Online ISSN 1848-3380, Print ISSN 0005-1144
ATKAFF 55(4), 514–525(2014)

Danko Basch, Dorian Ivančić, Nikica Hlupić

Mark without much Sweep Algorithm for Garbage Collection

DOI 10.7305/automatika.2014.11.857
UDK 004.33; 621.377.6
IFAC 4.5.3; 5.0

Original scientific paper

In this paper two simple improvements over traditional mark-sweep collector are proposed. The core idea is placing small objects of the same type in buckets. The buckets are organised in such way to eliminate the internal fragmentation, sweeping, and freeing inside them. The measured improvement of garbage collection time over traditional mark-sweep is 19%. Another proposed improvement is more general and is applicable to other garbage collection algorithms as well. It uses heuristics to control the heap growth. The regularities in behaviour of objects of particular types are used to determine whether the collection should be performed or avoided in favour of immediate heap expansion. The heap expansion algorithm reduces garbage collection time over traditional mark-sweep for 49% while keeping the heap size approximately the same.

Key words: Mark-sweep, Garbage collection, Heap sizing, Memory management

Algoritam za zbrinjavanje memorije s označavanjem i smanjenim oslobađanjem. U ovom članku opisana su dva jednostavna poboljšanja algoritma označi-oslobodi. Osnovna ideja jest smještanje malih objekata istog tipa u pretince. Pretinci su organizirani tako da se u njima ne pojavljuje unutarnja fragmentacija, a uklanja se i potreba za oslobađanjem blokova zauzetih nedohvatljivim objektima. Vrijeme provedeno u zbrinjavanju manje je za 19% u odnosu na klasični algoritam označi-oslobodi. Drugo poboljšanje je općenitije i moguće ga je primijeniti i na druge algoritme za zbrinjavanje memorije. U njemu rastom gomile upravlja heuristički algoritam koji koristi pravilnosti u ponašanju objekata različitih tipova. Na temelju njih, algoritam odlučuje hoće li gomila biti zbrinuta ili odmah proširena. Heuristička inačica algoritma smanjuje vrijeme provedeno u zbrinjavanju u odnosu na tradicionalni algoritam označi-oslobodi za 49%, a da pri tome zahtijeva približno istu količinu memorije.

Ključne riječi: označi-oslobodi, zbrinjavanje memorije, dimenzioniranje gomile, upravljanje memorijom

1 INTRODUCTION

The basic tracing algorithms for garbage collection (GC) are mark-sweep, copying, and mark-compact. They visit live objects by tracing pointers in the heap, starting from a set of root-pointers. While tracing pointers, mark-sweep marks all live objects and in the second phase it sweeps the heap sequentially freeing the unmarked objects. Mark-sweep is non-moving algorithm so free and used memory blocks are scattered throughout the heap. Like any other GC algorithm, mark-sweep (MS) is not ideal, yet it is still widely used in some of its numerous variants, either as a standalone collector or more often as a part of advanced algorithms like generational, conservative, or incremental collectors (e.g. in JRockit, Dalvik, and Hotspot).

In this work the aim is to investigate some new ideas for improvements of MS in order to speed up execution but without increasing memory consumption too much.

All the proposed changes are very simple and they do not require any modifications in operating system (OS) or run-time environment, other than changes in memory management, of course. Also, the proposed algorithm does not rely on any kind of static analysis and/or profiling of programs, neither does it rely on any user-defined parameters. The only assumption is that the GC is precise rather than conservative (conservative marking traces everything that looks like a pointer so it can retain some dead memory objects in a heap). We feel that simple changes have more chances to be accepted in practice than complex mechanisms regardless of good results they may achieve.

In the proposed modification of MS we classify objects as large and small, depending on a threshold size. Small objects are allocated in memory areas named buckets. Large objects encompass the buckets themselves, arrays of any size, and all other objects larger than the threshold. The large objects are allocated by using any standard allocation algorithm. The buckets are sized to contain 32 small objects of the same type or class. This completely eliminates the internal fragmentation for small objects (ex-

ternal fragmentation is addressed later in the paper). The buckets are organised so that small objects never need to be swept and therefore we named the proposed algorithm "Mark-Without-Much-Sweep" (MwmS). Another benefit of our bucket organization is entirely eliminated overhead of freeing small objects. Also, the allocation of small objects is simplified and, consequently, made faster. With the proposed changes the GC time (cumulative time spent by garbage collector during the program run) is on average reduced by 19% over traditional MS.

GC increases the programmer's productivity but decreases performance of applications in comparison to explicit memory management. GC's impact on performance highly depends on a heap size and therefore we pay attention to heap expansion as well. Increasing the heap is not the right answer (at least not always) in reducing performance impact of GC. For example, if the heap becomes larger than the available physical memory, paging will occur and the execution will slow down. In embedded and mobile systems the memory resources are limited and swap space usually does not exist so the heap cannot always grow. Besides, heap that is too large can create memory pressure that can slow down other applications. Motivated by all these difficulties, we propose a simple heuristic algorithm for heap expansion. The algorithm is balanced in order to improve the execution time while keeping the heap as small as possible. In short, if a program creates long-lived objects, the heap is expanded immediately rather than doing a collection after which the heap will be expanded anyway. On the other hand, if created objects are short-lived, the GC is preferred. The MwmS with the heuristic heap expansion reduces GC time by 37% in comparison to the basic MwmS. In relation to traditional MS, the speedups are even better (49%) and the heap size remains approximately the same.

Two main contributions of this paper are: (1) novel organisation of small objects in buckets with integrated bitmaps, which improves allocation time and internal fragmentation, and avoids the sweep phase and freeing inside buckets; (2) simple yet effective heap expansion algorithm that adapts to the application behavior in order to reduce the execution time without too much heap expansion.

2 BACKGROUND

Allocation. GC can use a general purpose allocator built in the system libraries. In that case GC and allocator have separate headers. The allocator headers typically contain block size and used/free bits, while the GC headers may contain a mark-bit, counter, or pointer etc. – depending on the GC algorithm. MS needs only one mark-bit which can usually be placed into the allocator header.

Allocators may use bitmaps where each bit is mapped to several bytes in the heap. Each bit denotes whether its associated bytes are free or used. On allocation request the allocator scans the bitmaps in search for sufficiently long sequence of zeroes. To free the block, all its related bits should be cleared so the block size has to be known [1, 2].

Two-level allocators alleviate the fragmentation problems and are therefore often used with MS algorithms. At the lower level the allocation is performed in blocks. At the higher level small objects (of the same size or type) are allocated inside the blocks [1–5].

Moving algorithms have a contiguous free space and they use bump allocation which is simple and fast (blocks are allocated simply by moving a so called bump pointer which points to the beginning of the free space). MS usually requires an underlying allocator based on free lists which is slower than bump allocation. The allocator causes internal and external fragmentation and needs space for headers. However, MS needs less memory than copying algorithms although more than compacting algorithms. By using state-of-the-art allocator, the fragmentation can be reduced and allocation speed can be increased [6].

Mark phase. The simplest variant of mark phase is recursive marking. Its numerous function calls produce overhead and may cause system stack overflow. Explicit marking-stack is often used since its overflow is handled much easier and it avoids the function calls [1, 4, 7, 8].

Mark-bits organization. A mark-bit can be placed inside an object header or in a separate part of memory in bitmaps (not to be confused with an allocator's bitmap). Bitmaps are typically used for conservative GC which can falsely identify a part of memory as a live object but must not change it. If regions of memory hold the same sized objects, each bit can denote an entire object, which simplifies bitmap operations and reduces space overhead. When objects of different sizes are held together in a heap, each bit in a bitmap denotes each possible starting location of an object [2].

Accessing bits in bitmaps needs mapping (determination of bit position according to the object address) which is slower than accessing header bit. Marking in bitmaps has better locality than using header bits. This can compensate mapping duration and can make bitmaps faster. However, when the whole tracing loop is taken into account, together with object scanning, the performances of header bits and bitmaps are very similar [9].

Sweep phase. Collection pauses can be reduced with lazy sweep (also called mark-and-don't sweep). Lazy sweep is performed on allocation requests but incrementally – only few objects are swept, just enough to complete

the allocation. Alternatively, a certain amount of the heap can be swept. Lazy sweep can also reduce the overall GC time due to less coalescing of free objects. It also improves locality because the swept area is small and will probably be immediately used by an allocator and mutator (user program is called mutator because it mutates the connections between objects) [2, 4, 10].

Heap sizing. In small heaps GCs are frequent which slows an execution. A total execution can be faster in larger heaps where GCs are less frequent (although each of them may last longer). If a heap outgrows the available memory, paging will occur which will degrade performance significantly. Heap size optimization is hard because it depends on many dynamic factors. The heap can be reduced when using moving collectors, since their free space is contiguous and the unused end of the heap can be returned to an OS. The heap cannot be reduced when using non-moving collectors, like MS, because their heap contains scattered live objects. A virtual memory manager may swap out nearly empty and rarely used pages, but only until the next collection when they will be touched again [11]. Therefore, for non-moving collectors the heap should grow more sparingly than for the moving collectors.

The heap sizing algorithms can use various data when deciding whether to resize the heap. These data can include total memory size, heap size, GC efficiency, current memory footprint, real memory usage, memory pressure, empirically collected data about program behaviour, etc. Some of the algorithms rely on empirically tuned values [12, 13] or on data collected by running benchmarks [11, 14]. Other algorithms require changes in VM or OS or both [15]. Some approaches do not require user participation or OS modifications but only dynamically loadable kernel module [16].

3 THE IDEA OF MARK-WITHOUT-SWEEP

Here we present a simplified idea of MS algorithm improvements. We propose the integration of two-level bitmap allocator and bitmap based MS collector, where bitmaps are the key integration point. In MS, mark-bits are set during marking phase for all reachable objects and used during sweep phase to distinguish between live objects and garbage. In bitmap allocator bits in bitmaps are used to distinguish between used and free memory blocks.

To understand the possibility for integration, let us imagine an environment which uses built-in bitmap allocator and naïve MS collector. Now suppose that all objects are of exactly the same size, hence all allocator blocks will also be of equal size. Each block will hold exactly one object. In this allocator 1:1 mapping between bits in bitmap and memory blocks is possible – each bit denotes exactly one memory block and hence exactly one object. In this simplified environment one set of bits can be used for both allocation and garbage collection.

The above concept can be generalized as shown by Algorithm 1, which is the pseudo-code for allocation of a memory block using a single set of bits. When allocation of memory block is requested the bitmap is searched for the first free block which is returned if available. The corresponding bit in the bitmap is set which denotes that the block is used and the object is reachable. When there are no free memory blocks for further allocation, the bitmap is cleared (only the bitmap is changed in this step, without sweeping the heap objects). After that all live objects are traversed and marked in the bitmap (like in mark phase of MS). At the end of marking all memory blocks with unset bits are populated by garbage and may be allocated again.

```
func new()
  if ( NULL == ( addr=find_empty_block() ) )
    clear_bitmap()
    mark()
    if ( NULL == ( addr=find_empty_block() ) )
      abort()
  set_bit(addr)
  return (adr)
```

Algorithm 1. Allocation of memory block.

By closely integrating bitmap allocator with mark phase of MS collector through common use of bits, we rendered the sweep phase unnecessary. Therefore, the complexity is reduced compared to traditional MS collector and is proportional only to the amount of live data.

4 MARK-WITHOUT-MUCH-SWEEP

We utilize and further develop the presented idea of bitmap integration in the design of a new algorithm that we call mark-without-much-sweep (MwmS), since it still has to sweep some objects. We did not use other known techniques like lazy sweep or non-recursive marking, because they are orthogonal to our algorithm and we also wanted to isolate the effects of the proposed changes in relation to traditional MS algorithm.

4.1 Objects Categories

The objects are categorised as small and large. The large objects include arrays of any size, buckets, and all other objects larger than the threshold size. The large objects are held directly in the heap, whereas the small objects are held in the buckets.

The threshold size can be adjusted with a command line parameter. In this research we have used 120B as the threshold value. In all used benchmarks all objects, except some arrays, were smaller than the chosen threshold value.

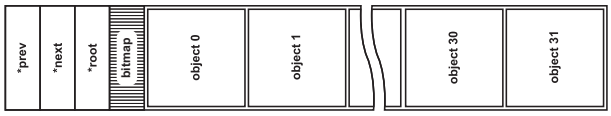


Fig. 1. Bucket organisation.

MwmS uses two-level allocator. At the lower level large objects (including buckets) are allocated by using any general purpose allocator. Higher level allocator allocates small objects in the buckets.

For low level allocation we have used Doug Lea's allocator [17]. Lea's allocator is well balanced with respect to fragmentation and execution speed, and is often used as part of the researches similar to ours [5, 6, 18].

4.2 Organisation and Allocation of Small Objects

The buckets implement the idea of mark-without-sweep on a small scale. Small objects of the same class are allocated using bitmaps inside buckets. They consist of a header containing bitmap and space used for placement of objects. Figure 1 shows bucket organisation. Each bucket is sized to hold 32 objects thus it has a 32-bit bitmap. The larger or smaller buckets can be implemented, but we choose 32 objects in order to fit the bitmap in one machine word of 32 bits. Buckets of the same type are linked together in doubly-linked list (*prev and *next). The list header is located in the class descriptor. The bucket header also includes a pointer to the class descriptor (*root).

Each small object has one word header which holds index of the object in the bucket and its offset from the bucket header. These values are used during marking to access the bitmap and generate the appropriate bit-mask for the bitmap. The index and offset can be derived from each other by knowing the object size. To avoid the computation we store both values. This does not incur space penalty since both values are stored inside single word.

It is worth noting that this organisation of buckets eliminates internal fragmentation in small objects. The internal fragmentation exists only at the lower level allocation. External fragmentation is different in comparison to traditional MS, because lower level allocator now handles bigger blocks of memory.

A roving pointer together with bitmap scanning is employed for higher level allocation (Algorithm 2). The roving pointer points to the bucket in which the last object of the corresponding class was allocated. When a new object is about to be created, the allocator first scans the bitmap of the bucket pointed to by the roving pointer. If all bits are set, roving pointer proceeds to the next bucket of the same type, after which bit scanning is carried out again. If scanning fails for the last bucket, the new bucket is requested from the lower level allocator. If this fails, the GC is performed after which all roving pointers are reset to the beginning of their corresponding bucket lists. The GC may free small objects and large objects including entire buckets. The higher level allocator then retries to allocate the small object and if this fails, the lower level allocator retries to allocate the new bucket. If the new bucket allocation fails, the heap is expanded by using sbrk() and the new bucket is created. The heap expansion fails when maximum heap is reached (this is not shown on simplified pseudo-code). Finally, allocation request is satisfied by returning an address of the first block in the new bucket.

```
func alloc_small_object(class)
  if ( NULL == ( object=alloc_in_bucket(class) ) )
    if ( NULL == ( bucket=create_bucket(class) ) )
      gc ()
      if ( NULL == ( object=alloc_in_bucket(class) ) )
        if ( NULL == ( bucket=create_bucket(class) ) )
          sbrk ()
          bucket = create_bucket(class)
        object = alloc_in_bucket(class)
    else
      object = alloc_in_bucket(class)
  return (object)
```

Algorithm 2. Small object allocation.

Rather than by shifting and masking, bitmap is more efficiently scanned by using GCC built in function which counts trailing zeroes [19]. It is supported on a number of architectures (e.g. Intel, AMD, ARM, Power, etc.). On our platform (Intel) `__builtin_ctz` compiles to single assembly instruction (`bsf`).

4.3 Organisation and Allocation of Large Objects

As already mentioned, the large objects are allocated by the lower level allocator. They use header bit for marking, which is stored in allocator header. During GC, the large objects are traced together with small objects, but the sweep phase scans only the large ones.

4.4 Bitmaps and Header Bits

During the mutator run, bitmap bits are used for allocation. The set bits denote used memory blocks which hold either reachable or non-reachable objects. In order to use the same bits for marking, they have to be cleared before GC (in traditional MS bits are already cleared by previous sweeping). We wanted to avoid an additional pass for clearing the bitmaps in buckets prior to marking phase. Hence, each bitmap is cleared when the bucket is accessed for the first time during marking (at the same time the mark-bit of the bucket is normally set, which is also an indication that the bitmap is cleared).

After marking is completed, all live objects (including buckets which contain reachable small objects) will have their associated mark-bits set. The sweep phase scans allocator headers of large objects, frees all unmarked memory blocks and clears mark-bits of marked ones.

4.5 Generality of MwmS

Standard improvements used in MS are applicable to MwmS as well. Large objects in MwmS can be swept lazily. The benefit would not be as significant as in MS since small blocks (which are in majority) are not swept at all. Different marking schemes (using explicit marking-stack with or without FIFO [9]) or various scanning techniques [20] are orthogonal to MwmS and could be freely employed in marking phase. Of course, marking of small objects should also clear bitmap and set mark-bit of the bucket, as described in the previous section.

MwmS can be used as a mature space collector in generational collection instead of traditional MS (like in e.g GenMS in MMTk which is regarded as high performance collector [5]). During promotion of objects from the younger generation into mature space, object should be allocated using MwmS two-level allocator, and their content copied as usual. Also, MwmS can be used as a backup tracer for cycles in reference counting.

However, to be used as a conservative collector, MwmS should be adapted. This is due to the fact that conservative GC can falsely identify a part of memory as a live object but must not change it [2]. Therefore, conservative GC uses bitmaps for marking. The large objects in MwmS have header mark-bits which should be displaced in separate bitmaps stored aside. The same should be done with bitmaps of small objects.

5 HEURISTIC HEAP EXPANSION

When a program starts, operating system assigns it a part of memory, of which some is used as a heap. In the presence of GC, the heap is traditionally expanded in the following way: if an allocation request was unable to find sufficiently large free block, the GC cycle is performed; if the GC was also unsuccessful (i.e. if it was unable to find and free enough dead objects in order to satisfy the preceding allocation request), the heap will be expanded.

Heap size impacts performance: if it is too small, garbage collection will occur more frequently and degrade system performance; on the other hand, if it is too large, the performance will be degraded by page swapping. Heap sizing policy is often hard-coded in a virtual machine and based on trial-and-error experiments [14]. In this research we developed heap expansion policy based on objects' behaviour rather than on values such as time taken for GC or current amount of live data in the heap.

In MwmS, objects of the same classes are held in the same type of buckets. Therefore the algorithm can easily keep track of behaviour and profile each class of objects during program runtime.

Ramps, peaks and plateaus are three patterns of memory usage observed in variety of programs [21]. These patterns represent: constant allocation, allocation followed by freeing, and equal rate of allocation and freeing, respectively. The phase behaviour of programs is also well-known and exploited in GC algorithms (e.g. [11, 15]). Peaks, ramps, and ramps combined with peaks are found in the benchmarks that we used (explained in Section 6). The behaviour of the DaCapo benchmarks is quite similar with the addition of a few plateaus [22].

For GC algorithms the worst case is ramp phase because collections are fruitless and they are followed by imminent heap expansion.

In the used benchmarks we observed that in ramp phases the majority of created objects are long-lived and belong to only few classes (often only one). When collection is called during the ramp phase the space is rarely freed, and is often not large enough to hold a new bucket. Therefore we concluded: (1) during ramp phases the GCs should be avoided, and (2) ramp phases can be recognized when no long-lived objects of prolific classes are freed in the collection.

Based on these observations we propose the following simple heuristics. If a program keeps creating long-lived objects of a certain class (objects are classified as long-lived if they survive the previous GCs), the corresponding bucket list should be expanded immediately rather than doing a GC after which the bucket list will be expanded anyway. To create the new bucket in a ramp phase, after the unsuccessful allocation, the heap has to be expanded. On the other hand, if objects of certain class are short-lived, the GC is preferred.

In our research we developed several variants of above-mentioned algorithm with heuristic behaviour. Each variant shows different results but basically it all boils down to space vs. speed trade-off. In this paper we present the variant which shows the best balance between extra heap required and speed gained due to the shorter collection.

The heuristic heap expansion (Algorithm 3) uses counters for each class of objects (and also for arrays) to keep the track of a number of unsuccessful garbage collections and a number of consecutive heap expansions. Initially both counters are set to zero. The counters are updated for the class that triggered the collection. When the heap is exhausted for the first time, garbage collection is called. Collection is successful if it frees a memory block sufficient to fulfil the allocation request (short-lived objects). This means the collection will be the preferred method

of gaining space for the object of that class the next time when the heap is full. On the other hand, if the collection is not successful, the next time when the heap fills it will be expanded instead (long-lived objects). This may be repeated several times. The number of repeats is determined by the number of unsuccessful collections and heap expansion factor. The heap expansion factor increases the effect of unsuccessful collections on the number of heap expansions. The factor may be set by a user (all results presented later are obtained with default value of one).

```
if ( 0 < class -> number_of_heap_expansions )
  class -> number_of_heap_expansions --
  sbrk ()
else
  gc ()
  if (NULL == ( object = alloc_object(class) )
    class -> number_of_unsucessful_GCs +=
             heap_expansion_factor
    class -> number_of_heap_expansions =
             class -> number_of_unsucessful_GCs
    sbrk ()
  else
    if ( 0 < class -> number_of_unsucessful_GCs )
      class -> number_of_unsucessful_GCs --
```

Algorithm 3. Heuristic heap expansion.

Finally, we should comment the fact that our heap sizing algorithm never decreases the heap size. It is a direct consequence of MS being non-moving algorithm. For the benchmarks used in this research, we examined the graphs where heap occupancy is plotted against time expressed in bytes. No single benchmark exhibits behaviour where a high peak is followed by significant drop of memory usage that remains at the low level for longer period. When the memory usage drops, the next phase quickly occurs with the peak of similar or even greater size. In DaCapo [22] the comparable behaviour is exhibited by all benchmarks except one (eclipse). This does not mean that the heap reduction is useless because some programs will certainly behave differently than the benchmarks (e.g. interactive programs that sequentially process data of different size, servers that occasionally receive “large” request, etc.). It only means that heap reduction would not be recognized as beneficial by using the aforementioned benchmarks.

6 METHODOLOGY

6.1 AGCS Simulator

Results presented in this paper are obtained with AGCS [23]. AGCS is trace-driven simulator intended for comparison of GC and allocation algorithms. Numerous types of results can be obtained: total time of memory-related events, allocation-time, GC-time, time of particular phases in GC and allocation algorithms, number of GCs and allocations, memory consumption, estimation of fragmentation, memory objects profile, stack depth and usage etc. AGCS is composed of several modules, each of which describes a part of a memory system such as heap, stack, memory allocator, garbage collector, etc. For each algorithm a separate module is written in C and compiled. Simulation implies execution of selected modules, i.e. algorithms for GC and allocation which are investigated.

There are several advantages of AGCS. It has very low dispersion of results (we explain this in more details shortly). The real organisation of heap and objects are replicated in memory and real GC/allocation algorithms are executed during simulation. It also means that the real cache and virtual memory mechanism are used and measured. AGCS can simulate GC and allocator which do not depend on particular language (e.g. Java) or runtime environment (e.g. JVM). Memory related events (object creation and deletion, reference assignments, stack-frame creation and deletion) are simulated and isolated which enables easier observation of behaviour of memory objects. Finally, AGCS is platform independent (providing that GCC compiler is present).

AGCS also has some disadvantages. Due to simulation of memory related events only, the rest of a mutator is not simulated so the total execution time cannot be evaluated. The influence of GC and allocator on the locality of the mutator cannot be evaluated for the same reason (with the exception of pointer assignments). Furthermore, the simulation is slower than the execution of real benchmarks and trace-files are very large (up to hundred gigabytes).

A trace-file contains memory-related events. It has a well-defined structure and is language independent, hence trace-files may be obtained by executing any program on instrumented version of any runtime environment (for this research we use Java benchmarks and instrumented Kaffe VM [24]). Precision of the simulation depends on the precision of the traced-events.

6.2 Benchmarks

We used three suites of benchmarks: all ten benchmarks of jOlden set [25, 26], four benchmarks of SPECjvm98 [27], and 12 benchmarks of JavaGrande [28]. We do not perform standard benchmarking. Instead, we use the benchmarks only as a source of trace-files with memory related events for simulation.

The first section of JavaGrande benchmarks is not used due to trivial memory behaviour. We did not used three benchmarks of SPECjvm98 (jess, jack, mpegaudio) due to the number of warnings in a simulation. AGCS checks all events from a trace-file, and reports a warning if it finds any inconsistency. Inconsistencies are result of imperfect

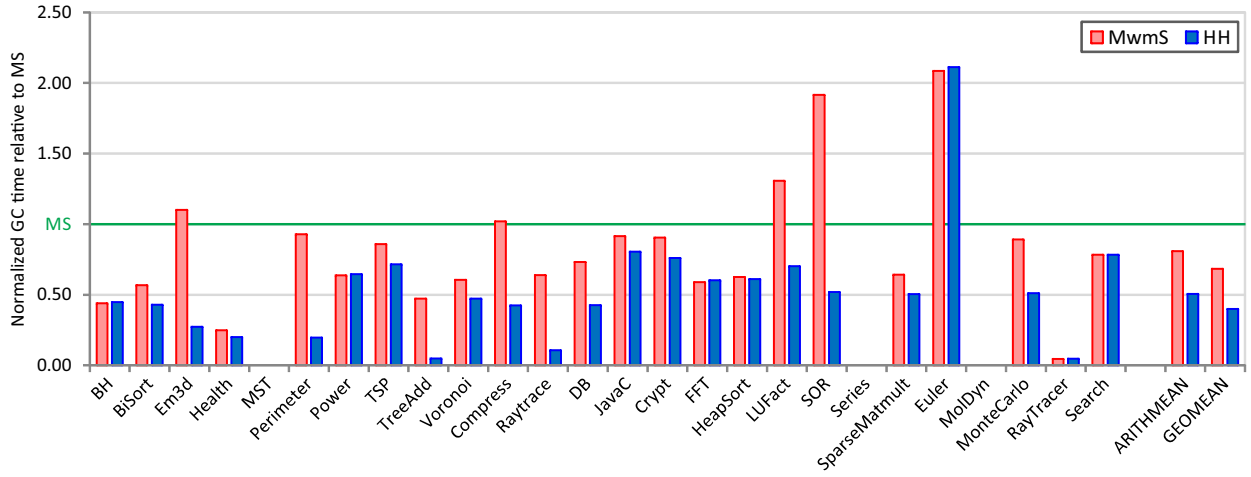


Fig. 2. Normalized GC time relative to Mark-Sweep (for all benchmarks).

tracing of some events related to operand stack in the instrumented Kaffe. We discard all benchmarks which have more than 1% of warnings (except DB of SPECjvm98). In all used benchmarks the number of warnings is much less than 0.1% of total simulated events, except in DB (6%).

Finally, for this research we used 26 benchmarks. Unfortunately, we could not use DaCapo benchmarks [22] since the instrumented version of Kaffe is older than needed for execution of DaCapo. However, the comparison of MS and MwmS could be scaled to larger benchmarks (like DaCapo) since GC and total times are proportional to the number of objects (except in case of cache misses and paging).

6.3 Environment

All simulations were performed on a system with Intel Core i3 (3.3GHz) and 4GB of memory, running Ubuntu GNU/Linux 13.04 distribution with and Linux kernel 3.8.0-31-generic (with word length of 32 bits).

6.4 Measurements

In this paper we measure and present three performance characteristics: garbage collection time, total time (time needed for the simulation of all memory-related events) and heap size. We chose the average number of processor cycles as the metric to measure the performance of the first two. Time was measured by reading time-stamp counter register with RDTSC/RDTSCP instructions [29]. To reduce the impact of other processes, simulations were performed in a single user mode. The dispersion of the obtained results is very small – relative standard deviation (coefficient of variation) is typically less than 1%. Very rarely, a result is obtained with greater dispersion due to the measurement of short time intervals and due to the multitasking nature of Linux kernel (even in single-user mode). Such results are discarded and simulation is repeated.

7 RESULTS

We now evaluate the effectiveness of MwmS, with and without heuristic heap expansion, and relate it to traditional MS (our own implementation in AGCS). Evaluation is performed by comparing the three performance characteristics obtained by simulating memory-related events of aforementioned benchmarks.

All results are presented in graphs where performance characteristics achieved by traditional MS collector are used as a baseline (value 1.00), and results achieved by our algorithms are expressed relative to MS. Results for each benchmark are given and computed as arithmetic mean across ten runs. The dispersion of the results is not shown because relative standard deviation is typically less than 1% and always less than 3%.

All simulations are performed with the same initial heap size of 1M and the heap increment value of 256K. Six benchmarks (BH, MST, Power, Series, MolDyn, and RayTracer) have highest watermark lower than 1M, therefore they can be executed in heaps with size smaller than initial 1M. Three of them (MST, Series, MolDyn) have total allocation requests less than 1M, hence they do not perform GC. We included them in the results anyway, in order to see the effect of allocation on total time.

At the end of this section we give the measurement results for time related to fixed heap sizes.

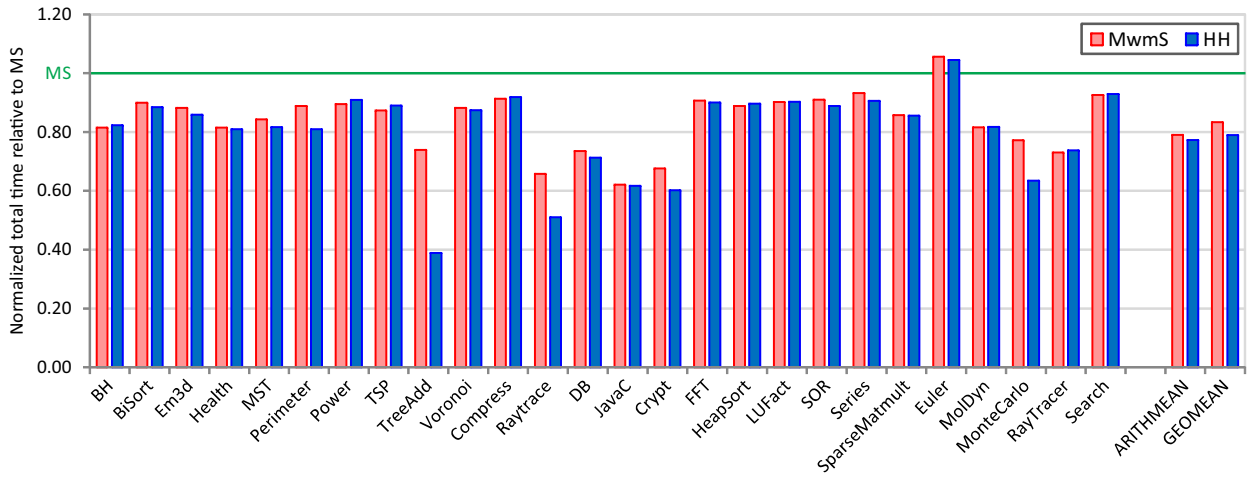


Fig. 3. Normalized total time relative to Mark-Sweep (for all benchmarks).

7.1 Garbage Collection Time

As show in Fig. 2, MwmS reduces the time spent in garbage collection for majority of benchmarks compared to traditional MS. Exceptions are Em3d, Compress, LUFact, SOR and Euler. However, by introducing heuristics in heap expansion policy (label HH in graphs – Heuristic Heap expansion), the time spent in collection compared to MS reduces for all benchmarks except for Euler.

Majority of objects and arrays in Euler are short-lived. At the beginning of benchmark run arrays are allocated whilst the allocation of objects starts later. MS places small objects easily in between arrays. Therefore, in MS collections occur with longer intervals leaving the objects enough time to die. In MwmS much larger buckets have to be placed in the heap and at that moment the free holes are too small. When collection is called it creates only small amount of free space because the objects and arrays did not have enough time to die. Thrashing [2] repeats which results in increased number of frequent collections and in turn with longer GC time in comparison to MS. Since majority of collections manage to free some space in each cycle, heap is almost never expanded and heuristics never picks up. For that reason the results for both of our algorithms are similar.

In some benchmarks (Em3d, Compress, LUFact, SOR) MwmS degrades GC time. Since these benchmarks demonstrate ramp behaviour, the heuristic variant of MwmS recognizes ramps, avoids collections and expands the heap, which reduces GC time.

Significant improvement of GC time between the two variants of our algorithms is evident in benchmarks which create large number of same types of objects during ramp phase. Typical example is TreeAdd which creates more than 1M objects of class TreeNode. In the concrete example of TreeAdd, MwmS spent 52% less time in GC than MS. Version with heuristic heap expansion reduces GC time by 90% in comparison to MwmS. This gives total of 95% improvement of heuristic version of MwmS when compared to traditional MS.

On average, MwmS improves GC time by 19% (31% geometric mean). Heuristic MwmS is even better. It reduces average GC time by 49% (60% geometric mean) in comparison to MS. When calculating arithmetic and geometric means we excluded benchmarks which have zero GC time (MST, Series, MolDyn).

7.2 Total Time

Figure 3 shows total time (time needed for simulation of all memory-related events) relative to MS. The total time is reduced across all benchmarks and with both versions of MwmS (exception being Euler which has doubled GC time for reasons explained earlier).

Reduced total time is partially attributed to speedup gained in allocation which is fast due to it's simplicity (as explained earlier). The other reason for speedup is faster GC (on average) when using MwmS. However, the contributions of allocation and GC speedups are still insufficient to explain the entire improvement of total time. We can only assume that it can be attributed to better cache locality of reference assignments. This is also indicated by the number of minor page faults which is always somewhat greater in MS.

MwmS improves total time by 20% on average (16% geometric mean), whereas Heuristic versions improves it by 22% (21% geometric mean).

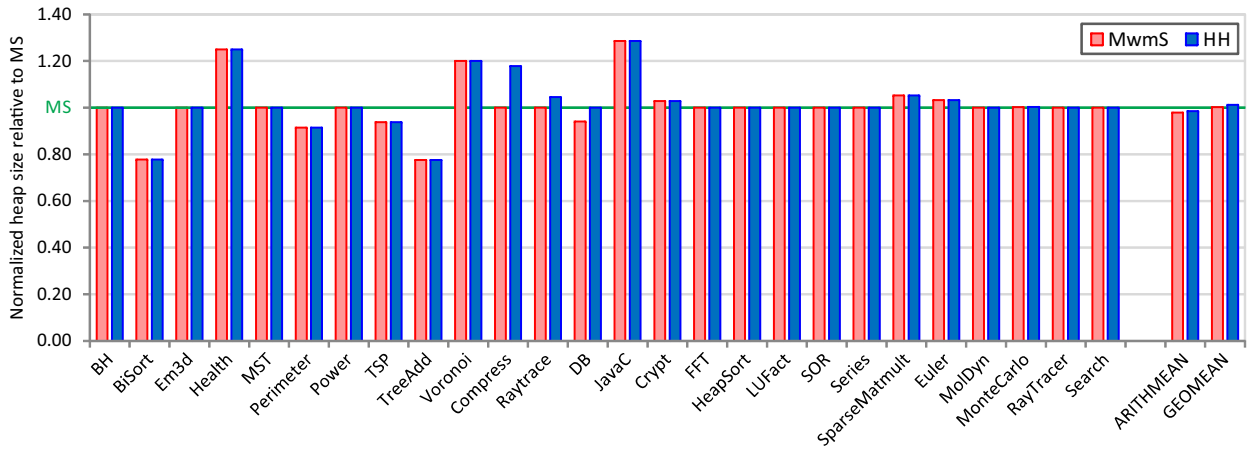


Fig. 4. Normalized heap size relative to Mark-Sweep (for all benchmarks).

7.3 Maximum Heap Size

Although the reduced internal fragmentation is one of the advantages of MwmS, we do not measure it directly. There are several definitions which summarize the fragmentation for the whole application and neither one is ideal [6]. It would be impractical to measure the fragmentation at every point of allocation/freeing. Even then, the fragmentation causes problems only when allocation requests are adverse. Therefore, we measure the maximum heap size that is achieved during benchmark run, because this value shows the total effect of fragmentation on the memory consumption. Maximum achieved heap size relative to MS for each benchmark is shown in Fig. 4.

There are only three benchmarks (Health, Voronoi, JavaC) in which the heap is noticeably larger (20% or more) when using either version of our algorithm in comparison to MS. The reason is external fragmentation (in traditional MS all objects are placed together in a heap, so big arrays and smaller objects are interleaved giving smaller external fragmentation). Solution would involve implementation of compaction mechanism for MwmS.

For several benchmarks (e.g. Em3d, Perimeter, TreeAdd, MonteCarlo) the heuristic version has reduced the GC time in comparison to MwmS. Although it may seem counterintuitive, this does not enlarge the heap because the heap is expanded more aggressively to the required size earlier in the course of the execution. This reduces the number of GCs and in turn GC time.

There are benchmarks in which heuristics enlarges heap more than needed (e.g. Compress which has three ramp phases). This happens when the benchmark creates and keeps one class of objects alive during part of the program run but it renders them unreachable at the end of a ramp phase. In that particular case heuristic picks up the pace with object allocation and expands heap. The more objects are created and kept alive, the more aggressively heap is expanded. Algorithm does not recognize the exact point when objects start to die, which results in a few redundant expansions before switching back to collection.

Finally, there are several benchmarks where maximum achieved heap is smaller when using either version of our algorithm in comparison to MS (e.g. TreeAdd, BiSort, Perimeter, etc.). In these benchmarks relatively small number of arrays is created (e.g. 0.04% in case of TreeAdd), whilst large number of class instances are dominant (e.g. BlackNode, WhiteNode and GrayNode make up for 99.73% of all created objects in Perimeter). Small objects are packed together in buckets within memory blocks of exact size thus eliminating internal fragmentation and reducing the memory footprint, which results in the smaller maximum heap size.

Overall, the heap size is kept within reasonable limits having the same size as in traditional MS in most cases, and being somewhat better or worse in equal number of cases. Average differences are negligible, about 1 or 2 percent.

7.4 GC and Total Time Depending on Heap Size

Similarly to related researches, Figs. 5 and 6 plot GC and total time as a function of heap size. Time is measured for five benchmarks (BH, Health, Power, Raytrace and JavaC) which have relatively low ratio of high-watermark and total requested memory (6.5%, 10.8%, 11.7%, 29.8%, and 30.0% respectively). There is no point of measuring these results for benchmarks with higher ratio since they will fit completely in larger heaps resulting in GC time equal to zero. The five benchmarks shown here are more

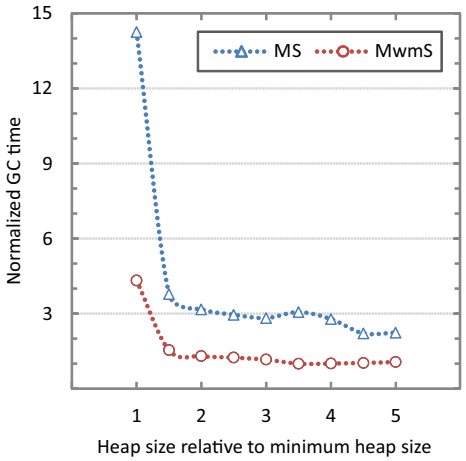


Fig. 5. Normalized GC time for MS and MwmS for selected benchmarks.

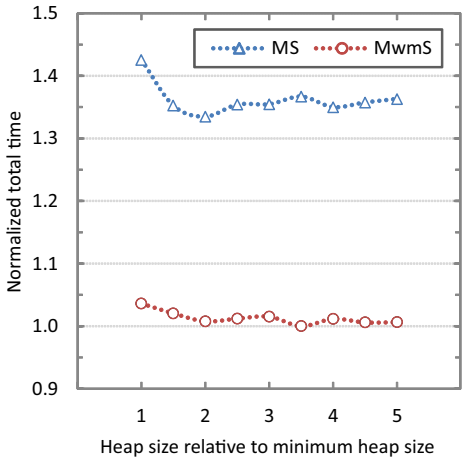


Fig. 6. Normalized total time for MS and MwmS for selected benchmarks.

similar to DaCapo than the other benchmarks we use, although DaCapo benchmarks are "bigger" and their ratio of high-watermark and total requested memory is even lower. The y-axis expresses time normalized to the fastest time. The x-axis expresses heap size as a multiple of the smallest heap size in which the benchmarks execute.

The results for heuristic variant are not included because the heap size is fixed and the same results would be measured with or without the heuristics.

For the minimum heap size MwmS shows 70% reduction of GC time in comparison to traditional MS. GC time decreases for both algorithms as heap grows, however for all sizes MwmS shows better results. At worst, GC time for MwmS is 52% better in comparison to traditional MS (63% on average).

In comparison to MS, the total time in MwmS is reduced by cca. 25% across all heap sizes.

8 RELATED WORK

Zorn's generational collector for LISP uses MS inside each generation which has its bitmap and two regions divided in areas. The areas in fixed-size region hold one type of objects. Size of the areas is predetermined so they store various numbers of objects. The variable-size region holds bodies and separate headers of variable-size objects. Two-level allocator holds free areas in a free list. A free area can be assigned to a type that maintains its areas in a list. When collection starts, the bitmap is cleared and marking proceeds as usual while sweeping is lazy [10].

Boehm-Demers-Weiser (BDW) conservative MS is suitable for C/C++. Small objects of the same size and kind are stored in memory chunks with typical size of one page. Large objects are bigger than half a chunk. They are placed in several grouped chunks. Due to conservative marking, the mark-bits are outside objects and organized in bitmaps. The bitmaps are placed in page headers which are stored separately in a list for better locality. Two-level allocator uses free lists segregated by size and kind (for small objects) and by several sizes (for large objects). Sweep is lazy in order to increase the locality of sweep and subsequent allocation [4].

Colnet et al. also use MS and divide memory into typed chunks for Eiffel. Objects have GC header with mark-bit. Bump-like allocation is used in chunks but each type has its free-list. Resizable objects of moderate size are stored in typed chunks of bigger size. Very large objects are treated like chunks with one object. Mark and sweep phases use customized functions optimized for a particular type [3].

In IMMIX the heap is divided into blocks (32KB) which are further divided into lines (128B). Several small objects are allocated in lines by using fast bump-like allocator. When objects are marked, the lines occupied by them are also marked. Sweep phase is made very fast by sweeping only lines instead of all objects. Fragmented blocks are occasionally defragmented by copying objects into target blocks. This also improves locality and reduces cache misses giving better total application time [5].

In aforementioned algorithms (Zorn, BDW and Colnet) and in MwmS objects are grouped by type, while IMMIX holds objects of different types together. However, in MwmS variable sized buckets hold fixed number of objects, in contrast to the other algorithms where memory blocks have fixed size and hold variable number of ob-

jects. While MwmS does not sweep small objects at all, BDW and Zorn sweep lazily and use free list for small object allocation. Colnet et al. have bump-like allocation, but they nevertheless use free lists segregated by type. IMMIX is similar in restricting the sweep to lines only (lines typically hold 1-4 objects), while in MwmS sweeping is restricted to the buckets only. All algorithms sweep the large (or variable sized) objects in traditional way.

9 FUTURE WORK

In some benchmarks (namely, Euler) heuristic MwmS does not achieve expected improvements. This can be addressed by extending heuristics rules. GC efficiency can be taken into account and if it is too low, the heap could be expanded immediately. We make some preliminary measurements and the results are promising. For example, if heap grows from 8M to 9M in Euler, the GC time is reduced for more than 50% in comparison to MS.

Possible direction for future research involves introduction of compacting mechanism both for arrays (e.g. with two semi-spaces) and small objects (e.g. two-finger compaction). Although compaction will increase collection time, it will reduce external fragmentation and allow heap reduction.

Yet another possibility is to develop an incremental version (e.g. utilising tricolour marking) and generational version of the algorithm to reduce collection pauses.

10 CONCLUSION

Mark-sweep collectors are still competitive especially in tight heaps but the simple modifications proposed here can further improve their performance. We introduced buckets which hold small objects segregated by type (i.e. class) and we integrated the collector bitmap and allocator bitmap. This novel organisation completely avoids the sweep phase and freeing of dead objects in buckets, thus improving the performance of GC in most of benchmarks. Further improvement of performance has been achieved by using simple heuristics to control heap expansion. The final algorithm outperforms the traditional MS in all but one benchmarks and it is significantly better in half of them. The allocation time is also reduced which improves the mutator time. Since the heap sizes of our algorithms are similar to traditional MS, the goal of keeping the heap in reasonable limits is achieved as well. Hence, the new algorithm is not only a good base for further investigation but also an excellent candidate for combination with other known techniques which can be expected to improve its performance even more.

REFERENCES

[1] R. Jones and R. Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, 1996.

[2] R. Jones, A. Hosking, and E. Moss, *The Garbage Collection Handbook: The Art of Automatic memory Management*. Chapman and Hall/CRC Press, 2012.

[3] D. Colnet, P. Coucaud, and O. Zendra, "Compiler Support to Customize the Mark and Sweep Algorithm," in *Proceedings of the International Symposium on Memory Management (ISMM'98)*, pp. 154–165, 1998.

[4] H.-J. Boehm, "Conservative GC Algorithmic Overview." http://www.hpl.hp.com/personal/Hans_Boehm/gc/gcdescr.html.

[5] S. Blackburn and K. McKinley, "Immix: A Mark-Region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance," in *Proceedings of the Conference on Programming Languages Design and Implementation (PLDI'08)*, pp. 22–32, 2008.

[6] M. Johnstone and P. Wilson, "The Memory Fragmentation Problem: Solved?," in *Proceedings of the International Symposium on Memory Management (ISMM'98)*, pp. 26–36, 1998.

[7] Y. Chung, S.-M. Moon, K. Ebcioglu, and D. Sahlin, "Reducing Sweep Time for a Nearly Empty Heap," in *Symposium on Principles of Programming Languages (POPL'00)*, pp. 378–389, 2000.

[8] B. Iyengar, E. Gehringer, M. Wolf, and K. Manivannan, "Scalable Concurrent and Parallel Mark," in *Proceedings of the International Symposium on Memory Management (ISMM'12)*, pp. 61–71, 2012.

[9] R. Garner, S. Blackburn, and D. Frampton, "Effective Prefetch for Mark-Sweep Garbage Collection," in *Proceedings of the International Symposium on Memory Management (ISMM'07)*, pp. 43–54, 2007.

[10] B. Zorn, *Comparative Performance Evaluation of Garbage Collection Algorithms*. PhD thesis, University of California at Berkeley, 1989.

[11] C. Zhang, K. Kelsey, X. Shen, C. Ding, M. Hertz, and M. Ogihara, "Program-level Adaptive Memory Management," in *Proceedings of the International Symposium on Memory Management (ISMM'06)*, pp. 174–183, 2006.

[12] T. Brecht, E. Arjom, C. Li, and H. Pham., "Controlling Garbage Collection and Heap Growth to Reduce Execution Time of Java Applications," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*, pp. 353–366, 2001.

[13] J. Singer, R. Jones, G. Brown, and M. Lujan, "The economics of garbage collection," in *Proceedings of the International Symposium on Memory Management (ISMM'10)*, pp. 103–112, 2010.

[14] D. White, J. Singer, J. Aitken, and R. Jones, "Control Theory for Principled Heap Sizing," in *Proceedings of the International Symposium on Memory Management (ISMM'13)*, pp. 27–38, 2013.

[15] T. Yang, E. Berger, M. Hertz, S. Kaplan, and J. Moss, "Automatic Heap Sizing: Taking Real Memory Into Account," in *Proceedings of the International Symposium on Memory Management (ISMM'04)*, pp. 61–72, 2004.

[16] C. Grzegorczyk, S. Soman, C. Krintz, and R. Wolski, "Isla Vista Heap Sizing: Using Feedback to Avoid Paging," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pp. 325–340, 2007.

[17] D. Lea., "A Memory Allocator." http://gee.cs.oswego.edu/dl/html/malloc.html, 2000.

[18] M. Hertz and E. Berger, "Quantifying the Performance of Garbage Collection vs. Explicit Memory Management," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pp. 313–326, 2005.

[19] GNU, "A GNU Compiler Collection Manual." http://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html.

[20] R. Garner, S. Blackburn, and D. Frampton, "A Comprehensive Evaluation of Object Scanning Techniques," in *Proceedings of the International Symposium on Memory Management (ISMM'11)*, pp. 33–42, 2011.

[21] P. Wilson, "Uniprocessor Garbage Collection Techniques," in *Proceedings of the International Workshop on Memory Management (IWMM'92)*, pp. 1–42, 1992.

[22] S. Blackburn, R. Garner, C. Hoffmann, A. Khan, K. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo Benchmarks: Java Benchmarking Development and Analysis," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'06)*, pp. 169–190, 2006.

[23] D. Ivančić, D. Basch, and N. Hlupić, "AGCS – Modular, Extensible and Portable Computer Memory Simulator," in *Proceedings of MIPRO 2007 International Convention*, pp. 27–32, 2007.

[24] D. Basch and J. Borozan, "Results of Profiling and Analysis of Behaviour of Memory Objects in Java," *International Journal of Simulation, Systems, Science and Technology*, vol. 6, no. 3–4, pp. 26–42, 2005.

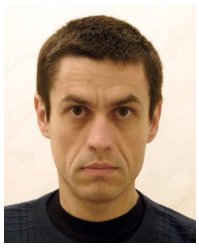
[25] B. Cahoon and K. McKinley, "Data Flow Analysis for Software Prefetching Linked Data Structures in Java," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques tions (PACT'01)*, pp. 280–291, 2001.

[26] "Java Olden Benchmarks." ftp://ftp.cs.umass.edu/pub/osl/benchmarks/jolden.tar.gz.

[27] Standard Performance Evaluation Corporation, "SPECjvm98 Documentation," 1999.

[28] EPCC, "The Java Grande Forum Benchmark Suite." http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html.

[29] G. Paoloni, "How to benchmark code execution times on intel® ia-32 and ia-64 instruction set architectures." http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf, 2010.



Danko Basch Danko Basch received B.Sc., M.Sc., and Ph.D. from Faculty of Electrical Engineering (FER), University of Zagreb in 1991, 1994, and 2000 respectively. Since 1991 he works at FER at Department of Control and Computer Engineering, where he is involved in scientific projects and educational activities. At present, he is full-time professor at FER. His research interests include programming language design and implementation, memory management, and modelling and simulation. Danko Basch is a author/coauthor of several papers and five books (in Croatian Language). He is a member of ACM.

Dorian Ivančić received his B.Sc. and M.Sc. degree in Computer Science from the Faculty of Electrical Engineering and Computing, University of Zagreb in 2001 and 2006, respectively. He is currently holding the position of Chief Information Security Officer in Croatia osiguranje. His interests span from low-level mechanisms like memory management and cpu organization, across virtualisation and networking security all the way to security management at the organisational level. He is an ISACA member since 2007 and holds CISA and CISM certifications.

Nikica Hlupić graduated at the Faculty of Electrical Engineering and Computing (FER) in Zagreb, Croatia, in 1993, where he also received M.Sc. degree in 1997 and Ph.D. degree in 2001. At present he is associate professor at the Department of Applied Computing, FER, where he lectures several undergraduate and graduate courses. His scientific interests include estimation theory, optimization, statistics and general theory of algorithm design. He has written a number of scientific and professional publications and he participated in several projects. He is a memeber of IEEE.

AUTHORS' ADDRESSES

Prof. Danko Basch, Ph.D.
Departement of Control and Computer Engineering,
Faculty of Electrical Engineering and Computing,
University of Zagreb,
Unska 3, HR-10000, Zagreb, Croatia
email: danko.basch@fer.hr

Dorian Ivančić, M.Sc.
Croatia osiguranje d.d.
Miramarska 22, HR-10000, Zagreb, Croatia
email: dorian.ivancic@gmail.com

Prof. Nikica Hlupić, Ph.D.
Departement of Applied Computing,
Faculty of Electrical Engineering and Computing,
University of Zagreb,
Unska 3, HR-10000, Zagreb, Croatia
email: nikica.hlupić@fer.hr

Received: 2014-05-07
Accepted: 2014-11-18