16	102
	garbage	4.05	14.21	62.78	403.3
LVM	*T(P')*	0.29	1.18	4.8	30.15
	T(GC)	0.01	0.02	0.14	1.81
	gc-calls	6	24	102	665
	garbage	0.95	3.77	15.8	98

Table 6.7 Quick-sort and Naïve Reverse – Comparison with SICStus 3.1

input-size		1	2	3	4	7
S_f	*T(P')*	0.33	2.18	6.66	19.61	fail
	T(GC)	0.04	0.48	1.73	5.66	
	gc-calls	1	6	15	33	
	garbage	1.91	27.49	106.07	419.2	
S_c	*T(P')*	0.7	4.55	13.59	39.39	fail
	T(GC)	0.04	0.47	1.71	5.61	
	gc-calls	1	6	15	33	
	garbage	1.91	27.49	106.07	419.2	
LVM	*T(P')*	1.41	8.75	25.65	73.87	1,757
	T(GC)	0.07	0.52	1.79	7.38	
	gc-calls	19	138	408	1343	
	garbage	4.29	29.14	87.97	256.2	

Table 6.8 Boyer-Moore – Comparison with SICStus 3.1

input-size		100	500	1K	2K	10K	100K	200K
S_f	*T(P')*	0.37	1.82	3.81	7.68	42.98	2341	7,996
	T(GC)	0.0	0.0	0.1	0.23	2.47	27.85	
	gc-calls	0	0	1	3	9	19	
	garbage	0.0	0.0	1.81	8.9	76		
S_c	*T(P')*	0.86	4.59	9.31	19.01	fail	fail	fail
	T(GC)	0.0	0.0	0.1	0.2			
	gc-calls	0	0	1	3			
	garbage	0.0	0.0	1.81	8.9			
LVM	*T(P')*	2.9	14.7	29.68	60.53	371	2,996	7,988
	T(GC)	0.0	0.001	0.03	0.12			
	gc-calls	0	2	6	14	74		
	garbage	0.0	0.32	0.95	2.5	11.8		

Table 6.9 Browse – Comparison with SICStus 3.1

input-size		22	24	26	28
S_f	*T(P')*	0.43	1.04	2.17	4.79
	T(GC)	0.0	0.0	0.0	0.0
	gc-calls	0	0	0	0
	garbage	0.0	0.0	0.0	0.0
S_c	*T(P')*	1.51	4.13	9.93	21.67
	T(GC)	0.0	0.0	0.0	0.0
	gc-calls	0	0	0	0
	garbage	0.0	0.0	0.0	0.0
LVM	*T(P')*	3.4	9.2	22.1	48.08
	T(GC)	0.0	0.0	0.0	0.0
	gc-calls	49	137	336	742
	garbage	11.3	31.7	76.2	166.4

Table 6.10 Tak – Comparison with SICStus 3.1

When input size is small, SICStus is about 3-8 times faster than LVM. For arithmetic-intensive benchmarks, such as *tak* and *tsp*, LVM is even slower. Nevertheless, this is an acceptable range of performance ratio for comparing a binary-code engine against a byte-code emulator. When we increase the input sizes, some benchmarks retain their performance ratio, whereas others greatly narrow the performance gap and at certain breakthrough points they perform better than their counterparts under SICStus. How could this happen? The key is garbage collection. For example, among the 32,505 seconds of the *match*(500k) benchmark under SICStus, 27,772 seconds are spent on garbage collection. That is, the performance penalty caused by garbage collection reaches 85 percent for this special case. On the other hand, LVM spent much less time on garbage collection. From our experiments, time consumed by the CGC algorithm is (linearly) proportional to input size. Furthermore, the costs of garbage collection are almost paid off by improvement in cache performance. Note that measurements using a wide range of applications must be tested. Nevertheless, benchmarks and their performance studied in this thesis suggest that LVM incorporated with the single stack paradigm and CGC

algorithm offers a novel, practical technology in the design of high-performance Prolog systems.

Chapter 7. Conclusion

7.1 Conclusions

We have presented a study of cache performance influenced by the CGC algorithm and found important factors influencing the latter's performance. Simulation results from the cache simulator fully support the experimental results gathered from the LVM system: the cost of CGC can be paid by improved cache performance. With small input size, SICStus is about 3-8 times faster than LVM. This is an acceptable range of performance ratio in comparing a binary-code engine against a byte-code emulator. With increased input size, some benchmarks keep the same performance ratio, whereas others greatly narrow the performance gap and at certain breakthrough points perform better than their SICStus counterparts.

Further, we found that memory reference patterns of our benchmarks share the same properties: most writes are for allocation and most reads are to recently-written objects. In addition, the results provided by this thesis show that the write-miss policy can have a dramatic effect on cache performance of the benchmarks. A write-validate policy gives the best performance.

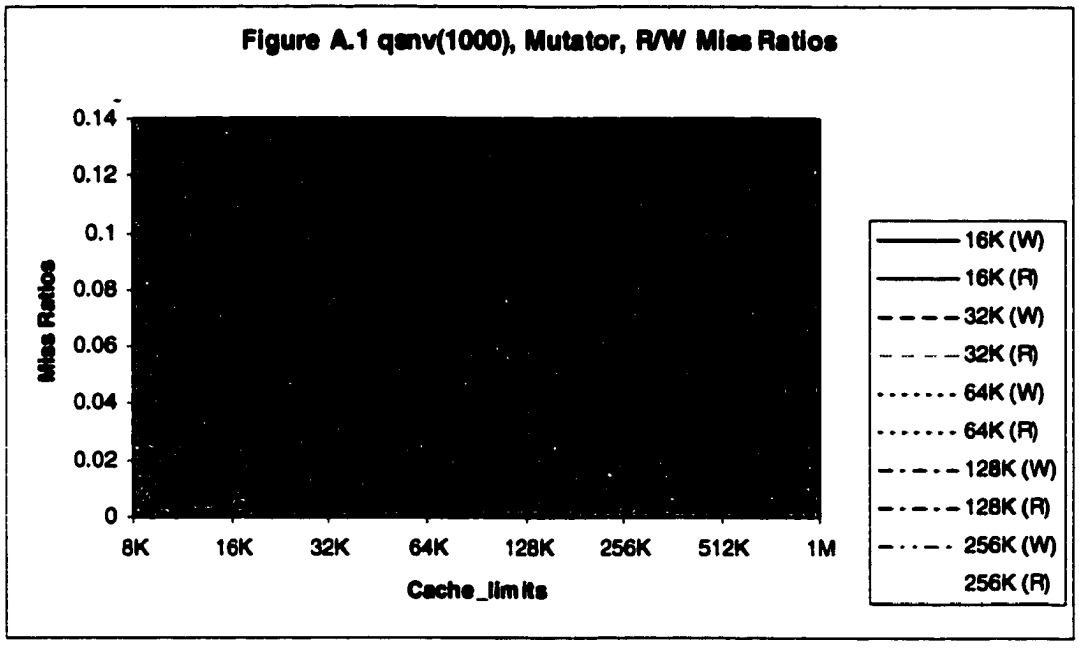
Finally, we point out that the concept of combining CGC and the single stack paradigm is probably more important than the actual implementation. Results presented in this study might be useful in related disciplines of functional logic, as well as object-oriented programming.

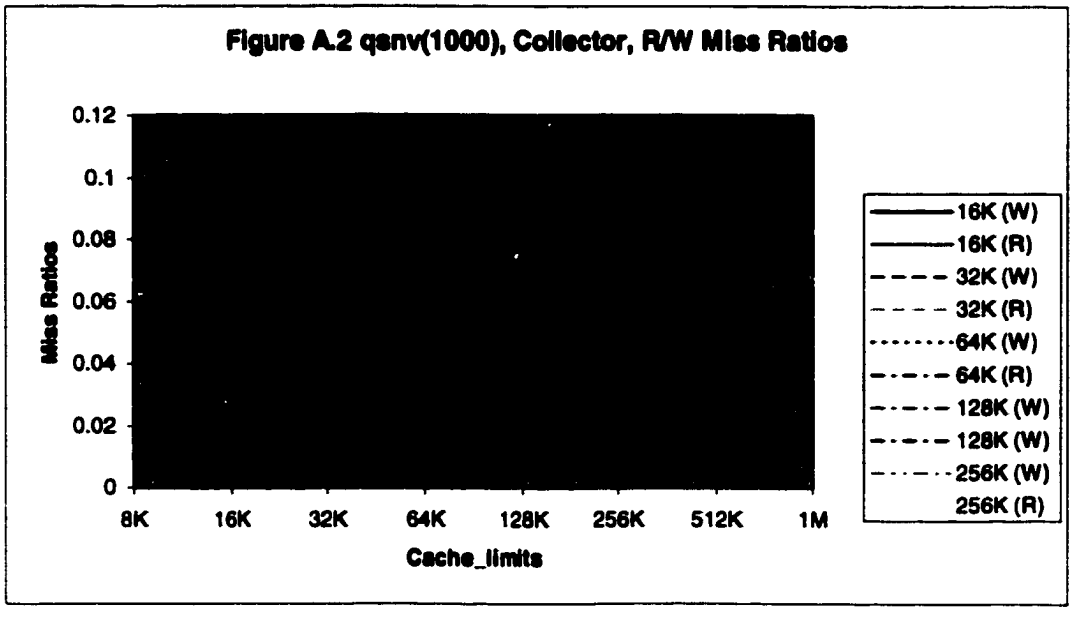
7.2 Future Work

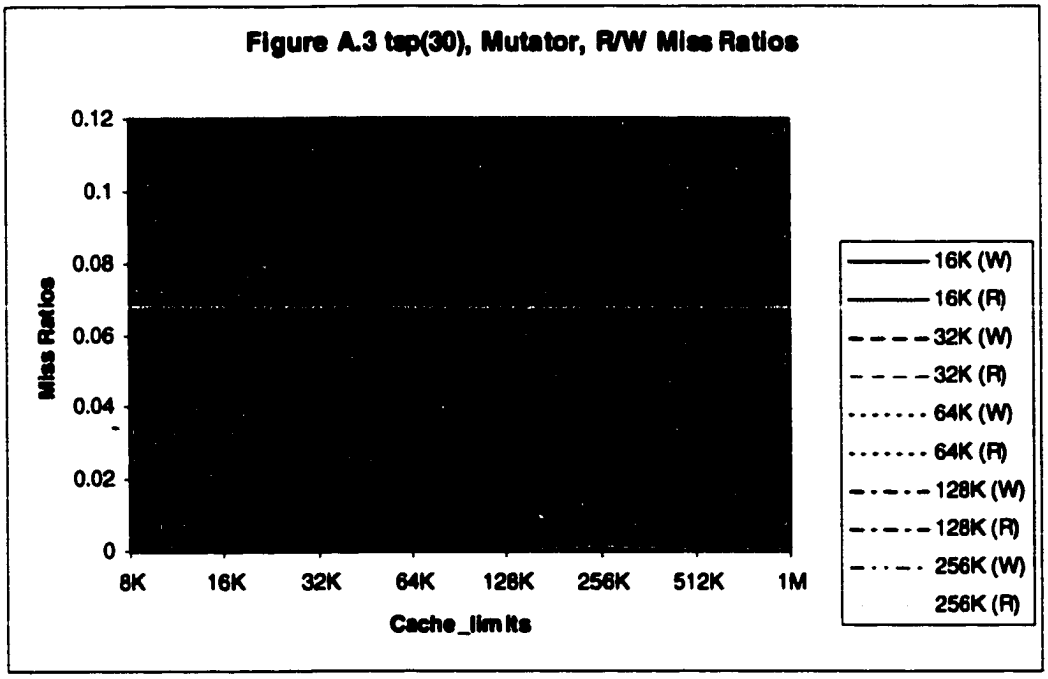
There are some constraints in our current cache simulation. Firstly, only one level of caching is considered. A more sophisticated simulator with multiple level caching is left to future work. Secondly, only data cache performance is studied. It is our intention to study that both data-cache and instruction-cache to evaluate CGC performance more completely.

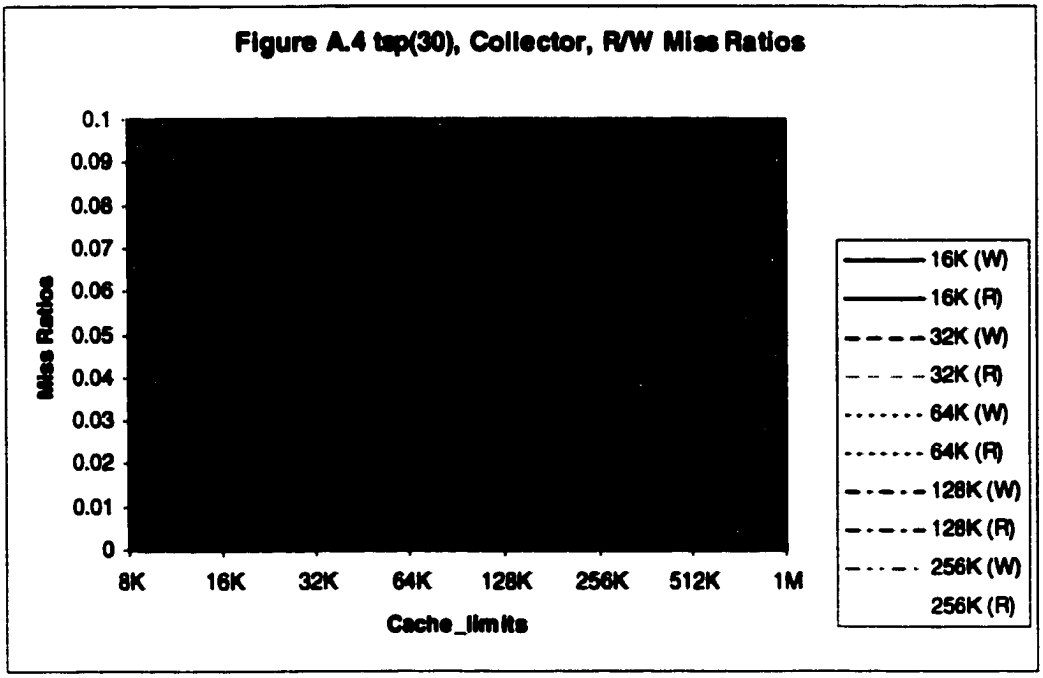
Appendix A. Cache-limit Figures

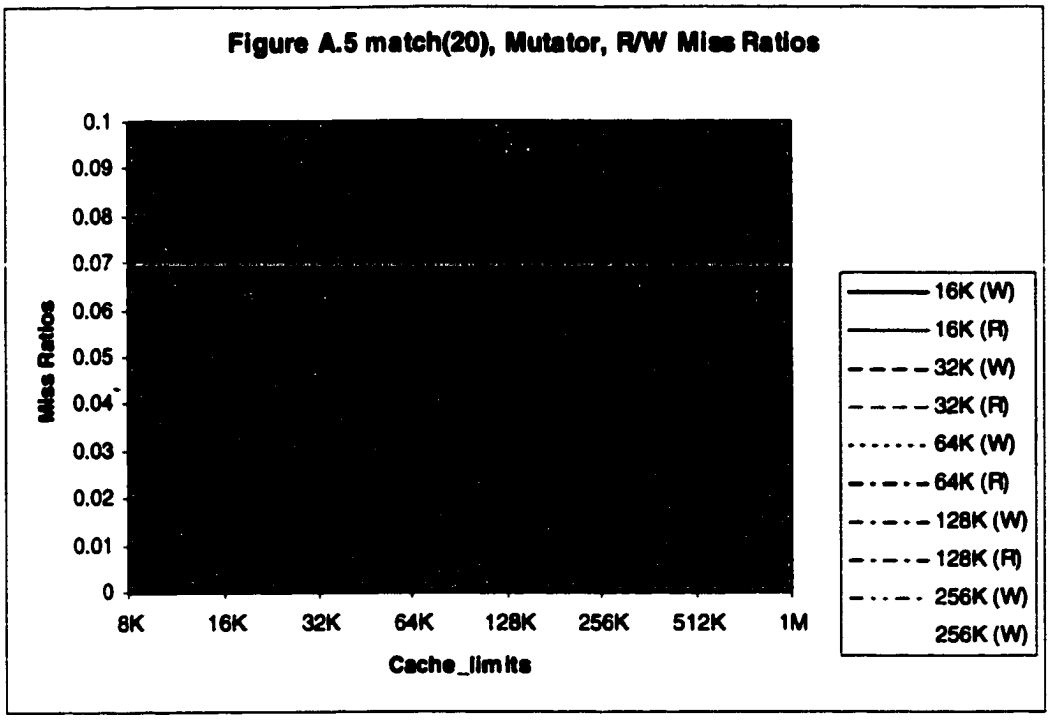
Appendix A shows miss ratios of our benchmarks varying with cache-limits. The x-axis of each figure represents cache-limits, and the y-axis indicates miss ratios. Five direct-mapped fetch-on-write caches are considered here, with cache size 16K, 32K, 64K, 128K, and 256K respectively, block size 32 bytes and no sub-block.

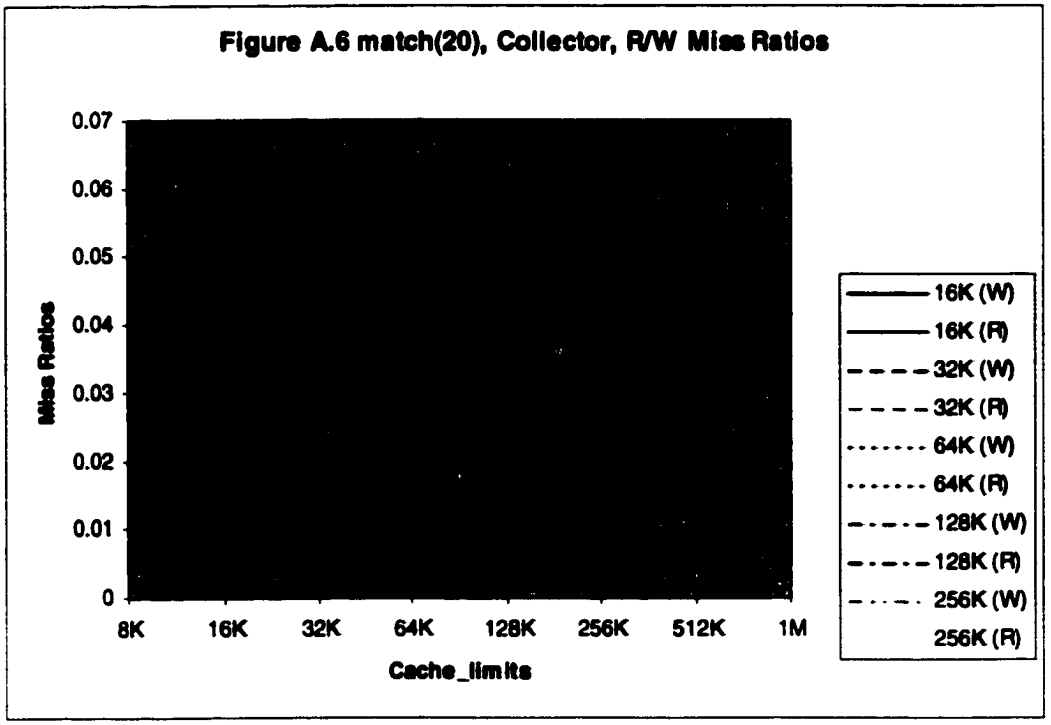












Appendix B. CGC Benchmarks Statistics

Appendix B presents benchmark statistics from the LVM emulator. Six benchmarks with different input size were executed on the Sun Enterprise 4000 environment. The maximum memory *malloc* can dynamically allocate is 256MB, so programs whose memory requirements lie beyond this limit could not be executed without garbage collection. Direct-mapped, fetch-on-write, 32-words/line cache was simulated. The Parameters to the simulator were 256KB cache size with 160KB cache-limit.

Each benchmark with the same input size was tested with gc-enabled and gc-disabled. As a matter of fact, a gc-disabled benchmark has all gc instructions removed. In each table, *P* represents gc-disabled test, and *P'* indicates gc-enabled test. The first row shows the name of the benchmark, and the second row gives the input sizes, and the third row presents the memory requirements for both test cases.

In column P, we show four rows of data: T(P) – the execution time, S(P) – the maximum stack used during execution, R(P) – the total memory references in execution, and M(P) – the cache misses collected in cache simulation. Execution times (in second) were gathered by the Unix timing facility that returns usr/sys elapsed times, and an average was taken from a reasonable number of repetitions.

In column P', more statistics are presented. The first two rows have the same meaning as in column P. However, as the execution of P' involves the overhead of garbage collection, we have the following formulas:

- $T(P') = T(P_{gc}) + T(GC)$,
- $R(P') = R(P_{gc}) + R(GC)$, and

- $M(P') = M(P_{gc}) + M(GC)$,

where P_{gc} represents the original program P incorporating garbage collection, and GC the CGC algorithm.

In general, R(P) > R(P'). This is because R(P') involves memory references of the original program P and those of the CGC algorithm. However, some gc-enabled benchmarks may reduce the total number of memory references, such that R(P) < R(P'). The reason is that CGC implements variable shunting that discards all intermediate references residing in young generation. As the CGC algorithm is an incremental garbage collector invoked more frequently than other gc-algorithms, it is hard to time the actual costs of the CGC invocations. However, as the number of memory references is roughly proportional to the execution time, the time spent by the CGC can be estimated by:

$$T(GC) \approx T(P') \times \frac{R(GC)}{R(P')}$$

The rest rows in column P' give statistics related to garbage collection. The row gc-instr indicates the number of garbage collection instructions being executed. The actual number of invocations of garbage collection is given by gc-calls. The next two rows show the total numbers of garbage and collected useful objects. The ratios of memory references and cache misses of GC against P' are given in the following rows. Finally, for each benchmark we show the ratios of execution times, memory references, and cache misses of P's against P.

match					
input-size		20	100	500	1000
memory		8.12M	40.6M	203.2M	
P	*T(P)*	1.43/0.01	7.28/0.9	37.33/3.61	
	S(P)	7.24M	36.36M	181.6M	
	R(P)	15.51M	77.55M	388.22M	
	M(P)	283895	1425673	7434320	
P'	*T(P')*	1.41/0.0	7.16/0.0	36.06/0.0	72.81/0.0
	S(P')	0.34M	0.44M	0.45M	0.58M
	R(P')	15.53M	77.76M	390.97M	786.82M
	R(P_{gc})	15.46M	77.29M	386.46M	772.91M
	R(GC)	0.07M	0.47M	4.51M	13.91M
	M(P')	112979	619098	3025097	6401148
	M(P_{gc})	110289	582201	2817542	5524466
	M(GC)	2690	36897	207555	776682
	gc-instr	680	3400	17000	34000
	gc-call	42	214	1075	2150
	garbage	6.94M	35.3M	177.5M	354.9M
	useful-obj	0.04M	0.18M	1.03M	2.05M
	R(GC)/ R(P')	0.005	0.006	0.01	0.02
	M(GC)/ M(P')	0.02	0.06	0.07	0.12
T(P')/ T(P)		0.98	0.88	0.88	
R(P')/ R(P)		1.00	1.00	1.01	
M(P')/ M(P)		0.40	0.43	0.41	

Table B.1 DNA Matching

tsp					
input-size		30	50	70	100
memory		8.36M	60.2M	224.8M	916M
P	*T(P)*	1.78/0.17	13.78/1.13	56.28/4.72	
	S(P)	8.36M	60.2M	224.8M	
	R(P)	20.09M	150.09M	597.14M	
	M(P)	384579	2662473	10957367	
P'	*T(P')*	1.75/0.0	12.52/0.0	48.03/0.0	204.2/0.0
	S(P')	0.31M	0.43M	0.53M	0.55M
	R(P')	19.78M	144.13M	556.02M	2404M
	R(P_{gc})	19.64M	142.0M	542.45M	2315M
	R(GC)	0.14M	2.13M	13.57M	89M
	M(P')	117271	753556	2737599	14814347
	M(P_{gc})	115699	712690	2423219	11853240
	M(GC)	1572	40866	314380	2961107
	gc-instr	13980	63800	174020	505100
	gc-call	45	331	1178	4026
	garbage	8.18M	58.74M	219.7M	894.8M
	useful-obj	0.008M	0.1M	0.5M	2.46M
	R(GC)/ R(P')	0.008	0.1	0.02	0.04
	M(GC)/ M(P')	0.01	0.05	0.11	0.20
T(P')/ T(P)		0.90	0.84	0.78	
R(P')/ R(P)		0.98	0.96	0.93	
M(P')/ M(P)		0.30	0.28	0.25	

Table B.2 Travelling Salesman

qsnv					
input-size		500	1000	2000	5000
memory		1.07M	4.14M	16.34M	100.8M
P	*T(P)*	0.29/0.0	1.18/0.1	4.7/0.3	29.5/1.6
	S(P)	1.07M	4.14M	16.34M	100.8M
	R(P)	3.11M	12.02M	47.26M	290.57M
	M(P)	34100	131900	519464	3199804
P'	*T(P')*	0.29/0.0	1.18/0.0	4.8/0.0	30.15/0.0
	S(P')	0.22M	0.30M	0.32M	0.69M
	R(P')	3.13M	12.16M	48.52M	310.76M
	$R(P_{gc})$	3.11M	12.02M	47.26M	290.58M
	R(GC)	0.02M	0.15M	1.26M	20.17M
	M(P')	7350	12407	21824	100404
	$M(P_{gc})$	12309	13207	20453	94938
	M(GC)	51	98	1389	5446
	gc-instr	1000	2000	4000	10000
	gc-call	6	24	102	665
	garbage	0.95M	3.77M	15.8M	98M
	useful-obj	0.02M	0.13M	1.08M	17.5M
	R(GC)/ R(P')	0.006	0.01	0.03	0.06
	M(GC)/ M(P')	0.007	0.01	0.06	0.05
T(P')/ T(P)		1.00	1.00	0.96	0.97
R(P')/ R(P)		1.01	1.01	1.03	1.07
M(P')/ M(P)		0.22	0.09	0.04	0.03

Table B.3 Quick-sort and Naïve Reverse

boyer					
input-size		1	2	3	4
memory		7.73M	47.96M	137.6M	387.5M
P	*T(P)*	1.32/0.02	8.46/0.35	24.46/1.51	
	S(P)	4.87M	30.48M	91.48M	
	R(P)	16.13M	101M	288.5M	
	M(P)	303827	1988545	5904241	
P'	*T(P')*	1.41/0.0	8.75/0.0	25.65/0.06	73.87/0.1
	S(P')	1.57M	2.58M	3.89M	7.6M
	R(P')	17.04M	106.95M	311.87M	898.5M
	$R(P_{gc})$	16.25M	101.06M	289.15M	810.4M
	R(GC)	0.79M	5.89M	22.7M	88.1M
	M(P')	291058	2086266	6336001	18201513
	$M(P_{gc})$	242376	1710270	4736617	11729694
	M(GC)	48682	375966	1599384	6471819
	gc-instr	38348	243711	687263	1899772
	gc-call	19	138	408	1343
	garbage	4.29M	29.14M	87.97M	256.2M
	useful-obj	0.64M	4.7M	18.05M	70M
	R(GC)/ R(P')	0.05	0.06	0.07	0.1
	M(GC)/ M(P')	0.18	0.18	0.25	0.35
T(P')/ T(P)		1.05	0.99	0.98	
R(P')/ R(P)		1.06	1.06	1.08	
M(P')/ M(P)		0.96	1.05	1.07	

Table B.4 Boyer-Moore Theorem Prover

browse					
input-size		100	500	1000	2000
memory		11.57M	50.08M	116.3M	232.8M
P	*T(P)*	2.9/0.0	14.81/0.02	30.08/0.05	61.25/0.05
	S(P)	0.08M	0.67M	1.46M	3.04M
	R(P)	33.91M	171.53M	343.45M	701.24M
	M(P)	149143	758146	1875851	9859692
P'	*T(P')*	2.9/0.0	14.7/0.0	29.68/0.0	60.53/0.02
	S(P')	0.08M	0.34M	0.55M	0.9M
	R(P')	33.91M	171.61M	343.62M	697.2M
	R(P_{gc})	33.91M	171.53M	343.44M	695.8M
	R(GC)	0M	0.08M	0.18M	1.4M
	M(P')	149149	761063	1875841	9692493
	M(P_{gc})	149149	751356	1841176	9466111
	M(GC)	0	9707	34665	226382
	gc-instr	200	1000	2000	4000
	gc-call	0	2	6	14
	garbage	0M	0.32M	0.95M	2.5M
	useful-obj	0K	5K	15K	34K
	R(GC)/ R(P')	0.0	0.0001	0.001	0.002
	M(GC)/ M(P')	0.0	0.004	0.02	0.02
T(P')/ T(P)		1.0	0.99	0.99	0.99
R(P')/ R(P)		1.0	1.0	1.0	0.99
M(P')/ M(P)		1.0	0.99	0.98	0.96

Table B.5 Build and Query a Database

tak(x, 16, 8)					
input-size		22	24	26	28
memory		11.8M	32.4M	78.0M	170.4M
P	*T(P)*	3.35/0.25	9.2/0.65	22.31/1.48	49.23/2.93
	S(P)	11.8M	32.4M	78.0M	170.4M
	R(P)	50.72M	139.63M	336.28M	734.33M
	M(P)	2643013	7276480	17524191	38269526
P'	*T(P')*	3.4/0.0	9.2/0.0	22.1/0.0	48.08/0.0
	S(P')	0.63M	0.71M	0.82M	0.82M
	R(P')	50.94M	140.25M	337.78M	737.62M
	R(P_{gc})	50.94M	140.25M	337.78M	737.61M
	R(GC)	882	0.002	0.006	0.013
	M(P')	2351838	6489599	15629936	34125057
	M(P_{gc})	2351780	6489462	15629598	34124332
	M(GC)	49	137	338	725
	gc-instr	226421	623337	1501232	3278264
	gc-call	49	137	336	742
	garbage	11.3M	31.7M	76.2M	166.4M
	useful-obj	0	0	0	0
	R(GC)/ R(P')	0.00002	0.00002	0.00002	0.00002
	M(GC)/ M(P')	0.00002	0.00002	0.00002	0.00002
T(P')/ T(P)		0.94	0.93	0.93	0.92
R(P')/ R(P)		1.00	1.00	1.00	1.00
M(P')/ M(P)		0.89	0.89	0.89	0.89

Table B.6 Recursive Interger Arithmetic

BIBLIOGRAPHY

[1] P. R. Wilson, M. S. Lam, and T. G. Moher. *Caching Considerations for Generational Garbage Collection.* In Proceedings on Lisp and Functional Programming, ACM, 1992, pp. 32-42

[2] B. G. Zorn. *The Effect of Garbage Collection on Cache Performance.* Technical Report CU-CS-528-91, Department of Computer Science, University of Colorado at Boulder, 1991.

[3] M. B. Reinhold. *Cache Performance of Garbage Collected Program*, SIGPLAN 94-6, ACM, 1994, pp. 206-217

[4] M. J. R. Goncalves and A. W. Appel. *"Cache performance of fast-allocating programs"*, In Proceedings of the 7th Conference on Functional Programming Languages and Computer Architecture. ACM Press, June 1995.

[5] N. P. Jouppi. *Cache Write Policies and Performance*, In 20th Annual International Symposium on Computer Architecture. San Diego, CA, May 1993, IEEE Press. pp. 191-201

[6] R. Jones and R. Lins. *Garbage Collection – Algorithm for Automatic Dynamic Memory Management.* John Wiley & Sons Press, 1996.

[7] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. *On-the-fly Garbage Collection: An Exercise in Cooperation.* Communications of the ACM, 21(11), November, 1978, pp. 965-975

[8] L. P. Deutsch and D. G. Bobrow. *An Efficient Incremental Automatic Garbage Collector.* Communications of the ACM, 19(7), July 1976, pp. 522-526

[9] D. S. Wise. *Stop and one-bit reference counting.* Technical Report 360, Indiana University, Computer Science Department, March 1993.

[10] H. Boehm and M. Weiser. *Garbage Collection in an Uncooperative environment.* Software Practice and Experience, 18(9), 1988, pp. 807-820

[11] B. G. Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms.* Ph. D thesis, University of California at Berkeley, March 1989. Technical report UCB/CSD 89/544.

[12] R. R. Fenichel and J. C. Yochelson. *A Lisp Garbage Collector for Virtual Memory Computer Systems.* Communications of the ACM, 12(11):611-612, November 1969, pp. 611-612

[13] C. J. Cheney. *A Non-recursive List Compacting Algorithm.* Communications of the ACM, 13(11), November 1970, pp. 677-678

[14] P. R. Wilson. *Uniprocessor Garbage Collection Techniques.* Technical report, University of Texas, January 1994. Expanded version of the IWMM92 paper to appear in Computing Surveys.

[15] D. M. Ungar and F. Jackson. *An Adaptive Tenuring Policy for Generation Scavengers.* ACM Transactions on Programming Languages and Systems, 14(1), 1992, pp. 1-17

[16] D. A. Barrett and B. G. Zorn. *Garbage Collection Using a Dynamic Threatening Boundary.* Computer Science Technical Report CU-CS-659-93, University of Colorado, July 1993.

[17] M. Bruynooghe. *The Memory Management of Prolog Implementations.* Logic Programming, Academic Press, 1982.

[18] S. Le Huitouze. *A New Data Structure for Implementing Extensions to Prolog.* PLILP'90, LNCS456, Springer, 1990.

[19] H. Touati and T. Hama. *A Light-weight Prolog Garbage Collector.* Proceedings of the International Conference on fifth Generation Computing Systems, 1988.

[20] J. Bevenmyr and T. Lindgren. *Simple and Efficient Copying Garbage Collection for Prolog.* PLILP'94, LNCS 844, Springer, 1994.

[21] P. Tarau. *Ecological Memory Management in a Continuation Passing Prolog Engine.* International Workshop IWMM92, LNCS 637, Springer, 1992.

[22] Y. Bekkers, O. Ridoux and L. Ungaro. *Dynamic Memory Management for Sequential Logic Programming Languages.* International Workshop IWMM92, LNCS 637, Springer, 1992.

[23] H. Ait-Kaci. *Warren's Abstract Machine: a Tutorial Reconstruction.* MIT Press, 1991.

[24] C. S. Mellish. *An Alternative to Structure Sharing in the Implementation of a Prolog Interpreter,* Logic Programming, Academic Press, 1982.

[25] X. Li. *A New Term Representation Method for Prolog,* To appear on J. Logic Programming, Vol. 34(1), 1998.

[26] X. Li. *Exploring Single Stack Architecture for Prolog.* Submitted to Journal of Functional and Logic Programming, 1998.

[27] Y. Wang. *Compiling Prolog to Logic-Inference Virtual Machine.* Master thesis, Lakehead University, May 1999.

[28] P. Messina, D. Culler, W. Pfeiffer, W. Martin, J. Oden and G. Smith, *The High-Performance Computing Continuum: Architecture* Communications of ACM., 41(11), 1998, pp. 36-44

[29] M. D. Hill and A. J. Smith, *Evaluating Associativity in CPU Caches,* IEEE Trans. on Computer, Vol. 38, No. 12, 1989, pp. 1612-1630

[30] H. S. Stone, *High-Performance Computer Architecture,* Addison-Wesley Publishing Company, 1990.