Kindergarten Cop: Dynamic Nursery Resizing for GHC

Henrique Ferreiro   Laura Castro
Department of Computer Science,
University of A Coruña
{hferreiro,lcastro}@udc.es

Vladimir Janjic   Kevin Hammond
School of Computer Science,
University of St Andrews
vj32@st-andrews.ac.uk, kevin@kevinhammond.net

Abstract

Generational garbage collectors are among the most popular garbage collectors used in programming language runtime systems. Their performance is known to depend heavily on choosing the appropriate size of the area where new objects are allocated (the *nursery*). In imperative languages, it is usual to make the nursery as large as possible, within the limits imposed by the heap size. Functional languages, however, have quite different memory behaviour. In this paper, we study the effect that the nursery size has on the performance of lazy functional programs, through the interplay between cache locality and the frequency of collections. We demonstrate that, in contrast with imperative programs, having large nurseries is not always the best solution. Based on these results, we propose two novel algorithms for dynamic nursery resizing that aim to achieve a compromise between good cache locality and the frequency of garbage collections. We present an implementation of these algorithms in the state-of-the-art GHC compiler for the functional language Haskell, and evaluate them using an extensive benchmark suite. In the best case, we demonstrate a reduction in total execution times of up to 88.5%, or an 8.7 overall speedup, compared to using the production GHC garbage collector. On average, our technique gives an improvement of 9.3% in overall performance across a standard suite of 63 benchmarks for the production GHC compiler.

Keywords Generational garbage collection, Cache locality, Allocation area, Functional programming

1. Introduction

For languages with automatic memory management, garbage collection (GC) performance is critical for achieving good overall application performance. One of the most widely used GC algorithms is *generational garbage collection*, in which new objects are initially allocated in a *nursery* area, and subsequently promoted to older generations if they survive sufficiently long. The size of the nursery is a crucial parameter for obtaining good performance since this impacts both the frequency of GCs and potentially on the cache behaviour. For imperative programs, a standard mechanism for reducing GC time is to make the nursery as large as possible. The rationale is that a larger nursery causes less frequent collections, and so increases the chances that more data becomes garbage. Current trends [2, 6, 14] focus on *dynamically* changing the nursery size while the program is running. Decisions are usually made based on the maximum heap size, as indicated by the user, and on various statistics about previous GCs in the same program run, such as the amount of live data.

Lazy functional languages, such as Haskell [8] are becoming increasingly used. Compared with imperative programming, lazy functional programming offers many benefits to programmers, such as a very high level of abstraction, referential transparency, transparent use of infinite data structures and implicit sharing of data. However, lazy functional programs have quite different memory behaviour to their imperative counterparts. Due to their reliance on immutable data, they typically perform many more memory allocations, requiring more frequent garbage collections and resulting in larger memory footprints. GC can easily become the limiting factor for achieving good performance. Despite the different allocation patterns, many runtime systems for functional languages use algorithms that have been tuned for imperative languages, ignoring, for example, the effects that these algorithms have on the cache behaviour in functional programs. In this paper, we study the effects that the nursery size has on the performance of functional programs, focusing on the state-of-the-art GHC compiler and runtime system for Haskell [8], which uses a generational, copying GC. The specific research contributions of this paper are:

- we *quantify* the relation between nursery size and program performance, expressed in terms of the effect of the nursery size on the runtime behaviour (percentage of cache misses, frequency of garbage collection, etc.) (Section 3);
- we propose the *TAA*$^+$ algorithm, a simple modification to an existing dynamic nursery resizing algorithm [2] that notably improves its performance (Section 4);
- we propose *SLR*, a novel algorithm for dynamic nursery resizing that *predicts* changes in the program's memory behaviour and *adapts* to them (Section 4);
- we implement these algorithms in the GHC Haskell compiler; and
- we evaluate the performance of these approaches on an extensive Haskell benchmark suite, the *nofib* suite, demonstrating a performance improvement of up to 50% for some programs against the production GHC garbage collector, with almost no programs adversely affected (Section 5).

The key novelty of our approach is that, in addition to garbage collection time, we also take into account the effect of the nursery size on the *mutator* time (i.e. the program execution without garbage collections), since this is directly affected by cache locality.

2. Background

This paper focuses on GHC [12], the most widely used compiler and runtime system for the purely functional programming language Haskell. However, our algorithms are applicable to any system that uses generational garbage collection. In this section, we introduce concepts related to generational garbage collection, and give a brief description of GHC, paying special attention to its garbage collector.

2.1 Generational Garbage Collection

Generational garbage collection is based on the assumption that most objects die young – this is called the *generational hypothesis* [7]. The program heap is divided into hierarchical regions called *generations* that hold objects of different ages. Each generation can be collected independently, using any suitable algorithm (e.g. mark and sweep or a copying approach). The most common configuration is to have only two generations: the *young generation* and the *old generation*. The young generation is usually divided into the *nursery* (or *allocation area*) where new objects are allocated, and the *reserved* or *survivor* area. After an object survives a number of collections, it is *promoted* to the next generation.

When the nursery becomes full, a *minor GC* is triggered on the young generation. Depending on the details of the generational collector, live objects from the nursery are either copied directly into the next generation, or are copied into the next space in the generation, in a process called *ageing*. Ageing is used to prevent the problem known as *premature promotion*, where recently allocated objects are promoted to older generations and become garbage shortly thereafter. When the occupancy of the old generation surpasses a given limit, a *major GC* is triggered, where the whole heap is collected. A generational collector is designed to perform *minor collections* (collecting the young generation) much more often than major ones (collecting older generations). Experiments have shown that the generational hypothesis generally holds for most programs and, therefore, each minor collection will only have to perform a small amount of work to collect live objects. By promoting them to older generations, objects with longer lifetimes are only traversed when major collections are triggered.

A generational garbage collector generally has better cache behaviour than a non-generational one, but its performance might be heavily dependent on the size of the nursery [16, 18]. Nurseries smaller than the L2 cache favour data locality and reduce the cache misses that are due to object allocation. However, increasing the size of the nursery will reduce the number of GCs by increasing the time spent in the mutator. This will allow more objects to become dead before the next GC, reducing the overall GC time. Most generational designs have considered variable-sized nurseries where, after calculating the space that is needed to perform a collection and the size of the old generation, the remainder of the heap space is left to the nursery [3, 6, 14].

Such designs clearly place more importance on GC performance than on cache locality. This is appropriate when dealing with imperative/object-oriented language implementations, such as Java or C++. There has been significantly less work on tuning nursery sizes for functional programming languages. In [1], it is claimed that lazy functional languages are a good match to current cache designs. In [2, 9], it is suggested that small nursery sizes give the best performance. Neither study provides a detailed analysis of how the tension between the cache behaviour of the mutator and the overall GC performance affects the overall performance of the program.

2.2 Garbage Collection in GHC

GHC currently uses a generational, copying, stop-the-world garbage collector [10]. While there have been attempts to introduce concurrent and per-core garbage collectors [5, 9], they have been abandoned due to complexity. The collector used in GHC uses a block-structured heap so that it does not have to be partitioned into contiguous spaces. It splits the heap into two generations where the young generation has a per-thread private nursery and a shared ageing area, which stages objects prior to their promotion to the old generation. Many of these features can be influenced by tunable runtime parameters. In this paper, we will just focus on the nursery size, however. By default, the nursery has a fixed size of 500KB, set at the start of the execution. A different fixed value can be provided for the nursery size, or the runtime system can be instructed to dynamically adapt the size to changes in memory usage using an algorithm that was originally described by Appel [3]. Following each GC, the heap usage of the program is checked, and after subtracting the space that is needed for the next GC, a *fraction* of the remaining heap is assigned to the nursery. To approximate these values, it is assumed that currently live data will remain live in the next GC. The new nursery size is calculated according to the formula:

$$\frac{H - N}{1 + p}$$

where N is proportional to the size of the live data[1], and p is the percentage of copied data in the nursery. The value of p allows the nursery to grow to over half of the calculated free space. For example, if the live data in the nursery amounts to a third of its size and we assume that this trend will continue, we would use 75% of the free space for the nursery.

We have deliberately avoided mentioning the value H until this point. H is calculated to be twice the amount of the globally live data. However, this value is *only calculated following a major collection* – only at this point can the complete memory usage be accounted for. The reasoning behind this decision is to allow the program to use the largest possible nursery without increasing its current memory requirements. Between two major garbage collections, the amount of live data promoted from the young generation will cause the value of N to increase while H will remain constant. As a result, the nursery will decrease its size until the next major GC. At this point, the new memory requirements of the program will be calculated again, and the nursery will grow by the amount of freed memory. Then, the cycle starts again. For programs with huge memory requirements, this algorithm can take some time to reach the peak memory usage, so GHC has the option of providing the initial value of H as a runtime parameter. There is also a standalone tool, `ghc-gc-tune` [13] that can automatically profile a Haskell program to discover the best static value for the nursery and heap size ($-A$ and $-H$ runtime parameters). This, however, involves running the program many times with different values for nursery and heap sizes, which may be infeasible for large programs. In the next section we explore the performance impact of the nursery sizing options that are available in GHC and offer some insights that will be used to derive an improved generational GC algorithm.

3. Impact of Nursery Size on Program Execution Time

As mentioned above, most nursery resizing policies try to make the nursery as large as possible, within some given heap limit. This has proved to be successful for imperative programs [4, 17], despite ignoring some important factors, such as cache locality. The question is whether this policy also works well for functional programs, which typically allocate much more data than their imperative equivalents. In this section, we attempt to address this question by investigating the impact that different nursery sizes have on the execution time of a functional program.

[1] All live data in the generations that are to be collected is counted twice.

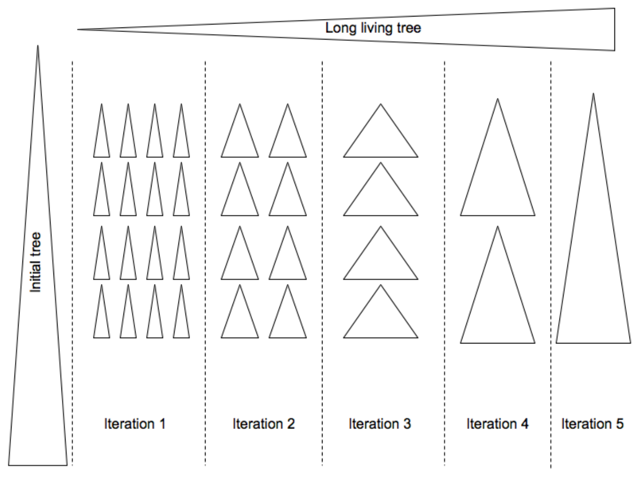

Figure 1. Behaviour of the `binary-trees` benchmark.

3.1 The `binary-trees` Benchmark

As our test example, we have selected the `binary-trees` benchmark, a "simplistic adaptation of Hans Boehm's GCBench"[2]. This benchmark was designed to test the GC behaviour when allocated objects have different lifetimes. Its intention is to simulate real programs with irregular memory behaviour in which some objects are retained throughout the whole execution of the program, and other objects (with varying lifetimes) are dynamically allocated and collected. The benchmark creates and traverses a number of balanced binary trees as shown in Figure 1. It first allocates and traverses an initial large binary tree, in order to "stretch" the memory. It then allocates a long-lived binary tree. Subsequently, over a number of iterations, a sequence of trees are allocated, traversed and freed. In each iteration, the sizes of the allocated trees are increased, and their number decreased, so that the number of tree nodes that are allocated remains constant, but the object lifetimes become increasingly longer. Finally, the long-lived tree is traversed and freed. Full source code is given in Appendix A.

3.2 Runtime Behaviour of `binary-trees`

Our measurements of the runtime behaviour of `binary-trees` were conducted on a 2.4GHz Intel Core2 quad-core machine, with separate 32KB, per-core L1 data and instruction caches, and a shared 4MB L2 cache, using GHC 7.6.3. The total amount of RAM available was 4GB. Figure 2 shows the amount of live data over the execution time of the program. This was measured by forcing the use of a single generation and counting the copied data at each collection. The effect of allocating the first tree can be observed at the beginning of the execution: 12MB of memory is allocated for this data structure, and then immediately freed. In the remainder of the execution, around 6MB are constantly used by the long-lived tree. We can observe several spikes on the graph, which correspond to peaks of memory usage when the lists of trees are allocated. The spikes progressively increase in size, which indicates that there is more live data when smaller lists of larger trees are allocated than with larger lists of smaller trees. This is due to the fact that, in the latter case, some of the trees are processed and collected before the whole list is constructed. This gives a smaller memory footprint.

[2] Taken from `http://benchmarksgame.alioth.debian.org/u64/performance.php?test=binarytrees`.

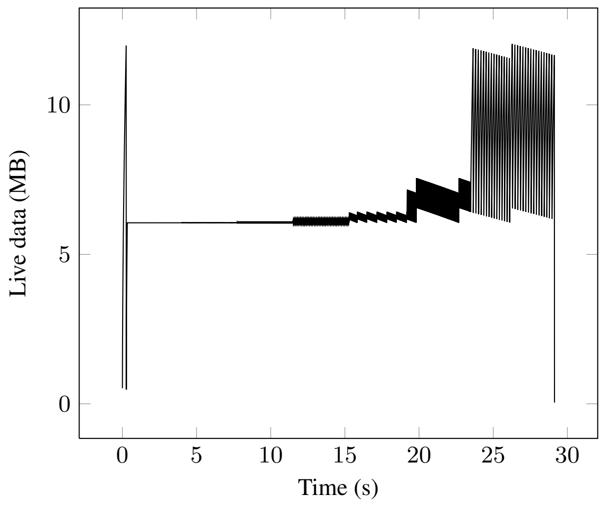

Figure 2. Memory usage of `binary-trees`.

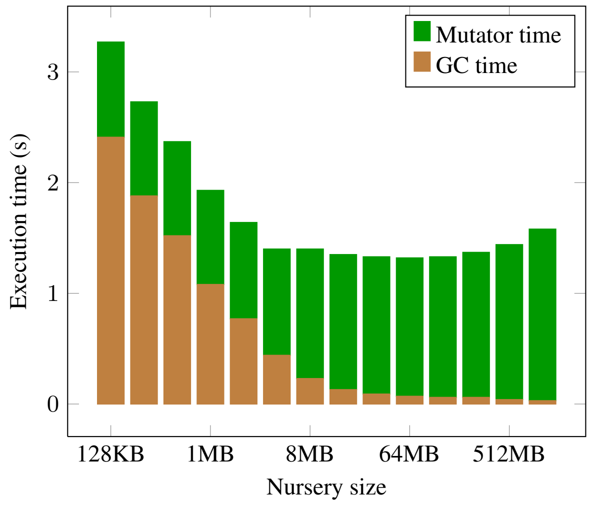

Figure 3. Runtime of `binary-trees` with different fixed-size nurseries.

3.3 Nursery Size and Cache Behaviour

Nursery size and Mutator and Garbage Collection time. We first want to verify whether the usual heuristics for nursery sizing are equally beneficial for functional programs, and to understand the effects that nursery size has on execution time. In order to do so, we ran `binary-trees` a number of times, with various fixed-size nurseries. Figure 3 plots the total runtime (on a logarithmic scale) against the nursery size. The mutator and garbage collection times are shown, respectively, as green and orange areas. We can observe that the execution time for `binary-trees` is highly dependent on the nursery size. In particular, increasing the nursery size can *halve* the execution time. Another interesting point to observe is that, beyond a specific size (around 128MB), the total execution time starts to increase. The division of time between the mutator and the collector gives a simple explanation for this: while increasing the nursery size has the anticipated result of reducing the time that is required for garbage collection, it also increases the mutator time. This is due to a loss of locality, indicating that it is not enough to

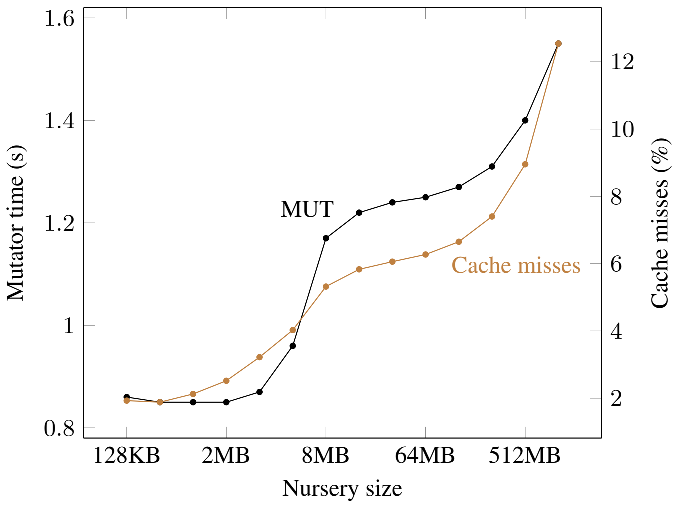

Figure 4. Relationship between mutator time and cache misses.

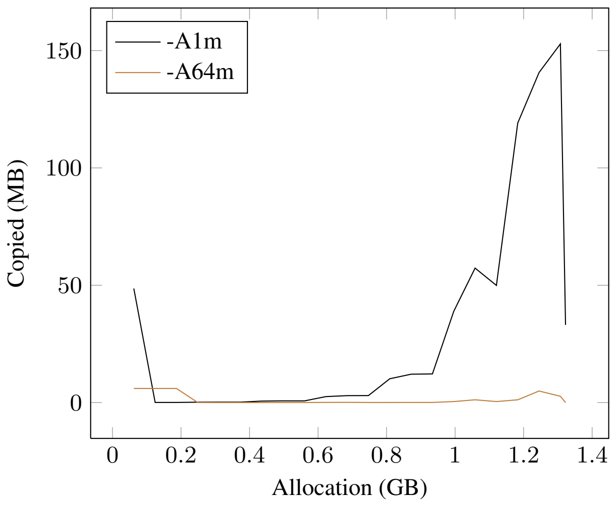

Figure 5. Copying comparison with different nursery size.

simply make the nursery as large as possible (as for an imperative program) in order to obtain the best execution time.

We conducted the same measurements for the *nofib* benchmarks that we will use in Section 5. We observed that the same pattern is repeated in many of them: while garbage collection time is proportional to the nursery size, as is already known from the literature, we verified that the mutator time is *inversely proportional to the nursery size*. Additionally, for many of these benchmarks, there is a nursery size threshold that marks the point after which the reduction in garbage collection time cannot compensate for the increased mutator time, and the overall execution time is therefore increased. This behaviour occurs in programs which spend a moderate amount of time doing GC. Additionally, we also observe that for programs that spend very little time in garbage collection, it is not possible to improve execution time by increasing the nursery size; *in these cases, the best time is obtained by using a small nursery*. Moreover, for programs that spend a significant time in garbage collection, but which also have a large amount of live data, increasing the nursery size has very little impact on the collection time. In this case, the only way to improve the execution time is to use the largest possible nursery size to counter the effect of the size of the *live set*.

In contrast to the experiences with imperative/object-oriented languages, the measurements given here collectively indicate that a larger nursery reduces garbage collection time (as expected), but also increases mutator time, which is not expected.

Nursery Size and Cache Misses. The explanation for the increase in mutator time with larger nurseries becomes clearer when we investigate the GHC garbage collection algorithm. GHC uses a *copying algorithm* where, after every collection, the live data is compacted, and is consequently fresh in the cache if the nursery size is small enough. When the nursery size gets close to or larger than the cache size, more allocations will trigger cache misses, so making the mutator slower. We tested precisely how this size relates to cache locality by recording the cache misses for each nursery size configuration. Figure 4 plots the percentage of cache misses against the nursery size for different executions of `binary-trees`. We also show the mutator time. We can conclude that the nursery size directly affects cache behaviour; the larger the nursery, the more cache misses are detected. One further observation provides strong reasons to conclude that cache locality is the main contributor to mutator performance: the best runtimes are obtained for nursery sizes which are smaller than the cache size (4MB) and their value is practically constant. Once the nursery is of a comparable size to the cache, the mutator time is significantly increased, but it subsequebtly only changes slightly with a further increases in the nursery size. In summary, having a nursery size that is small enough to fit in the cache is crucial for good mutator performance. The exact size is, however, almost irrelevant. Similar behaviour occurs if the nursery is too big, where the actual size of the nursery also does not make a significant difference in the runtime. The collection time, however, is inversely proportional to the nursery size. This is intuitively obvious: *as the time between successive collections increases, it also becomes more likely that objects are no longer alive and so less garbage needs to be collected.*

3.4 Phase analysis of `binary-trees`

We next study the effect of various nursery sizes in the different execution phases of a memory-irregular program, such as `binary-trees`. We have selected two executions which show radically different behaviour: one using a 1MB nursery size and another using a 64MB nursery. As shown in Figure 3, for a 1MB nursery, the mutator time dominates the execution time, whereas with a 64MB nursery, the garbage collection time dominates. In order to make it possible to compare two different executions with different elapsed times, we used the amount of *allocation* as a measure of execution progress, so that the same program state is maintained between different executions at the same allocation value, regardless of the collector algorithm or the specific performance obtained. We split the execution in the smallest common chunk (64MB) and calculated the accumulated runtime for each chunk. For a nursery size of 64MB, this runtime corresponds to the time between two successive garbage collections. For the 1MB nursery, the runtime value corresponds to the accumulated time the program expended in 64 periods of mutator and GC execution.

We provide two figures to show the relationship between nursery size and total execution time. Figure 5 shows the amount of copying against allocation for both nursery sizes in two overlapped plots. To explain this figure, we need to recall how our example `binary-trees` benchmark works. In the first part of the program, a large number of small trees are allocated. Because of the size of the individual trees, each tree quickly becomes garbage, and each GC involves copying a small amount of data in both cases. In the later phase of the execution, a smaller number of larger trees

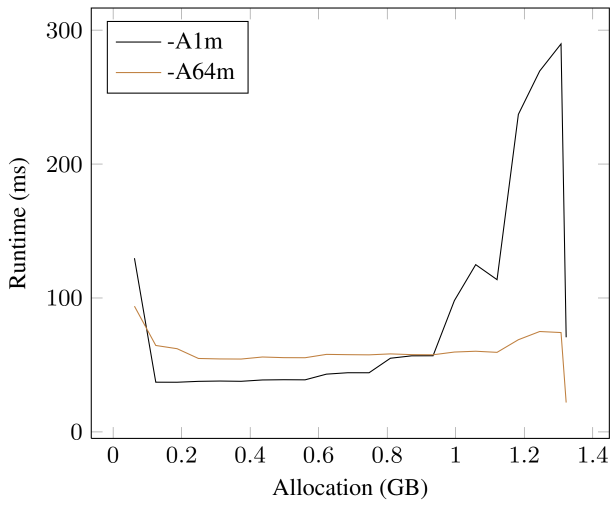

Figure 6. Runtime comparison of executions with different nursery size.

is allocated. The size of these trees is many times greater than a nursery of 1MB, forcing the garbage collector to copy the same objects multiple times. This is reflected on an increasing amount of copying in the graph. When using a nursery size of 64MB, the amount of copying is only slightly increased because no more than one tree is copied at each GC. We can therefore observe that, for `binary-trees`, there exist phases where it is beneficial to keep the nursery small, but there also exist phases where the nursery size should be larger in order to reduce the amount of data copying. This reveals the need for different nursery sizes in different phases of programs with irregular memory behaviour. Figure 6 shows the execution time that corresponds to each allocation point. We can observe how the amount of copying affects the execution time of the program. The graph has a very similar shape as the previous one, suggesting that the execution time is proportional to the amount of data that was copied in each GC. However, there is one further thing to note: *during the phase in which there is very little copying, the execution time with a small nursery (1MB) is better than that with a large nursery (64MB), while the reverse happens when the amount of copying increases.*

3.5 Preliminary Conclusions

To summarise, this section has shown several interesting points:

- the "proven" heuristic of increasing the nursery size to improve performance only works for situations where the cache does not have a major impact on the mutator time (Figure 3);
- furthermore, in programs with irregular memory behaviour (Figure 2), the nursery size plays a crucial role in the overall performance and the wrong choice of nursery size can significantly increase execution time (Figure 3);
- having the same static nursery size for the whole duration of the program may be suboptimal, as the program may have phases where different nursery sizes may be beneficial (Figure 6);
- in phases where little data is copied, using a small nursery size will be more efficient; this is because a small nursery size yields better cache behaviour, which is especially important for program phases where the mutator time dominates (Figure 5);
- in phases where a large amount of data needs to be copied, there is no point in trying to optimise the cache behaviour to improve the mutator time, because this will only have a minor impact on the total execution time, and a larger nursery size can effectively reduce GC time (Figures 5 and 6).

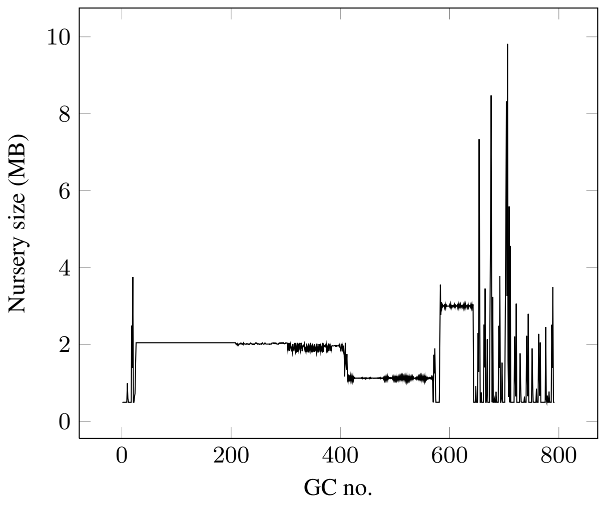

Figure 7. Nursery size with GHC's -H setting.

We will use these insights in the next section to develop better algorithms for dynamically changing the nursery size.

4. Algorithms for Fine-tuning the Nursery Size

In this section, we first analyse the behaviour of the existing GHC nursery resizing algorithm, and then propose two new algorithms for nursery resizing: *TAA*$^+$, a modest extension of the existing algorithm proposed by Anderson [2]; and *SLR*, a novel algorithm for dynamic nursery resizing. Finally, we compare the effects of these algorithms on the `binary-trees` benchmark.

4.1 Nursery Resizing in GHC

As described in Section 2.2, GHC uses an implementation of Appel's [3] algorithm for dynamic nursery resizing. The user can manually provide hints about the total heap size, or it can be dynamically adjusted by the runtime system at each major garbage collection. In the remainder of the paper, we will assume that the heap size is dynamically adjusted. Figure 7 illustrates the use of this algorithm for the `binary-trees` benchmark. It shows the nursery sizes, as chosen at each garbage collection. We can observe that the heap size is increased several times: once at the beginning of the execution, when the first tree is allocated, and then several times towards the end, once the sizes of the trees increase considerably. We can also observe that the nursery size is increased each time the heap size is recalculated, but that then it is reduced over time if more objects are kept alive. The idea behind this is to give as much memory as possible to the nursery, with the constraint that it should not use more memory than is currently in use by live objects. The drawback of this approach lies in using *all* the available space, even if this results in a loss in performance, due to bad cache behaviour. On the other hand, the size of the nursery is limited by the amount of live data, whereas in some cases increasing the nursery size beyond this limit would give better results. In the case of the `binary-trees` benchmark, we can observe the harmful effects of this policy towards the end of the execution, where the nursery size alternatively jumps from moderately large values to smaller ones due to the increased amount of live data.

4.2 Time-based Algorithms (TAA and TAA$^+$)

TAA. Dynamic nursery resizing for functional programs has previously been studied by T. A. Anderson et al. [2], who try to optimise the execution of a program by selecting the nursery size that results in *the best performing mix of PN (private nursery) and non-PN data in cache*. They use the nursery garbage collection time as a measure of the cache performance. To compare the GC performance for different nursery sizes, they define a time-per-byte metric (TPBM_n). This metric measures the time taken by the n^{th} garbage collection divided by the size of the nursery for that collection (S_n)[3]. This metric, therefore, accounts both for time it takes to do garbage collection and for frequency of collections. For example, if for two nursery sizes it takes the same amount of time to do garbage collection, TPBM will be smaller for larger nursery size, accounting for the fact that garbage collections with it are less frequent. The target of the algorithm is to reduce the TPBM as much as possible by either increasing or decreasing the nursery size. The algorithm works like a local search. It has two steps: first, it tries to adjust the nursery size quickly by either halving or doubling its value following each garbage collection. Next, a binary-search-like fine-grained adjustment between the last choices of nursery size is used to find its optimal value. The pseudocode of these two functions is shown in Figure 8. The first mode, outlined in `fast_update()`, starts with an initial nursery size S_1 already set. Then, following a number of garbage collections, TPBM_1 is calculated and a new nursery size S_2 is chosen by halving S_1. After the same number of GCs, TPBM_2 is calculated and compared to TPBM_1. If TPBM_2 is less than TPBM_1, the nursery size is halved again. This process is repeated until a nursery size S_n with a worse TPBM_n than S_{n-1} is found. At that point, the optimum nursery size must be between S_{n-2} and S_{n-1} or between S_{n-1} and S_n. To calculate the optimum value, `slow_update()` is called using the last three nursery sizes. This function calculates two new nursery sizes: S_x between the first two values and S_y between the last two. If $\mathsf{TPBM}_x < \mathsf{TPBM}_{n-1}$ then the search continues recursively with the values S_{n-2}, S_x and S_{n-1} as if they were the last three sizes. Alternatively, if S_y improves the results of S_{n-1}, the search continues recursively with the values S_{n-1}, S_y and S_n. If both values give worse results than S_{n-1}, then the search continues recursively with S_x, S_{n-1} and S_y. The algorithm terminates when S_{n-2} and S_n are too close to give substantially different results, and S_{n-1} is then selected as the optimal nursery size. Additionally, new TPBMs are collected as the program runs so that a change in performance initiates the algorithm again. In the remainder of the paper, we will denote this algorithm by *TAA*.

TAA$^+$. A key assumption for the *TAA* algorithm is that the collection time is a good measure of cache locality. We propose a simple modification to this algorithm, based on the conclusions of Section 3: instead of taking just the garbage collection time, we also add the mutator execution time immediately preceding that GC. We will denote this modified version as *TAA*$^+$. We evaluate both *TAA* and *TAA*$^+$ in Section 5.

4.3 Copying-based Algorithm (SLR)

One disadvantage of *TAA*/*TAA*$^+$ is that it reacts to changes in the program behaviour by guessing whether to increase or decrease the nursery, and then correcting the adjustment if the guess failed, or else continuing with it if it was right. This heuristic could work well for simple programs with regular memory behaviour. Our evaluation of the algorithm on several benchmarks reveals that this is not usually the case, however, and it needs to recalculate the

```
fun fast_update()
    $S_{n-2} = S_{n-1}$
    $S_{n-1} = S_n$
    $TPBM_n = GCTime_n/S_n$
    if $TPBM_n < TPBM_{n-1}$ then
        $S_n = S_n/2$
    else
        $S_n$ = slow_update($S_{n-2}$, $S_{n-1}$, $S_n$)
    end
    return $S_n$
end

fun slow_update($S_{n-2}$, $S_{n-1}$, $S_n$)
    if abs($S_n$- $S_{n-2}$) < threshold then
        return $S_{n-1}$
    end
    $S_x = (S_{n-1} + S_{n-2})/2$
    [. . . execution with nursery size $S_x$ . . . ]
    $TPBM_x = GCTime_x/S_x$
    if $TPBM_x < TPBM_{n-1}$ then
        return slow_update($S_{n-2}$, $S_x$, $S_{n-1}$)
    else
        $S_y = (S_n + S_{n-1})/2$
        [. . . execution with nursery size $S_y$ . . . ]
        $TPBM_y = GCTime_y/S_y$
        if $TPBM_y < TPBM_{n-1}$ then
            return slow_update($S_{n-1}$, $S_y$, $S_n$)
        else
            return slow_update($S_x$, $S_{n-1}$, $S_y$)
        end
    end
end
```

Figure 8. Pseudocode for T. A. Anderson's dynamic nursery resizing algorithm.

nursery size many times during the execution. In fact, we had to modify `slow_update()` to be able to abort its operation if a phase change was detected. This problem can be reduced to the fact that *TAA*/*TAA*$^+$ reacts to phase changes in the program by measuring TPBM only *after* the program behaviour has changed.

As shown earlier, we know that ratio of copying to nursery size can serve as an estimator to the execution performance of the program, because of the impact on cache locality of object liveness. Using this information, it follows that the best way to optimise the nursery size is not to find its "optimum" value, which changes very often, but to find the optimum value of the nursery size/live set ratio (SLR) and then to use this to calculate the nursery size. Given this target, we can use the same metric, TPBM, as a measure of the performance of a given SLR. We have, however, decided not to use the same search procedure because of the need for multiple special cases to handle phase changes and to restart its execution. Instead, we use a single step search that proceeds with increasingly smaller updates to the SLR until the delta change is below some threshold. The pseudocode for this new algorithm, SLR, is shown in Figure 9. The algorithm is seeded with an initial SLR_1 and $\mathsf{update\_factor}_0$, the value used to update the SLR. The update direction is calculated in the first two iterations as previously described for *TAA*/*TAA*$^+$.

Compared to *TAA*/*TAA*$^+$, the SLR algorithm shares some of the same problems: the SLR has to be either increased or decreased and there can be an initial slowdown if the direction is wrong. Additionally, the amount of copying is measured after each garbage collection and cannot be obtained beforehand. However, this algo-

[3] In fact, the metric is measured over a number of collections. We will retain the notation to ease the presentation of the algorithm.

```
fun resize()
    $\mathsf{TPBM}_{n-1} = \mathsf{TPBM}_n$
    $\mathsf{TPBM}_n = (\mathsf{MUTTime}_n + \mathsf{GCTime}_n)/\mathsf{S}_n$
    if abs($\mathsf{TPBM}_n$- $\mathsf{TPBM}_{n-1}$) < threshold then
        $\mathsf{update\_factor} = \mathsf{update\_factor}_0$ // reset value
        $\mathsf{SLR}_n = \mathsf{SLR}_{n-1}$
    else
        // if performance is worse, reverse
           update direction
        if $\mathsf{TPBM}_{n-1} > \mathsf{TPBM}_n$ then
            $\mathsf{update\_factor} = -0.9 \times \mathsf{update\_factor}$
        end
        $\mathsf{SLR}_n = \mathsf{SLR}_{n-1} \times (1 + \mathsf{update\_factor})$
    end
    return $\mathsf{SLR}_n \times \mathsf{copied}_n$
end
```

Figure 9. Pseudocode for the new copying-based dynamic nursery resizing algorithm.

GC configuration	Speedup	Execution Time
default	1.00	7.70s
-A2m	1.44	5.34s
-A8m	1.69	4.55s
-A64m	1.78	4.32s
-H	1.38	5.78s
TAA	1.72	4.47s
TAA$^+$	1.79	4.30s
SLR	1.96	3.92s

Table 1. Speedups of different nursery size settings/algorithms against the default fixed size of 500KB.

rithm should adapt better to changes in memory usage because, between each period in which a new TPBM is calculated, the nursery size is updated to maintain its relation to the amount of copying constant, rather than keeping a fixed value. Table 1 summarises the speedups that we obtain for `binary-trees`, using the default setting of 500KB nursery size as a baseline. We compare the original TAA algorithm, our modified TAA$^+$ algorithm, and the new copying-based SLR algorithm against three static configurations with differing constant nursery sizes (-A2m, -A8m, -A64m) and one builtin dynamic sizing algorithm (-H), where the garbage collector takes the size of the heap at the previous GC to be the new nursery size. It is clear that all three new dynamic algorithms give significant performance benefits over the default configuration or the builtin dynamic algorithm (-H). Similar performance to TAA/TAA$^+$ can only be achieved using large static nursery sizes (-A8m, -A64M). Even then, SLR gives the best overall performance, reducing the total execution time by almost half its original value.

5. Evaluation

We have implemented TAA, TAA$^+$ and SLR dynamic nursery resizing algorithms in the GHC compiler, and evaluated them against the default GHC garbage collection mechanism (described in Section 2.2) on the *nofib* benchmark suite [11]. *nofib* is the standard Haskell benchmark suite and is included in the GHC repository. It comprises over 60 benchmarks, ranging from synthetic microbenchmarks to real programs. We have modified the configuration of most benchmarks in order to increase their execution time, so that more substantial measurements could be obtained. Our modified version of the benchmark suite can be found at `http://github.com/hferreiro/nofib`. For each benchmark, we compiled the code using -O2, ran each nursery-resizing algorithm five times, and recorded the mean *total* execution times (i.e. including the costs of both GC *and* mutation). This avoids biasing our results towards algorithms that improve GC time at the expense of normal execution time, or *vice-versa*. Table 2 summarises our results, classifying them, for each algorithm, into positive (where the mean execution time is better with a given algorithm than with the default GHC mechanism), negative (where the mean execution time is worse than with the default GHC mechanism) and unaffected benchmarks (where the mean execution time under a given algorithm is within 5% of the default GHC mechanism). In the majority of cases ($>$ 50%), all the resizing algorithms give similar results to the default algorithm. We can clearly see, however, that TAA$^+$ and SLR have a positive impact in many more cases than TAA (30.6% and 33.9%, respectively versus 19.4%), and that SLR is as good as or better than the default GHC mechanism in 95.2% cases. *On balance, we can conclude that SLR is a better default than the current standard GHC mechanism.* It is clear that the unmodified TAA should be used with caution, however: it has a negative impact on more cases than those for which it gives a positive impact. Figure 10 tabulates the results with the greatest deviations from the default scheme. We now analyse the performance of the individual algorithms.

Performance	TAA	TAA$^+$	SLR
unaffected	36 (58.0%)	32 (51.6%)	38 (61.3%)
positive	12 (19.4%)	19 (30.6%)	21 (33.9%)
negative	14 (22.6%)	11 (17.7%)	3 (4.8%)
mean diff	**-2.04%**	**-6.08%**	**-9.30%**

Table 2. Proportion of affected benchmarks and mean difference in execution time over all benchmarks compared with the default GHC mechanism for each nursery resizing algorithm. "Positive" ("negative") denotes the number and percentage of benchmarks whose total execution time is reduced (increased) by more than 5% when using the specified algorithm.

Benchmark	Time (s)	TAA	TAA$^+$	SLR
gcd	7.6	-0.5%	+0.2%	+0.3%
hpg	7.0	-0.3%	+0.6%	+1.3%
k-nucleotide	7.7	-0.8%	-0.6%	-0.2%
linear	7.1	-0.1%	+2.2%	+1.9%
multiplier	6.7	-53.7%	-40.6%	-51.2%
queens	18.6	-0.1%	+0.0%	+0.0%
solid	9.5	-12.5%	-11.9%	-0.2%
wang	7.5	-32.5%	-16.9%	-7.3%

Table 3. Benchmarks for which the TAA algorithm gives the best execution time. The TAA, TAA$^+$ and SLR columns record the percentage change in execution time for each of the algorithms compared to the GHC default (column 2).

5.1 Performance of TAA

Table 3 shows the benchmarks for which the TAA algorithm gives the best total execution time. For most of these benchmarks, all four algorithms have similar performance. This is because only a small amount of data is copied during the garbage collection, which means that improving the garbage collection time does not

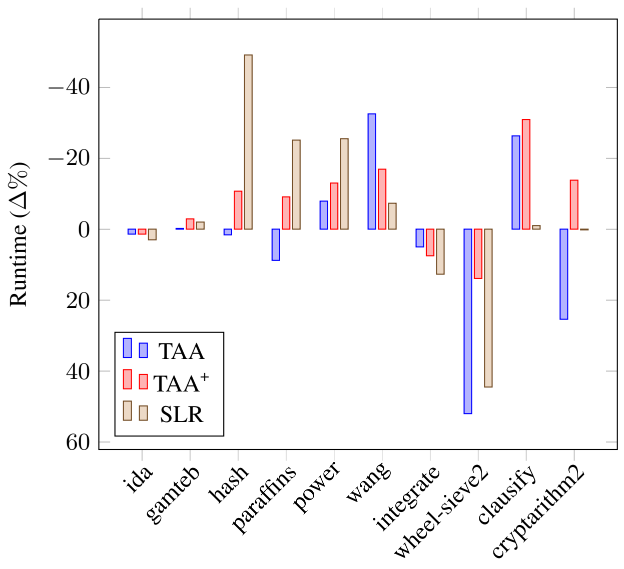

Figure 10. Results from *nofib* showing greatest deviations.

Benchmark	Time (s)	TAA	TAA+	SLR
atom	7.6	-54.1%	-56.7%	-52.1%
boyer	6.9	-21.7%	-31.5%	+0.2%
calendar	6.9	+28.4%	-3.1%	+0.4%
circsim	6.3	-23.7%	-35.6%	-11.5%
clausify	6.7	-26.3%	-30.9%	-1.0%
cryptarithm2	9.4	+25.4%	-13.8%	+0.2%
fft2	7.2	-12.7%	-18.5%	-12.8%
gamteb	7.0	-0.2%	-2.9%	-2.0%
gc_bench	6.9	-30.2%	-35.5%	-22.7%
mandel2	6.4	-0.2%	-0.3%	+0.8%
parstof	7.7	+0.7%	-0.7%	+0.1%
primes	7.1	-22.4%	-28.2%	-25.3%
sorting	7.4	-3.4%	-5.3%	-3.8%
transform	7.4	+1.9%	-25.2%	-6.0%

Table 4. Benchmarks for which the TAA+ algorithm gives the best execution time. The TAA, TAA+ and SLR columns record the percentage change in execution time for each of the algorithms compared to the GHC default (column 2).

have a significant impact on the total execution time. In such cases, it makes sense to keep the nursery small (which is also what `ghc-gc-tune` suggests), since increasing the nursery size will have a negative effect on the mutator time. Fortunately, all of the dynamic nursery-resizing algorithms that we consider are able to detect this situation and maintain small nursery sizes. For the `solid` benchmark, TAA and TAA+ show almost identical performance. Similarly, for the `multiplier` benchmark, the overall performance is almost the same for TAA and SLR. There is only one benchmark, `wang`, for which the TAA algorithm performs significantly better. This indicates that TAA is probably not worth using as a default mechanism.

5.2 Performance of TAA+

Table 4 shows the benchmarks for which the TAA+ algorithm gives the best execution time. Disregarding those cases where the total execution times are similar for all the algorithms, the reason why TAA+ outperforms the SLR algorithm involves the effect of an extremely small live set and its interplay with generational garbage collection. We will focus on the `clausify` benchmark, which has a very specific behaviour. For this benchmark, the live set ranges from 30KB to 100KB. In a generational collector, minor collections will copy an increasing amount of data, and major collections will reset the copied data to the live set every few collections. Because of its small size and the fact that it remains stable, if we manually select a nursery value that is 2MB or greater, we reduce the amount of copied data considerably and thus obtain a 30% reduction in the execution time. This behaviour is easily discovered by TAA+ because it optimizes the nursery size directly. With SLR, the algorithm starts increasing the nursery size through the size/copied data ratio, but following every few minor collections, either a major collection is triggered or we enter a phase with an especially small amount of copying. Even if the difference in copied memory is small, since `clausify` has a very small live set, the reduction can be an order of magnitude in scale. At this point, SLR will select a very small nursery size, so obtaining worse results and forcing the algorithm to reverse its direction. A new cycle then commences where these events are repeated. After less than a third of the execution, the nursery size will constantly be kept at the minimum value of 500KB. Figure 11 shows the selected nursery size and the amount of copying that leads to that choice at the very beginning of the execution.

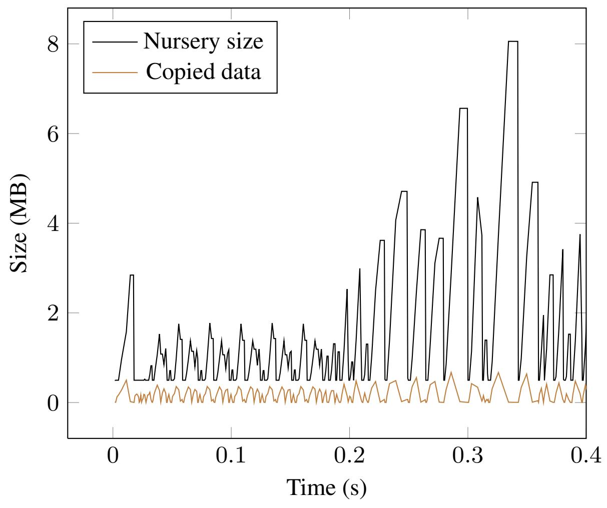

Figure 11. SLR behaviour for `clausify`.

5.3 Performance of SLR

Figure 5 shows the benchmarks for which the SLR algorithm gives the best execution time. As before, most of these benchmarks share some characteristics. In particular, the only option to obtain similar results to SLR when using a fixed-size nursery is to use very large initial nursery sizes. Two of these programs, `hash` and `mutstore1` appear to be affected by the time at which GC occurs, since their memory usage varies upwards and downwards as we increase the nursery size. In this case, it would be very difficult for TAA/TAA+ to be able to escape a local maximum in nursery size. We also note that some of the best improvements in execution time compared to the default GHC mechanism are obtained when using SLR (88.5% for `mutstore2` and 50.5% for `binary trees`).

Benchmark	Time (s)	TAA	TAA$^+$	SLR
anna	7.9	+4.0%	+1.2%	-0.2%
binary-trees	7.7	-47.0%	-45.6%	-50.5%
comp_lab_zift	7.7	-1.1%	+1.0%	-2.8%
constraints	18.3	+0.7%	-0.2%	-0.9%
event	7.0	+1.3%	+1.0%	-0.9%
fibheaps	7.0	+13.2%	+15.6	-5.4%
fulsom	6.1	-0.6%	-5.3%	-7.0%
gen_regexps	7.1	+10.8%	+8.4%	-17.6%
genfft	6.4	+37.6%	+5.3%	-0.1%
hash	10.2	+1.6%	-10.7%	-49.1%
lcss	7.3	-0.3%	-0.3%	-14.8%
mutstore1	6.4	+12.6%	+4.3%	-20.3%
mutstore2	7.6	-64.9%	-63.5%	-88.5%
paraffins	8.1	+8.8%	-9.1%	-25.1%
pic	7.3	+1.6%	-1.5%	-9.1%
power	7.4	-7.9%	-13.0%	-25.5%
reverse-complement	8.7	+10.0%	+6.4%	-3.8%
rewrite	7.3	+0.1%	+0.1%	-2.4%
scs	7.0	+8.2%	+5.3%	-3.9%
simple	7.4	+4.7%	+5.9%	-5.6%
wave4main	7.0	+0.4%	-0.1%	-5.8%
wheel-sieve1	7.9	-1.3%	+0.5%	-3.0%
x2n1	7.3	+4.6%	+0.2%	-14.9%

Table 5. Benchmarks for which the SLR algorithm gives the best execution time. The TAA, TAA$^+$ and SLR columns record the percentage change in execution time for each of the algorithms compared to the GHC default (column 2).

Benchmark	Time (s)	TAA	TAA$^+$	SLR
bernouilli	6.0	+1.7%	+2.0%	+1.5%
fasta (check r4)	8.7	+4.1%	+2.3%	+3.7%
fft	12.2	+17.5%	+18.6%	+4.2%
ida	7.3	+1.4%	+1.4%	+3.0%
integer	7.0	+12.8%	+0.1%	+3.4%
integrate	8.0	+5.0%	+7.5%	+12.7%
kahan	7.6	+2.6%	+0.1%	+1.4%
knights	7.5	+1.8%	+0.3%	+2.3%
mandel	8.4	+7.4%	+1.5%	+0.3%
n-body	7.3	+0.0%	+1.1%	+1.2%
pidigits	6.9	+12.4%	+28.2%	+0.8%
sphere	7.5	+2.3%	+13.3%	+19.5%
tak	7.7	+0.0%	+0.0%	+0.0%
wheel-sieve2	5.6	+52.0%	+13.9%	+44.5%

Table 6. Benchmarks for which all the nursery-resizing algorithms are worse than the default GHC mechanism. The TAA, TAA$^+$ and SLR columns record the percentage change in execution time for each of the algorithms compared to the GHC default (column 2).

5.4 Benchmarks with worse performance with TAA, TAA$^+$ and SLR

Finally, Table 6 shows the benchmarks for which all three advanced nursery-resizing algorithms yield worse total execution times than the default GHC mechanism. In most cases (29 out of 42, or 69%), the difference in execution times is within 5% of the execution time with the default GHC mechanism. However, there are five benchmarks where at least one of the algorithms is at least 10% worse than the default (`fft`, `integer`, `integrate`, `pidigits`, and `sphere`) , and one (`wheel-sieve2`) for which all three algorithms perform quite badly (between 13.9% and 52% worse than the default). These benchmarks can be classified into two sets: those for which GC involves a lot of copying and those with very little live data. While there no possibility of any improvement for the second set, the problem with the first set is that the live set is very large, so that increasing the nursery size has very little effect on the collection time. However, increasing the nursery size causes a significant increase in mutator time. This also means that manually setting the nursery size will incur similar penalties to the automatic sizing algorithms.

5.5 Summary of the Experiments

Although there are cases where the dynamic sizing algorithms do not give any improvement in execution time, our evaluation has shown that there are many examples where they significantly reduce the execution time. The SLR algorithm seems to be the best default option for GHC: 60 out of 63 benchmarks in the *nofib* suite have performance that is similar to or better than the default GHC algorithm, and only 3 have notably worse execution times ($> 5\%$). Some results were significantly better, yielding *performance improvements of up to 88.5% or up to 8.7$\times$ faster than the original program*. Significantly, these are improvements to the total execution time, including the mutator time, and not merely improvements in the GC time. Furthermore, SLR generally gives better execution time than TAA$^+$, except in a few cases that eaily can be characterised by common behaviour, and that can therefore be recognised by a slight modification of the SLR algorithm. This agrees with our predictions from the analysis in Section 4.3. As shown in Table 2, the mean improvement in execution time for all our benchmarks is higher with SLR (9.30%) than with TAA (2.04%) and TAA$^+$ (6.08%). Also, while there are cases where automatically discvoerd optimal static nusery size (using the `ghc-gc-tune` tool) gives better execution times than any of the algorithms we have considered, this is not generally a practical option. As described above, deriving the best nursery size can be very time consuming (especially for large programs), since it involves running the program many times with different parameters. In contrast, SLR is a fully automatic algorithm, that does not rely on any profiling. Finally, the SLR algorithm is relatively simple and easy to implement as part of a runtime system. For these reasons, we consider SLR to be a good conservative improvement over the current default setting for GHC, and also to be a good algorithm for nursery-resizing in runtime systems for functional languages that undertake a large number of memory allocations.

6. Related Work

Most generational garbage collectors use two generations: a nursery space and a mature space. Appel's garbage collector [3] dynamically tunes the size of the nursery space to be the size of the free space in the heap. It divides the nursery space into an allocation area and a reserved area. Until the nursery becomes larger than some predetermined value, only nursery GCs are performed. At this point, a GC is performed on the whole heap. Velasco et al. [14] aim to improve this strategy by allocating more of the nursery space to the allocation area than to the reserved area. After each collection, they use one of their two strategies to decide on the proportion of nursery space to give to the allocation area: *average*, where the average ratio between the data copied and the allocation area size over the last few collections is given to the allocation area; and *worst*, where the worst ratio is given to the allocation area. This technique may seem similar to ours. However, because they work with fixed-size generations, the proportions are only used to make it possible to assign more than half of the nursery to the allocation area, without running out of memory when copying live data to the remaining mature space.

The most closely related approach to our TAA$_+$ and SLR algorithm is, of course, Anderson et al.'s TAA algorithm [2]. We have shown that our new algorithms consistently outperform TAA, and also that they benefit more cases than TAA. In the best cases, we obtain significant performance improvements ($> 50\%$). Guan et al. [6] consider three different policies for dynamic nursery resizing in a HotSpot generational garbage collector: a *GC Ergonomic Policy* (measuring the GC pause time and throughput in each collection and adjusting the nursery size accordingly), a *Fixed Ratio Policy* (maintaining the same ratio between the nursery and the mature space) and a *Heap Availability Policy* (similar to Velasco's approach). They conclude that in most of the cases, the *Heap Availability Policy* yields the best results. Finally, White et al. [15] propose a strategy for adaptive resizing of the whole heap based on control theory. They show that heap sizing can be formulated as a control problem and that a standard controller program can be applied, yielding good improvements for some benchmarks.

7. Conclusions and Future Work

In this paper, we have shown that the performance of the garbage collector, and in particular the way in which the size of the nursery area where new objects are allocated is chosen, can have a significant impact on the performance of functional programs. We have studied the problem of choosing the appropriate nursery size during the execution of such programs, focusing on the GHC compiler for the purely functional language Haskell, and on programs with *irregular* memory behaviour (i.e. where there is a high variation in the amount of live data during the program's lifetime). We have established a relation between the nursery size and the execution time of the program as the interplay of cache locality and the amount of data copied during garbage collection. Subsequently, we have presented two novel algorithms for dynamic nursery resizing which try to achieve a balance between cache locality on one hand and the frequency and time costs of garbage collections on the other hand. We have evaluated the proposed algorithms on an extensive test suite and provided details about their performance in relation to their runtime behaviour. We have demonstrated that, with our algorithms, we can achieve up to 88.5% performance improvement or up to 8.7$\times$ speedup for some benchmarks than by using GHC's default settings or its own nursery resizing algorithm, and that in all but a few cases, our algorithm is at least as good as, or better than, the current default settings. We were also able to identify what characteristics of a program make it unsuitable for our dynamic resizing algorithms. On average, we were able to achieve a speedup of 9.3% *in total performance* over a suite of over 60 benchmarks compared with the highly-optimising production GHC Haskell compiler. Given the relative simplicity of our technique, and the fact that GHC is already highly tuned, this is a remarkably good result.

As future work, we aim to quantify the memory irregularity of programs, by defining a metric that describes precisely *how irregular* the memory behaviour is, and then to relate that to the improvements that we obtain with our algorithms. We also aim to study additional factors that can influence the nursery size and garbage collection, such as the amount of data that is actually accessed (rather than just alive) at different points in the program execution. In the longer term, we aim to study garbage collection for parallel programs, since the performance of these programs is very often limited solely by the performance of garbage collection.

References

[1] A. Ahmad and H. DeYoung. Cache Performance of Lazy Functional Programs on Current Hardware. Technical report, Carnegie Mellon University, 2009.

[2] T. A. Anderson. Optimizations in a Private Nursery-based Garbage Collector. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, pages 21–30, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0054-4. . URL `http://doi.acm.org/10.1145/1806651.1806655`.

[3] A. W. Appel. Simple Generational Garbage Collection and Fast Allocation, 1989.

[4] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and Realities: The Performance Impact of Garbage Collection. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04/Performance '04, pages 25–36, New York, NY, USA, 2004. ACM. ISBN 1-58113-873-3. . URL `http://doi.acm.org/10.1145/1005686.1005693`.

[5] A. M. Cheadle, A. J. Field, S. Marlow, S. L. P. Jones, and R. L. While. Exploring the Barrier to Entry - Incremental Generational Garbage Collection for Haskell. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 163–174, New York, NY, USA, 2004. ACM. ISBN 1-58113-945-4. . URL `http://doi.acm.org/10.1145/1029873.1029893`.

[6] X. Guan, W. Srisa-an, and C. Jia. Investigating the Effects of Using Different Nursery Sizing Policies on Performance. In *Proceedings of the 2009 International Symposium on Memory Management*, ISMM '09, pages 59–68, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-347-1. . URL `http://doi.acm.org/10.1145/1542431.1542441`.

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

[8] S. Marlow. Haskell 2010. Language Report, 2010. URL `http://www.haskell.org/onlinereport/haskell2010`.

[9] S. Marlow and S. Peyton Jones. Multicore Garbage Collection with Local Heaps. In *Proceedings of the 10th International Symposium on Memory Management*, ISMM '11, pages 21–32, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0263-0. . URL `http://doi.acm.org/10.1145/1993478.1993482`.

[10] S. Marlow, T. Harris, R. P. James, and S. Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, pages 11–20, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-134-7. . URL `http://doi.acm.org/10.1145/1375634.1375637`.

[11] W. Partain. The nofib Benchmark Suite of Haskell Programs. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 195–202, London, UK, UK, 1993. Springer-Verlag. ISBN 3-540-19820-2. URL `http://dl.acm.org/citation.cfm?id=647557.729913`.

[12] S. Peyton Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler. The Glasgow Haskell Compiler: a Technical Overview. In *Proc. JFIT '93, Keele*, Mar. 1993.

[13] D. Stewart. The ghc-gc-tune package. `https://hackage.haskell.org/package/ghc-gc-tune`, 2015.

[14] J. M. Velasco, A. Ortiz, K. Olcoz, and F. Tirado. Dynamic Management of Nursery Space Organization in Generational Collection. In *Proceedings of the 8th Annual Workshop on Interaction between Compilers and Computer Architectures*, INTERACT '04, pages 33–40, Los Alamitos, CA, USA, 2004. IEEE Computer Society. ISBN 0-7695-2061-8. .

[15] D. R. White, J. Singer, J. M. Aitken, and R. E. Jones. Control theory for principled heap sizing. In *Proceedings of the 2013 International Symposium on Memory Management*, ISMM '13, pages 27–38, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2100-6. . URL http://doi.acm.org/10.1145/2464157.2466481.

[16] P. R. Wilson, M. S. Lam, and T. G. Moher. Caching Considerations for Generational Garbage Collection. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, LFP '92, pages 32–42, New York, NY, USA, 1992. ACM. ISBN 0-89791-481-3. . URL http://doi.acm.org/10.1145/141471.141500.

[17] Y. Zhao, J. Shi, K. Zheng, H. Wang, H. Lin, and L. Shao. Allocation Wall: a Limiting Factor of Java Applications on Emerging Multi-core Platforms. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 361–376, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0. . URL http://doi.acm.org/10.1145/1640089.1640116.

[18] B. Zorn. The Effect of Garbage Collection on Cache Performance. Technical Report CU-CS-528-91, University of Colorado, Boulder, May 1991.

A. Binary Trees source code

```
import System.Environment
import Data.Bits
import Text.Printf

--
-- an artificially strict tree.
--
-- normally you would ensure the branches are lazy,
-- but this benchmark requires strict allocation.
--
data Tree = Nil | Node !Int !Tree !Tree

minN = 4

io s n t = printf "%s of depth %d\t check: %d\n" s n t

main = do
  n <- getArgs >>= readIO . head
  let maxN     = max (minN + 2) n
      stretchN = maxN + 1

  -- stretch memory tree
  let c = check (make 0 stretchN)
  io "stretch tree" stretchN c

  -- allocate a long lived tree
  let !long    = make 0 maxN

  -- allocate, walk, and deallocate many binary trees
  let vs = depth minN maxN
  mapM_ (\((m,d,i)) -> io (show m ++ "\t trees") d i) vs

  -- confirm the the long-lived binary tree still exists
  io "long lived tree" maxN (check long)

-- generate many trees
depth :: Int -> Int -> [(Int,Int,Int)]
depth d m
    | d <= m    = (2*n,d,sumT d n 0) : depth (d+2) m
    | otherwise = []
  where n = 1 `shiftL` (m - d + minN)

-- allocate and check lots of trees
sumT :: Int -> Int -> Int -> Int
sumT d 0 t = t
sumT  d i t = sumT d (i-1) (t + a + b)
  where a = check (make i    d)
        b = check (make (-i) d)

-- traverse the tree, counting up the nodes
check :: Tree -> Int
check Nil          = 0
check (Node i l r) = i + check l - check r

-- build a tree
make :: Int -> Int -> Tree
make i 0 = Node i Nil Nil
make i d = Node i (make (i2-1) d2) (make i2 d2)
  where i2 = 2*i; d2 = d-1
```