Cache-conscious Garbage Collection

Gernot Innthaler, Student, Johannes Kepler University, Linz

***Abstract* – The efficiency of nowadays computer systems is highly influenced by the performance of the system cache. With processor speeds getting higher and higher a good cache performance becomes even more important. This paper first gives a short overview of cache architectures. Furthermore it shows, how garbage collection can effect cache performance. This is supported by cache performance measurements of generational mark & sweep and generational copying garbage collection. These measurements consider different cache sizes, different placement strategies and different allocation thresholds.**

***Index Terms*-- allocation threshold, cache miss rate, cache performance, copying collection, garbage collection, generational garbage collection, incremental collection, mark & sweep collection**

1 Introduction

The usefulness of garbage collection techniques is out of question. The programmer is no longer responsible for freeing memory of unreferenced objects. It is not only, that the programmer doesn't have to think of freeing unused objects, implementing the memory management issues was usually a time consuming and fault-prone task. So garbage collection is comfortable for the developer and it improves the quality of the product as well.

But there are also disadvantages of garbage collection. Basically garbage collection interrupts the execution of the current task, so this slows the performance of the system down. Especially early garbage collection algorithms interacted badly with the memory system of computers. This is because the spatial and temporal locality of garbage collection algorithms is quite poor. The bad locality is explainable if we consider how a garbage collector works in principle. When there is no free space on the heap anymore the program halts and the garbage collector runs. To determine which objects are garbage the collector visits each object which is still in use. This traversal is made over the whole address space of the memory and usually leads to non-locality. This non-locality means poor memory performance.

Since modern processors get faster and faster it is very important that the data required for the CPU operations can be delivered very quickly to the processing unit. So the cache performance should be as good as possible. A good cache performance is obtained if the cache miss rate, the ratio between cache misses and the amount of executed instructions, is low. Several Studies have shown that the cache miss rate highly influences the overall system performance. For example Grunwald showed that lower miss rates increase the system performance by up to 25 percent [Grunwald93]. The results of Lam show a even more dramatic improvement of the system performance of about 400 percent [Lam91].

The cost for a cache miss depends on several aspects. First of all it depends on the architecture of the cache. Futhermore there is a difference between read and write misses. There is also to consider if only the first level cache is missed or the second level cache as well.

The actual processing time consists of the time where the processor can do useful work and the time where it has to wait for the cache. By assuming that the penalties for read and write misses have already been combined the CPU time *t* can be calculated as follows:

$$t = IC * (CPI + \frac{miss}{instruction} * misspenalty) * cycletime$$

IC ... instruction count
CPI ... average cycles per instruction

The increasing influence of the miss rate on the total performance on newer computers is shown on the following example.

The first calculation is based on an old VAX 11/780 (CISC processor) with a low miss penalty of 6 cycles, but with a high CPI average of 8.5. The miss rate should be 2 percent and there are 3 memory references per instruction:

$$t = IC * (8.5 + 3 * 0.02 * 6) * cycletime$$

$$t = IC * 8.86 * cycletime$$

The acutal CPI increased from 8.5 to 8.86 cycles, which is an increase of about 4 percent.

The second calculation is for a newer DEC Alpha AXP

(RISC processor) with a miss penalty of 50 and a CPI average of 2. The miss rate is again 2 percent, there are 1.33 memory references per instruction.

$$t = IC * (2 + 1.33 * 0.02 * 50) * cycletime$$

$$t = IC * 3.33 * cycletime$$

The actual CPI increased from 2 to 3.33 cycles which is an increase of about 66 percent.

2 Cache architecture

A cache is a memory storage either for data or instructions. Both types can be stored in the same cache. However , many hardware producers design their products with splitted caches, one in charge for data the other one in charge for instructions. Nowadays the cache is usually integrated in the CPU. Furthermore the cache is split in a level 1 and a level 2 cache. The level 1 cache the closest to the CPU and is smaller then the level 2 cache.

In order to be able to draw meaningful conclusions about the influence of garbage collection on cache performance caches should be distinguished by their cache size, their placement policy and their write stragety. The next sections will explain these terms.

2.1 Cache size

Depending on the actual implementation the cache sizes may vary between 8 kilobytes and 2 megabytes. As stated in a later section the cache size has a high impact on the cooperation of the garbage collector and the cache.

2.2 Placement policy

A cache is divided into a number of blocks. The size of a block is typically in a range of 4 to 128 bytes. There are coherences between the blocksize an the miss rate and between the block size and the penalty for a miss. Programs with a good spatial locality benefit from a larger blocksize, the miss rate will decrease. This is because it will be more likely that subsequent references will be to addresses in the same block. But on the other hand if the blocksize becomes to large in proportion to the size of the whole cache, the miss rate will rise again. In this case there are simply to few blocks in the cache available and so the blocks have to be replaced very often. An efficient blocksize is therefore a compromise of making the blocks large enough, so the program can gain efficiency and providing enough blocks in the cache.

Another question is how the blocks of the main memory are mapped into the cache. Therefore we can distinguish three block placement policies.

Direct mapped

In a direct mapped cache each block of the main memory is exactly mapped to a particular block in the cache. This can be achieved for example by calculating the main memory address modulo the number of blocks in the cache. Direct mapping caches are simple to build and the searching for a block can be done quickly. The drawback is, that there might easily arise conflicts of different main memory blocks mapped to the same block in the cache. If the data of these rivalling blocks has to be accessed alternatingly, there is always a miss and so the performance is going down.

Fully associative

In a fully associative cache the main memory blocks can be placed anywhere in the entire cache. In this way conflicts as described for a direct mapped cache can be avoided because there are no main memory blocks which have to map to the same cache block. But the fully associative cache has disadvantages as well. Searching the cache for a particular block would be either rather slow or require expensive parallel hardware. Therefore fully associative memory is more likely to be used for smaller units.

There are several strategies how a block can actually be placed in a fully associative cache. The nowadays commonly used technique is by randomly picking a cache block and placing the data there. Another possibility would be using the Least-Recently-Used (LRU) procedure. With this technique the least recently used block would be replaced. This would be the most efficient strategy but it is hard to implement. There are some strategies which nearly implement this behaviour. A further placing strategy would be First-In-First-Out (FIFO). This means, the first block written into the cache is the first one which is replaced. This technique is not recommended because it leads to a poor cache performance.

Set-associative

Set-associativity combines the concepts of direct mapping and fully associative. The cache is splitted into several sets. Common numbers of sets would be two or four. A main memory block is now always mapped into the same set. This could be done like choosing the right block with a direct mapped cache. Within this set the block can be placed anywhere. Searching for a particular block is now quicker than with a fully associative cache. We already know, that a block can only be in a particular set. So we only have to search within this set to determine if the block is already in the cache or not. For placing a block within a set the same strategies as for the

fully associative cache can be used, e.g. random, least-recently-used or first-in-first-out.

Both the direct mapped and the fully associated cache can be seen as special cases of a set-associative cache. A direct mapped cache is a set-associative cache with as many sets as there are blocks in the cache. On the other hand a fully associative cache would be a set-associative cache with only a single set.

2.3 Writing strategy

The performance of garbage collection algorithms seems to depend on the way how write misses are treated within the cache. Therefore I give a brief overview of writing strategies.

At first I want to consider the case that a write hit occurs. There are two possibilities what to do.

Write-through

In this case the data is written in the cache and in the main memory. If there are several cache levels the data is written in these levels as well. The advantage of this is that if the block is going to be replaced the data is already written into the next level memory. So there is no additional effort necessary to save the data. But if the data in a block is changed several times without the block being replaced in the meantime, I still have to write the data to the main memory each time it is changed. This is a bit of a waste of bus bandwidth.

Write-back

Using the write-back (also called copy back) strategy the data is only modified in the cache. It is only written back to the main memory if the block is going to be replaced. There is also an improvement for this strategy. If a block is going to be replace, but the data has not been changed since it was loaded, there is no need to write it back to main memory. Therefore each block has a so called dirty bit. This bit indicates wether the data in the block has been changed since it was loaded. So it is possible to decide, if the data has to be written back into the main memory or not. The advantages and disadvantages are the opposite of the write-through technique. Write-back saves bus bandwidth but if a block has to be replaced there are additional efforts necessary to save the data.

There are also two different techniques if a write miss occurs. The question hereby is wether the block should be loaded into the cache.

Write-allocate

With write-allocate (or fetch-on-write as it is called as well) the block is allocated into the cache. Now it can be handled as it had been there right from the beginning, so the procedure would be like for a write hit.

Write-no-allocate

With write-no-allocate (or write-around) the block is not loaded into the cache. The data is going to be written in the next level memory (either the next cache level or the main memory) right away. This saves bus bandwith because the transfer of the block into the cache is missing.

Each cache has to combine a technique for write hit and write miss. Usually write-through is combined with write-no-allocate and write-back is combined with write-allocate. Write-through writes into the main memory anyway so therefore it is better write into the main memory with write-no-allocate in first place. The combination of write-back and write-allocate makes sense as well. With write-allocate the block is placed into the cache. Then we hope, that there are write operations on the block before it is replaced. So it was cheaper to put the block into the cache.

3 Memory access of garbage collectors

The cache architecture we have seen so far benefits from high spatial and temporal locality of programs. This assumption can be made for most programs. Garbage collector however basically have bad spatial and temporal locality. In order to determine if garbage collectors are able to cooperate with caches in a good way this section discusses the general memory access patterns of various garbage collection strategies.

3.1 Mark & Sweep garbage collector

Considering a simple mark & sweep collector the references to the heap are likely to be random reads and writes. Using a more advanced mark & sweep implementation make the prediction of the memory access easier. Those advanced methods could be for example collectors using mark bitmaps and lazy sweeping or generational mark & sweep collectors.

During marking the branch points are stored in a stack and a bitmap is used to mark the objects. The pointers are traced by accessing heap data. The read and write references on the stack show high locality. The references in the bitmap should show quite a good spatial locality, too. By tracing the graph random read-only accesses to the heap are created. Generational mark & sweep collectors restrict the range of references to a certain extent to limited region of the heap. These observations imply that a mark & sweep garbage collector has random reads and highly localised writes. [Jones96]

The sweep phase produces highly sequential reads and writes to mark bits. If first-fit allocation with separate free-lists for each common object size is used the allocation pattern consists of sequential, initialising writes.

3.2 Copying garbage collector

A copying garbage collector basically compacts the heap, if it is a generational collector it compacts the regions of the heap. This means, the allocation is strictly linear, objects directly allocated after each other are directly place next to each other on the heap.

At first a copying garbage collecter has to scan the objects in the grey part of the Tospace. Each word referenced by a pointer is read and then updated. So this memory access pattern consists of sequential reads and writes. The forwarding address of the object where the pointer refers to, is also read. In case the Fromspace object has not been copied it has to be copied to the Tospace and an update of the forwarding address is necessary. This behaviour results in reads to a random location with a possible write and after that sequential reads in Fromspace and sequential writes in Tospace. References to Tospace are around the address, where *scan* points to. This is for read and write. Furthermore there are references to addresses pointed to by *free*. These are writes. After a word has been marked as black, which means it was scanned and updated, the collector does not have to touch it in this cycle again. This description of memory access patterns is a bit simplified since some issues, like copying objects by remapping virtual memory, are not considered. [Jones96]

If linear allocation is assumed, it results in an easy predictable pattern of access to Tospace. This would be a sequence of initialising writes. It is not uncommon for systems using garbage collection to have very high rates of allocation [Jones96]. Generally it is assumed that compared to reads writes occur only rarely. Typically writes occur for about ten percent of all executed instructions. But Diwan *et al.* and Goncalves and Appel found programs where number of writes rise up to about 25 percent of all executed instructions [Diwan94] [Goncalves95]. The majority of these writes occured due to allocation. Since a copying garbage collector uses two semispaces, there is a “back-and-forth“ allocation pattern to observe. It is likely that this pattern works against the block replacement policy of the cache. This leads to the conclusion, that linear allocation can be expensive in cycles, although it is quite cheap if we only consider the amount of instructions executed by the allocator. [Jones96]

3.3 Incremental garbage collector

Incremental garbage collectors access the memory alternating with the mutator and the collector within a short time. This could lead to the conclusion that the cache miss rate is going up. An explanation would be that data accessed by the mutator and data accessed by the collecter are mapped to the same cache block. This results in a high number of necessary replacements. But Zorn delivers some evidence that the performance of incremental garbage collection might be better than as stated in the assumption [Zorn89].

A significant factor for the cache performance of incremental collectors will be the style and the granularity. Read-barriers trap the mutator access. So the collector works with data the mutator is using at that particular time. Some incremental collectors use the memory protection system of the operating system. In this case, the granularity of the read-barrier is a page and therefore the behaviour explained above is weakened. There are write-barriers which do not need the support of the virtual memory. They mark data subjects to the mutator writes and therefore behave in a similar. Dijkstra and Steele for example developed collectors with that behaviour. A design goal for some incremental collectors is to improve locality. Therefore they are clustering objects in Tospace according to their references in the mutator.

It seems, that there are no studies made yet, that discuss the cache performance of incremental garbage collection. Therefore, this paper can not answer the question, if the stated assumptions are correct. [Jones96]

4 Improving the cache performance

Generally speaking the cache performance can be improved in three ways:

- by reducing the cache miss rate
- by reducing the miss penalty
- by reducing the time to hit

In context to garbage collection reducing the cache miss rate is the import point. There are some know techniques to reduce the miss rate, for example increasing the size of the cache, increasing the size of a block in the cache or increasing the associativity.

On the basis of studies by Zorn [Zorn91] and Goncalves and Appel [Goncalves95] the influence of these parameters in context to garbage collection is shown. Goncalves and Appels study is based on a generational copying collector. Zorn used a mark & sweep collector as well as a copying collector. Both algorithms use generational garbage collecting techniques and to get comparable results the algorithms are as similar as possible. The cache block size is 32 bytes. As a trigger to run the garbage collector Zorn used the *allocation threshold* technique. This means the collector is started after a fix amount of memory has been allocated. In

contrast a *fixed-space* technique would wait until the whole memory is allocated and then run the garbage collector. Figures 1 and 2 illustrate both concepts.

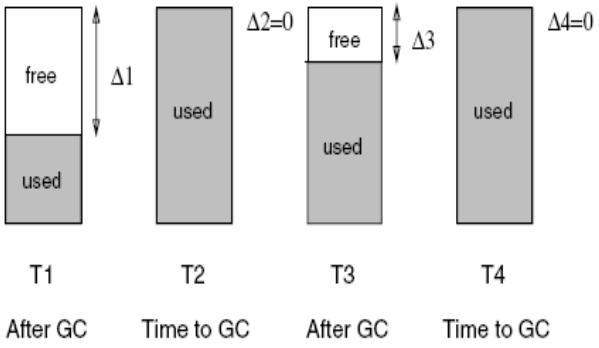

Fig. 1 Fixed space collection policy

The amount of allocated memory is different between each garbage collector runs. The disadvantage of the fixe-size concept is, that it can lead to thrashing. This happens when most of the memory in newspace is allocated to live objects. Newspace fills quickly and so the garbage collector is invoked more often but it can recover less garbage each time.

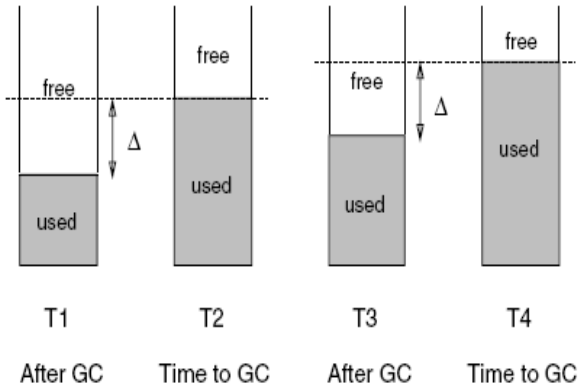

Fig. 1 Allocation threshold collection policy

The amount of allocated memory is the same between to garbage collector runs. The allocation threshold method has several advantages. On one hand it solves the thrashing problem of a fixed-size collection. On the other hand the allocation behaviour is independent of the used garbage collection algorithm. This means, each collector is invoked the same number of times. This makes it easier to compare different garbage collection algorithms.

The garbage collection performance is highly influence by the allocation threshold. Making the threshold smaller, the garbage collector is invoked more often. This has both negative and positive effects. If the collecter runs more often, fewer objects can become garbage between two collector runs. Also the CPU time for the collection increases. If the generational behaviour is implemented in that way, that an object is promoted to the next generation after a fixed amount of collector runs, then a small allocation threshold increases the promotion to older generations as well. An advantage of a small threshold is the good spatial reference locality since garbage objects are reused quickly.

4.1 Mark & Sweep garbage collector

At first I want to analyse the performance of the collector with a direct mapped cache architecture.

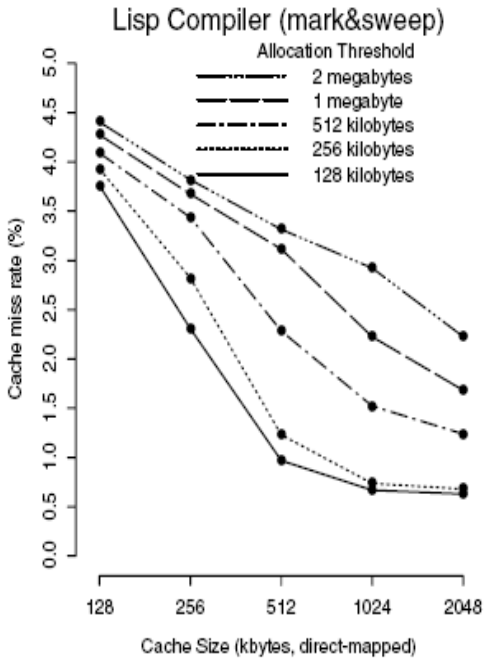

Fig. 3 Cache miss rate with mark & sweep collector, direct mapped

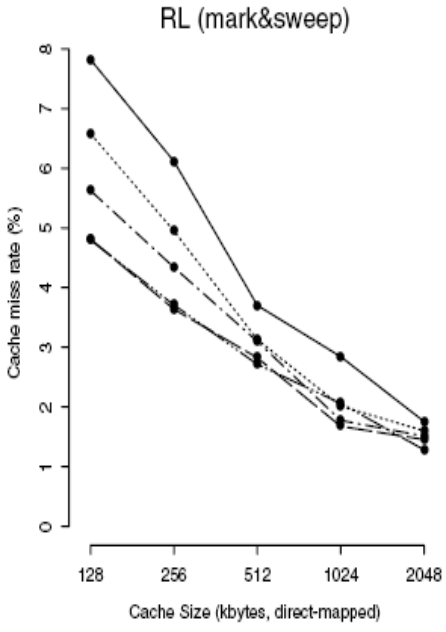

Fig. 4 Cache miss rate with mark & sweep collector, direct mapped

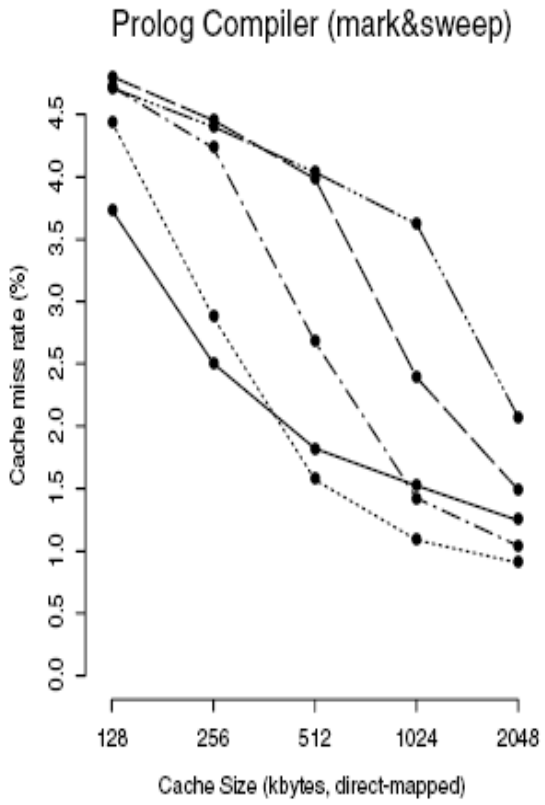

Fig. 5 Cache miss rate with mark & sweep collector, direct mapped

Figures 3 to 5 show the cache miss rate for different cache sizes of a mark & sweep garbage collector on a direct mapped cache. As we can see, the cache miss rate is highly influenced by the cache size. Generally can be said that increasing the cache size reduces the miss rate. The diagrams show, that garbage collecting programs "fit" better in larger caches. Doubling the cache size until the "well fit" point is reached effects the miss rate to decrease by about one percent. After that the curves are a bit flatter.

The allocation threshold also can have a major impact on the cache miss rate. If we take, for example, the Lisp compiler running at a cache with 512 kilobytes, the miss rate for an allocation threshold of 2 megabytes is more than three times higher than with a threshold of 128 kilobytes. The Prolog compiler shows similar results. The miss rates for the RL compiler are not that much influenced by different thresholds at a certain cache size than for the other two compilers. But the tendency is still there.

Another interesting observeration is, that a smaller threshold does not necessarily perform better. If we take the Lisp compiler, a threshold of 128 kilobytes shows the lowest miss rates. On the other hand running the RL compiler it turns out to have the highest miss rates of all thresholds. Therefore the best threshold also depends on the running program.

There also seems to be a relation between the cache size and the threshold. In nearly every curve we can see a knee were both the cache size and the allocation threshold have the right size, so that the program can "fit" well.

The next figures show how the associativity influences the cache miss rate. The threshold is set to 128 kilobytes.

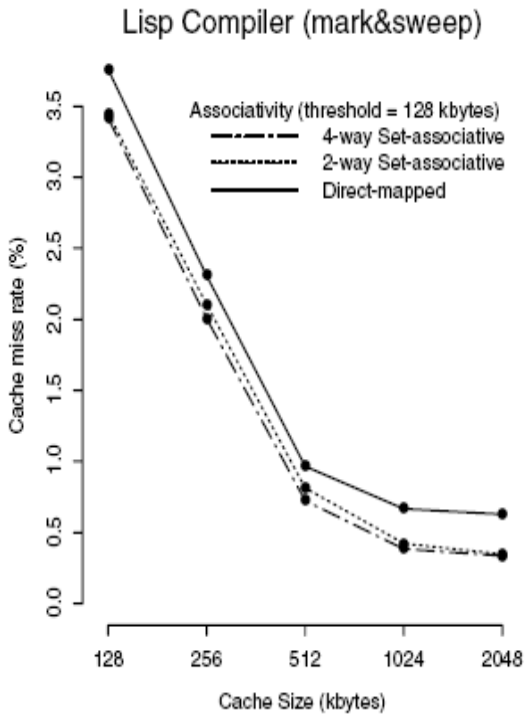

Fig. 6 Cache miss rate with mark & sweep, different associativity

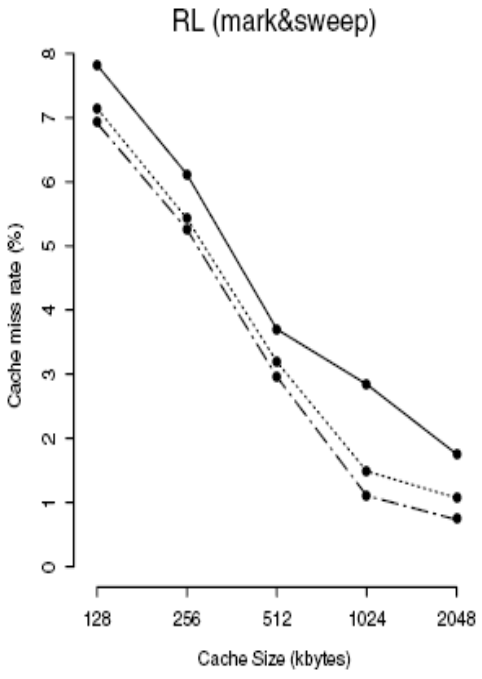

Fig. 7 Cache miss rate with mark & sweep, different associativity

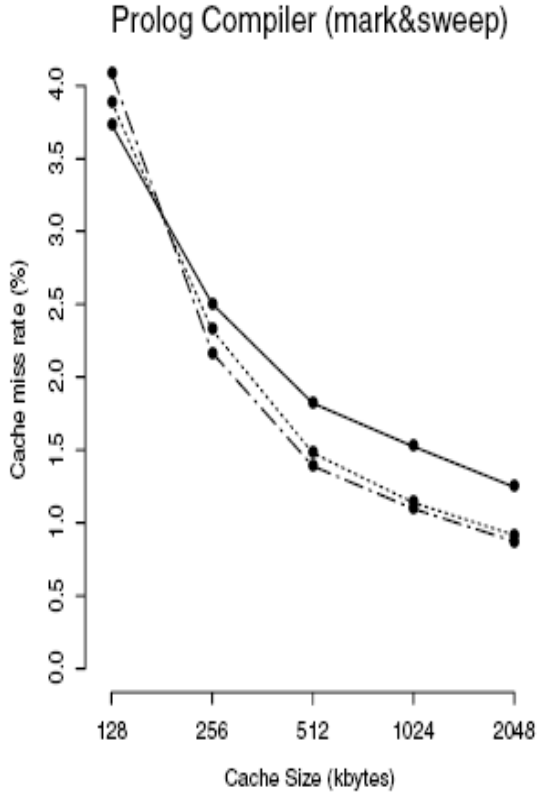

Fig. 8 Cache miss rate with mark & sweep, different associativity

Figures 6 to 8 show that the influence of the cache associativity on mark & sweep collection is not very high. Generally set-associative caches perform a bit better than a direct mapped one. But even to this there is a small exception. Running the Prolog compiler with a cache of size 128 kilobytes the direct mapped cache has lower miss rates than the set associative ones.

The different miss rates between the 2-way-set-associative and the 4-way-set-associative caches are negligible.

Since mark & sweep collectors do not copy objects, the only exception is to promote them, the benefit of using set-associative cache architectures is only little. The allocated objects in newspace stay in the same place until they are promoted to the next generation. Assuming a large enough cache size so that it can contain the objects in newspace, only few collisions will arise by referencing this generation. Collisions can only arise between references to newspace and older generations. The older generations are much larger than newspace and therefore cache blocks are accessed more randomly. This avoids repeated, systematic collisions.

4.2 Copying collector

The same measurements as for the mark & sweep garbage collector were done with the copying collector as well. I start with the analysis for the direct mapped architecture again.

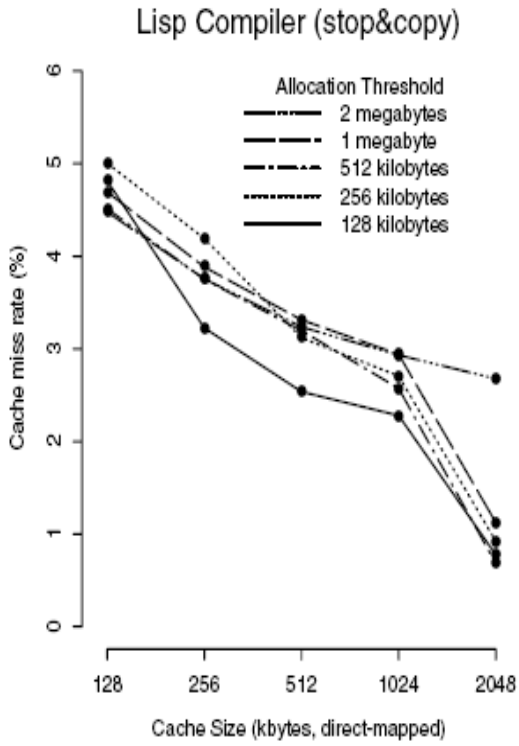

Fig. 9 Cache miss rate with copying collector, direct mapped

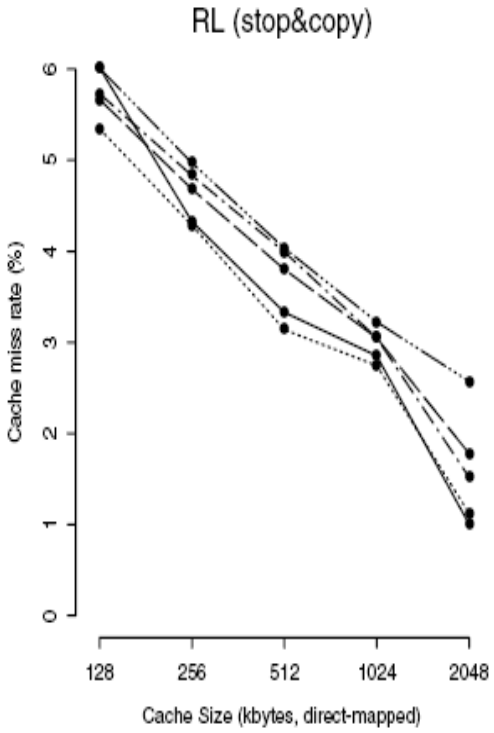

Fig. 10 Cache miss rate with copying collector, direct mapped

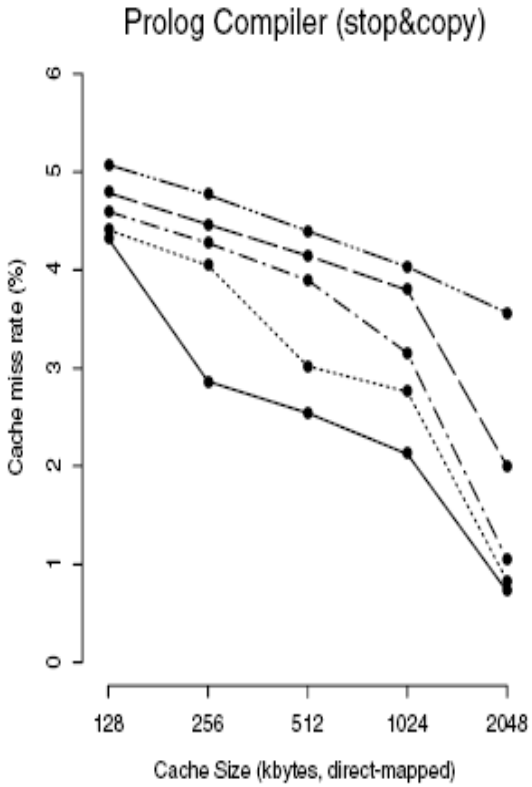

Fig. 11 Cache miss rate with copying collector, direct mapped

Figures 9 to 11 show cache miss rates for a direct mapped cache running with a copying collector. As observed for the mark & sweep collector, larger caches result in a lower miss rate.

The size of the threshold has still some influence on the cache performance, but the differences are most of the time not as high as for the other garbage collector. A lower allocation threshold seems to result in a better performance. But now there are not only exceptions to that with the RL compiler, the Lisp compiler combined with a cache size of 128 kilobytes also shows that the smallest threshold is not necessarily the best.

Generally can be said, that the miss rates are higher than with using a mark & sweep garbage collector

The following figures show the influence of the associativity on the miss rate for the copying collector. The allocation threshold is again 128 kilobytes.

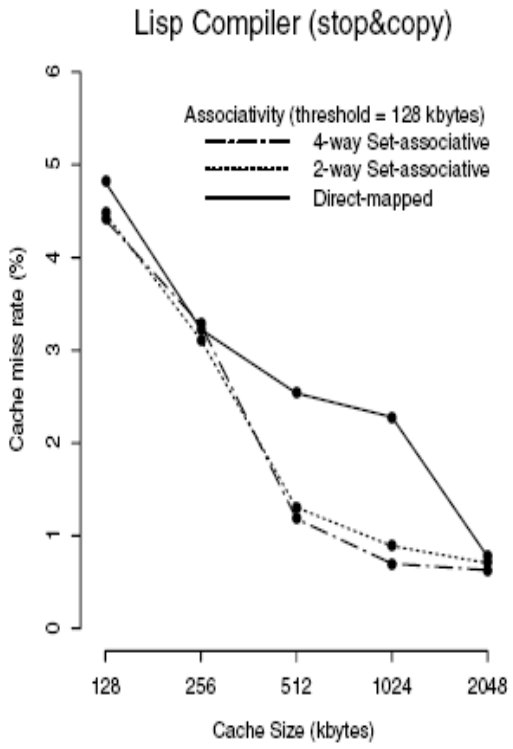

Fig. 12 Cache miss rate with copying collector, different associativity

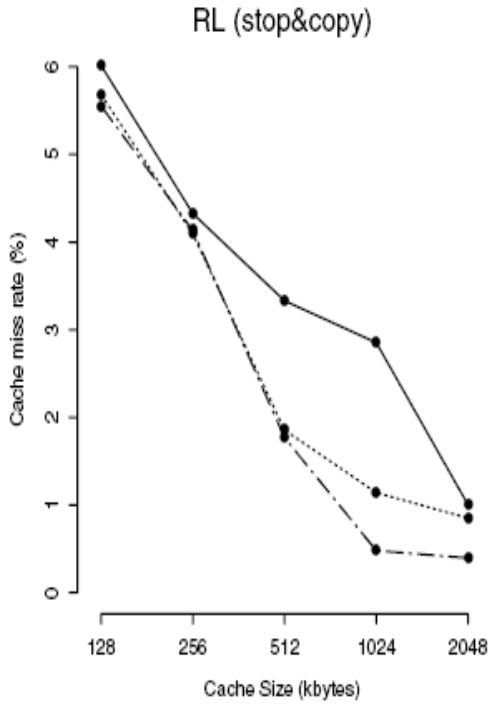

Fig. 13 Cache miss rate with copying collector, different associativity

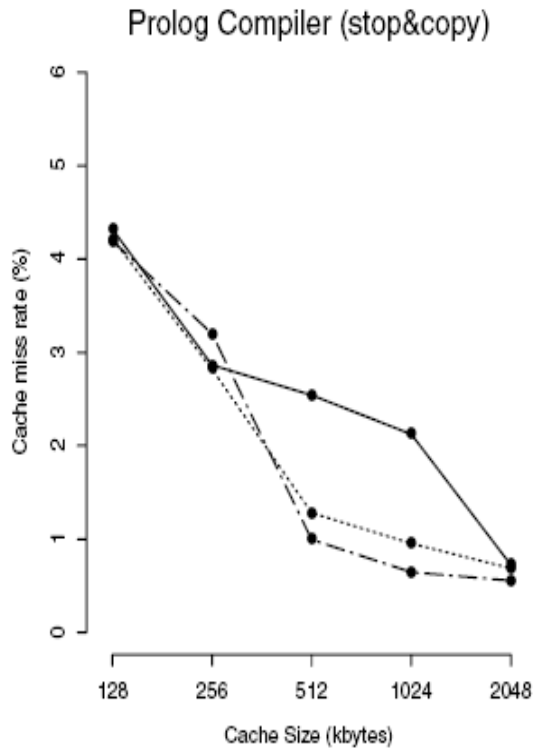

Fig. 14 Cache miss rate with copying collector, different associativity

In contrast to mark & sweep collection algorithms figures 12 to 14 show that copying collectors benefit from a set-associative architecture a lot if the cache is large enough. If this is teh case, the cache can hold both semispaces and the amount of collisions decreases. To explain the benefit over the direct mapped cache it has to be considered, that since the cache uses an allocation threshold the two semispaces do not always have the same size. It depends on how many objects live through a collection. If the semispaces become as large, so that they do not fit in the cache anymore, then many collisions arise in the direct mapped cache. In the set-associative cache the blocks can be placed freely to a certain extent. So most of the collisions are prevented.

There is another interesting aspect to observe. If the cache size is large enough so that it can hold both semispaces with their largest possible dimensions, the performance of the direct mapped cache is again nearly as good as the performance of the set-associative versions.

The cache miss rates of the 4-way-set-associative cache are a bit lower than the miss rates of the 2-way-set-associative cache, but they are nearly negligible.

Goncalves and Appel ovbserved in their study the influence of the cache block size on the miss rate. They also differentiated between read and write misses. The cache is direct mapped with write-allocate.

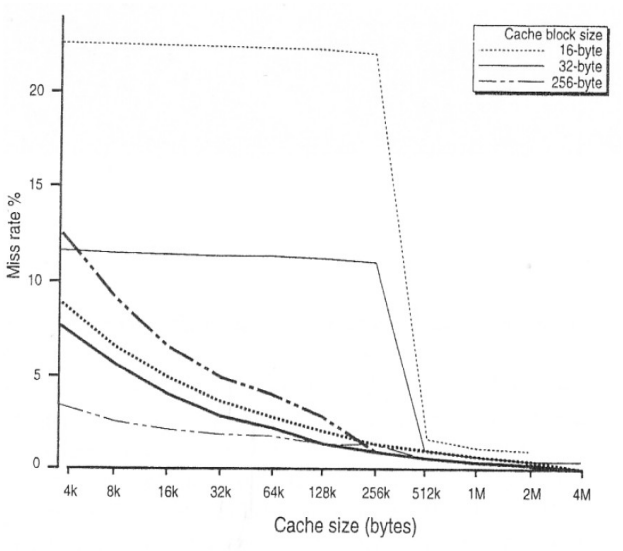

Fig. 15 Cache miss rate in contrast to different cache block sizes

Figure 15 shows the cache miss rate in context to the cache size for different cache block sizes. Read misses are drawn with heavier graphs.

As describe in section 3.2 copying collection lineary updates pointers in the grey region. Allocation and evacuation are done lineary as well. This leads to the expection, that increasing the block size would decrease the miss rates. The diagram shows that this assumption is right. Especially the probability of write misses (thinner lines in the diagram) can be reduced dramatically. By doubling the size of a cache block from 16 bytes to 32 bytes the miss rate could be reduced over 10 percent.

Since mark & sweep collectors also benefit from spatial locality, similar results can be expected.

4.3 Comparison of Mark & Sweep and Copying collectors

This section directly compares the cache miss rates of mark & sweep and copying garbage collectors.

The allocation threshold is set to a size of 256 kilobytes. Figures 16 and 17 show the comparision of the two collectors using a direct mapped cache. Figures 18 and 19 show the comparison in context of a 2-way-set-associative cache.

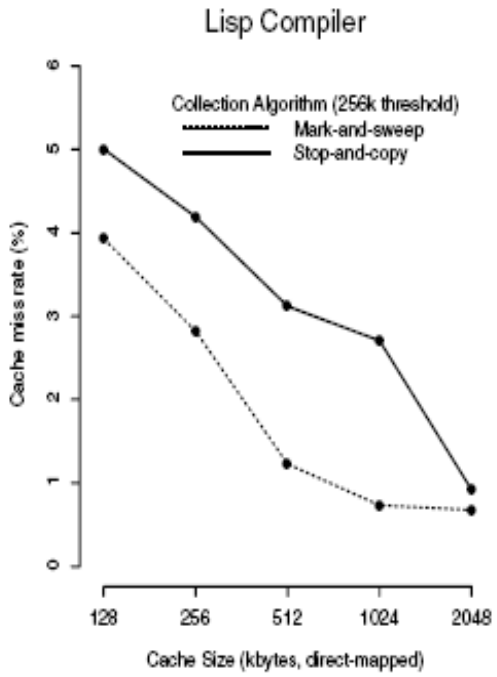

Fig. 16 Cache miss rates of mark & sweep and copying garbage collectors, direct mapped cache

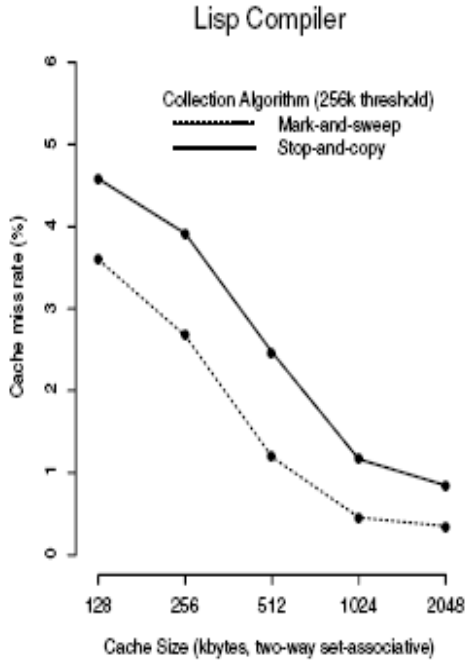

Fig. 18 Cache miss rates of mark & sweep and copying garbage collectors, 2-way-set-associative cache

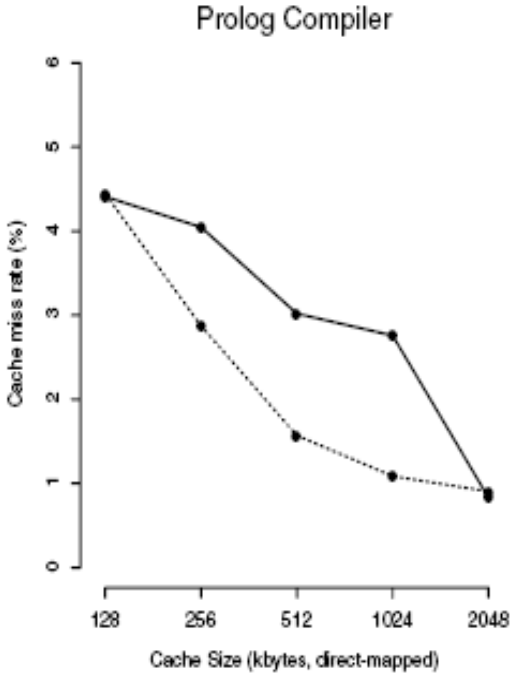

Fig. 17 Cache miss rates of mark & sweep and copying garbage collectors, direct mapped cache

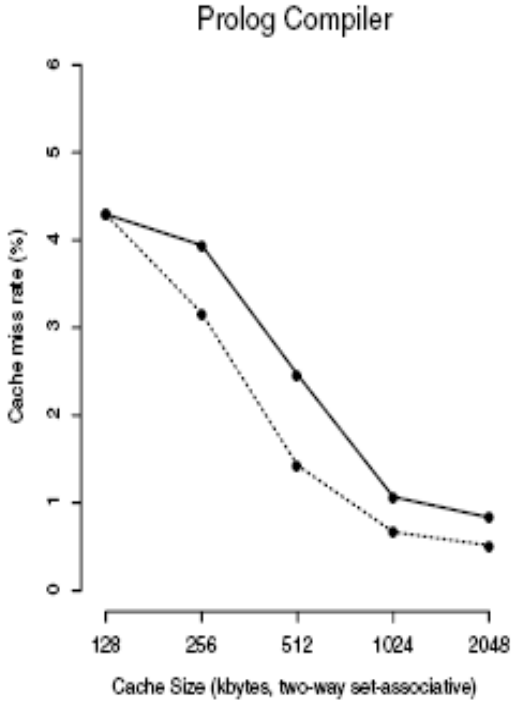

Fig. 19 Cache miss rates of mark & sweep and copying garbage collectors, 2-way-set-associative cache

With both cache architectures the mark & sweep algorithm has a better cache performance. This is for one reason for the better locality of a mark & sweep garbage collector. In addition it also benefits more from adjusting the allocation threshold. Remarkably there is a major difference in the cache miss rates for midsize caches

whereas the gap between the two algorithms for small and large caches is much smaller or not existing at all. Because of the high promotion rate for smaller caches the mark & sweep implementation looses some of its benefit. In large caches the programs can fit well with both techniques and so the miss rates are low anyway. The gap between the two collectors is small with a set-associative cache. This is because the copying collector benefits from the 2-way-set-associative architecture whereas mark & and copy shows hardly any improvement.

5 Conclusion

The paper discusses the influence of garbage collectors on the cache performance. The common opinion that garbage collectors have a poor cache performance is not necessarily true. By changing the cache size, the size of a cache block or by using a set-associative cache architecture the performance can be improved dramatically. The cache size should be large enough so that the program can fit well. An increased block size decreases the cache miss rate as well (as long as the amount of blocks in the cache is still large enough). Generally mark & sweep garbage collectors have better cache performance. This is due to their working principle. Furthermore they react better to cache tuning measures. Copying collectors benefit from a set-associative architecture.

6 Bibliography

Books

[Jones96] R. Jones, “Garbage Collection: algorithms for automatic dynamic memory management”, Wiley & Sons, 1996, pp 277 - 298

Papers

[Chilimbi98] T.M. Chilimbi, J.R. Larus, "Using generational garbage collection to implement cache-conscious data placement", *First International Symposium on Memory Management* , October 1998

[Diwan94] A. Diwan, D. Tarditi, J.E.B. Moss, "Memory subsystem performance of programs using copying garbage collection", *Annual ACM Symposium on Principles of Programming Languages* , January 1994

[Goncalves95] J.R. Goncalves, A.W. Appel, "Cache performance of fast-allocating programs", *Conference on Functional Programming and Computer Architecture* , June 1995

[Grunwald93] D. Grunwald, B. Zorn, R. Hendersson "Improving the cache locality of memory allocation ", *Conference on Programming Language Design and Implementation* , June 1993

[Lam91] M.S. Lam, “The cache performance and optimizations of blocked algorithms“, *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* , April 1991

[Wilson92] P.R. Wilson, M.S. Lam, T.G. Moher "Cache considerations for generational garbage collection", *ACM Symposium on Lisp and Functional Programming* , June 1992

Technical Reports

[Zorn91] B. Zorn, "The effect of garbage collection on cache performance", Technical Report CU-CS-528-91, University of Colorado at Boulder, May 1991

Dissertations

[Zorn89] B. Zorn, “Comparative Performance Evaluation of Garbage Collection Algorithms“. PhD thesis, University of California at Berkeley, March 1989. Technical Report UCB/CSD 89/544