CACHE PERFORMANCE OF CHRONOLOGICAL GARBAGE COLLECTION

Yuping Ding and Xining Li

Department of Computer Science
Lakehead University
Thunder Bay, ON, Canada P7B 5E1
Email: ypding@flash.lakeheadu.ca, xli@flash.lakeheadu.ca

ABSTRACT

This paper will present the cache performance analysis of Chronological Garbage Collection algorithm used in the LVM system. The 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 recuperate space not as deliberate garbage collection operation but as a natural activity of the LVM engine to gather useful objects. This algorithm takes 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 an emulator to do the trace-driven cache simulation. Direct-mapped cache and set-associative cache with different cache sizes, block sizes and set associativities are simulated and measured. The objectives of this simulation are to verify and validate our experimental results, and to find important factors which influence the performance of the CGC algorithm.

1. INTRODUCTION

As the gap of speeds between processors 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, automatic storage reclamation is essential for practical implementations. The 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, the Chronological Garbage Collection (CGC), as a part of its engine.

We have implemented an experimental LVM emulator which 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 runtime 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 its own cost.

The related problems of the cache performance have been widely studied by other researchers. From their measurements of four Scheme programs, Wilson et al. [1] were the first to suggest 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 shown that the data-cache miss ratios of the programs are 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 function style should perform well when a simple linear storage allocation and an infrequently-run generational compacting collector are employed. He also claimed that sophisticated collectors intended to improve the cache performance are unlikely to be necessary. 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.

In order to verify and validate our experimental results, and to find important factors which influence 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 be able to determine the extend to which the cache performance of the test programs has been improved under the CGC algorithm, we have simulated benchmarks with different cache sizes, block sizes, and set associativities. With regard to the cache write policies, we only consider two popular architectures, fetch-on-write and write-validate, in our simulation.

2. THE LVM AND THE CGC ALGORITHM

The LVM uses a single stack to replace the roles of the traditional environment stack and heap; it adopts a hybrid of Program Sharing [7] and Structure Copying [8] to represent Prolog terms, and it cooperates with the CGC to manage the memory automatically. The LVM compiler transforms a Prolog program into the LVM bytecode. The compiler will do the qualitative garbage estimation and insert garbage collection instructions as well as their associated initial root sets into the proper positions of the control sequence.

The LVM engine is an interpreter which sequentially fetches a LVM instruction, tokenizes it and executes it. When a garbage collection instruction is met, the LVM engine divides the whole stack into two dynamical generations: objects in the young generation are threatened for collection and the objects in the old generation are immune from collection. However, the LVM engine does not invoke the CGC algorithm unless the old-young generation gap is greater than a predefined system constant: the cache-limit. This condition looks like a two-edged sword. On one side, we want to control the collection frequency such that the collector is not invoked until there is a reasonable amount of accumulated garbage. On the other side, we want to control useful objects more frequently than ordinary copying/generational collectors so that most working objects are kept in the cache.

The CGC algorithm takes advantages from several existing garbage collection algorithms [9]. Like copying, 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 area: 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 way to divide generations. Finally, CGC controls the frequency of collector invocations by a factor of cache size, and therefore collects garbage incrementally with a trivial pause time.

During collection, all shared (PS) instances will be transformed to copied (SC) instances. As soon as a young instance has been copied, a forward pointer is set which will prevent from creating duplicated copies. Copied objects are temporarily placed in the free area, however, references in these copies have already been calculated to the final destination addresses. In consequence, the compact phase will move these copies, by a simple wordwise loop, back to the young area without any pointer adjustment. CGC does not need early reset because the LVM compiler has already taken aways uninitialized variables from the initial root set. Furthermore, CGC implements a cheap variable shunting by simply dropping all intermediate references which reside at the young generation.

Four benchmarks have been tested under the LVM system. Traveling Salesman Problem (tsp) and DNA matching (dna) come from [8]. For tsp, tours of 30 was computed. The dna program implements a dynamic algorithm for comparing DNA-sequences. One sequence of length 32 was compared with other 20 sequences. The other two programs are part of the Berkley Benchmark suite. Tak implements a recursive arithmetic computation. It was tested with an input (22, 16, 8).

The reason of choosing this benchmark is to test the LVM performance for the case that no long-lived (heap) objects are involved in execution. Benchmark gssv is a program of quick-sort followed by naive reverse and was tested with a list of 1,000 integers. This benchmark is particularly interesting because a fraction of collected objects might survive through many collections. All benchmarks were run on a SPARC IPC workstation with a 8MB memory and a 64KB virtual cache.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>gc/size</th>
<th>stack</th>
<th>memory</th>
<th>time(D)</th>
<th>time(S)</th>
<th>L/D</th>
</tr>
</thead>
<tbody>
<tr>
<td>tsp20</td>
<td>130/12K</td>
<td>82K</td>
<td>3.95M</td>
<td>6.13/0.01</td>
<td>44.63/1.36</td>
<td>80%</td>
</tr>
<tr>
<td>dna20</td>
<td>500/12K</td>
<td>101K</td>
<td>8.12M</td>
<td>7.25/0.02</td>
<td>7.42/1.32</td>
<td>80%</td>
</tr>
<tr>
<td>tsp22</td>
<td>921/28K</td>
<td>272K</td>
<td>11.76M</td>
<td>32.1/0.01</td>
<td>22.9/3.18</td>
<td>84%</td>
</tr>
<tr>
<td>gssv</td>
<td>111/26K</td>
<td>98K</td>
<td>4.15M</td>
<td>5.57/0.01</td>
<td>3.50/0.15</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table 1: Benchmark Statistics

Table 1 gives statistics of benchmarks. All times are measured in second by the Unix timing facility which returns user/sys elapsed time, and all memory related figures are in byte. Column gc/size gives the actual counts of GC invocations and the average size of garbage collected each time. Column stack gives the maximum occupancy of the stack. Here we shall point out that the actual memory required in running each benchmark will never exceed the double of 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, they would represent the maximum allocated stack sizes if the CGC was disabled. It is worth to 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 the execution times in gc-Enabled and gc-Disabled
tests. The final column gives the ratios for gc-enabled and gc-disabled tests.

The results in Table 1 are very promising. For these benchmarks, their performance with CGC is better than, or at least as good as they are executed on a machine with infinite virtual memory. One significant 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. The second important reason is that the single stack paradigm incorporated with CGC improves program’s locality. This improvement greatly reduced overheads caused by the gaps between cache and main memory, and between main memory and the secondary virtual storage. In order to determine the extent to which the cache performance of the test programs has been improved, we developed the LVM cache emulator to do the trace-driven cache simulation.

3. CACHE SIMULATOR

A cache is a small and fast memory that locates between the processor and the main memory. There may be several levels of caching, only one level is studied in this paper. 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 which memory block is stored in each cache block. Apart from the tag, the directory contains control bits that keep status information on each cache block, such as if the block contains valid data (valid bit), or if its contents have been modified (dirty bit). A subblock is the smallest part of a cache with which a valid bit is associated. Cache blocks are grouped into sets. A memory block may reside in the cache in exactly one set, or may reside in any block within the set. A cache with sets of size \( n \) is said to be \( n \)-way associative. If \( n = 1 \), the cache is called direct-mapped.

A memory address may be divided into three parts, the tag, the index, and the block offset. The index is used to select the set, while the block offset selects a word within a block. When the program makes a memory read reference, this 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 cache. If the block is in the cache, the word is selected from this cache block by the offset segment. Otherwise, copy the missing block from the main memory to the cache. The way that write references are handle depends on the write policy. When writes hit, there are two write hit policies to choose: write-through and write-back. In a write-through cache, writes to the cache go to main memory immediately. In a write-back cache, writes to the cache are actually written to the cache and they eventually go to main memory when they are removed from the cache. Write-back caches can reduce the traffics between the cache and main memory. When writes miss, there are two main write miss policies: write-allocate vs. no-write-allocate and fetch-on-write vs. no-fetch-on-write. On a write-allocate cache, a block is allocated for the word being written, while on a no-write-allocate cache, the word is written only to memory and no cache block is allocated. If the block size is greater than one word and the policy is no-fetch-on-write, then the remaining words in the block must be invalidated. If fetch-on-write is used, the block containing the word being written should be fetched from main memory first. Then all the words in the block will be valid after write.

There are four classes cache by the different possible combinations of these write policies. In our work, only two of them are considered: Fetch-on-write and Write-validate. A cache is fetch-on-write cache if it uses a write-allocate and fetch-on-write policy. A cache is write-validate if it uses a write-allocate and no-fetch-on-write policy. Fetch-on-write and write-validate can be used with either write-through or write-back policy.

4. PERFORMANCE ANALYSIS

The same set of benchmarks has been tested under the LVM cache emulator. For each benchmark, we carried out different tests with respect to varying cache sizes, cache-limits, write policies, etc. We also collected data, such as memory references and read/write cache misses, from each test program and the CGC collector respectively. For the sake of space limit, it is impossible to present all the statistic results from a tremendous amount of information. Hence, we present our analysis based on these results and show a few typical results.

First, we tested benchmarks with different cache-limits. When cache-limits are larger than the corresponding cache sizes, the miss ratios of both user program and collector are high and eventually have a prominent plateau. Does this imply we should choose a cache-limit as small as possible? The answer is negative. If the cache-limit is too small, the collector will be activated more aggressively. This will increase garbage collection costs which consequentially degenerate the overall execution performance. From our experimental results, half to 2/3 of the size of data cache would be a proper range for selecting a cache-limit.

Table 2 shows statistics of memory references of all benchmarks, where the mutator represents user pro-
gram and the collector refers to the CGC algorithm, and all numbers are in millions. The conditions of these tests are 64K data cache and 40K cache-limit.

<table>
<thead>
<tr>
<th>Test</th>
<th>gc-disabled</th>
<th>gc-enabled</th>
<th>reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>tsp20</td>
<td>10.63</td>
<td>9.677</td>
<td>.929</td>
</tr>
<tr>
<td>dna20</td>
<td>5.847</td>
<td>5.407</td>
<td>.967</td>
</tr>
<tr>
<td>tsp22</td>
<td>16.76</td>
<td>16.76</td>
<td>.999</td>
</tr>
<tr>
<td>qsnv</td>
<td>1.072</td>
<td>1.072</td>
<td>.981</td>
</tr>
</tbody>
</table>

Table 2: Memory References

From Table 2 we found that benchmarks, except qsnv, only contribute a very small amount of memory references to garbage collection. These are approximately proportional to the garbage collection overhead. We also found that CGC reduces the number of memory references in some benchmarks. This is the 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 spend a big chunk of overhead on CGC. In fact, it is possibly one of the worst cases for the CGC algorithm: successive collections will repeatedly copy results which survive across all earlier collections. Does this mean that CGC is not suitable for such kind of programs? Our further investigation on cache-performance gives an promising answer.

<table>
<thead>
<tr>
<th>Test</th>
<th>gc-disabled</th>
<th>gc-enabled</th>
<th>reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>tsp20</td>
<td>347,702</td>
<td>27,511</td>
<td>0.35</td>
</tr>
<tr>
<td>dna20</td>
<td>220,831</td>
<td>16,940</td>
<td>0.772</td>
</tr>
<tr>
<td>tsp22</td>
<td>397,983</td>
<td>149,351</td>
<td>0.533</td>
</tr>
<tr>
<td>qsnv</td>
<td>120,372</td>
<td>2,091</td>
<td>0.981</td>
</tr>
</tbody>
</table>

Table 3: Cache Misses

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

We also compared different write policies on these benchmarks. We found that a write-validate cache has almost or more than enough to pay its own cost. However, this number was reduced to 3,609 (mutator + collector), a 97.2% reduction when it cooperated with the CGC algorithm. The simulation results from the cache emulator fully support the experimental results gathered from the LVM system: the cost of CGC could almost be paid by improved cache performance.

There are some constraints in our current cache simulation. First, only one level of caching is considered. A more sophisticated simulator with multiple level caching is left to our future work. Second, only data cache performance is studied. It is our another intention that not only data cache but also instruction cache will be studied in order to evaluate the CGC performance completely.

Finally, we would like to express our appreciation to the Natural Science and Engineering Council of Canada for supporting this research.

5. CONCLUSION AND FUTURE WORK

In this paper, we studied the cache performance influenced by the CGC algorithm. The simulation results from the cache emulator fully support the experimental results gathered from the LVM system: the cost of CGC could almost be paid by improved cache performance.

6. REFERENCES


