Kindergarten Cop: Profiling-based Dynamic Nursery Resizing for GHC

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

Vladimir Janjic David Castro
Kevin Hammond
School of Computer Science,
University of St Andrews
{vj32,dc84}@st-andrews.ac.uk,
kh@cs.st-andrews.ac.uk

Abstract

In this paper, we will describe a new method for dynamic nursery resizing during the execution of Haskell programs under GHC. Our method is novel in that it relies on the memory profile of the program being run, as recorded in previous runs of the same program. It will, therefore, be able to make informed decisions about the nursery size, based on the expected amount of live and accessed data in each phase of the program execution. We will present the evaluation of our method on a set of synthetic and realistic benchmark programs.

1. Introduction

Compared to traditional imperative programming, lazy functional programming offers many benefits to programmers, such as very-high level of abstraction, referential transparency, transparent use of infinite data-structures and implicit sharing of data. However, we know that nothing comes for free. One of the main problems when it comes to lazy functional programs is their runtime behaviour –in particular, they tend to do many more memory allocations than equivalent imperative programs, yielding a need for frequent garbage collections. Garbage collection (GC) becomes a real issue when we start parallelising functional programs. While by no means easy, parallelising actual computation done by a program is under control of programmer. However, programmer can do very little to parallelise GC, as it is completely under control of runtime system. Ultimately, the performance of the GC can be a significant limiting factor for achieving a good performance in both sequential and parallel functional programs. Therefore, a mechanism is needed that reduces the amount of time a program spends in garbage collection while not significantly increasing the time it spends in computation.

In a generational GC, one of the most used GC algorithms, a standard mechanism for reducing GC time is tuning the size of the nursery (the space where young objects are allocated), as most of the GC time is spent collecting the young generation. A large nursery results in fewer GCs, but may also ruin the cache behaviour of a program. A small nursery results in good cache behaviour, but frequent GCs that can, depending on the amount of live data that needs to be copied, be quite expensive. Therefore, some compromise is needed. Current trends [**? ? ?**] focus on dynamically changing the nursery size while the program is running, and the decisions are usually made based on the maximum total heap size as indicated by the user and the various stats about previous GCs in the same program run (such as the amount of live data). The main drawback of these methods is that they are done without taking into account foreknowledge about the program itself, which can be obtained using profiling.

In this paper, we will present a new method for dynamically changing the nursery size in Haskell programs. Our method will be novel in that it uses the program's memory profile from previous runs to estimate its memory behaviour. In this way, if we know in advance what stage the program is entering (e.g. in terms of how much data will be allocated, how much of it will be survive the next GC and how much will actually be accessed), we can make informed decisions about nursery sizing *before* the program enters that stage, rather than as a response to programs behaviour *after* it enters the stage. Our method will also focus on increasing the potential for parallelisation of programs, rather than reducing their sequential runtime, which means that we will make some nursery sizing decisions that may be considered bad in purely sequential settings. We will also present an implementation of our method in the state-of-the-art GHC Haskell compiler, an evaluation of how it affects the program's parallel potential and performance, and a comparison with the state-of-the-art methods for dynamically changing the nursery size.

The specific research contributions of this paper will be:

- we will describe a novel algorithm for dynamically resizing the nursery, based on a previous profile of the program;
- we will describe the implementation of our algorithm in the GHC Haskell compiler and
- we will present the evaluation of our method on a set of benchmarks and realistic Haskell programs.

2. Background

In the rest of the text, we will use the following terminology. In a generational GC, the *nursery* or the *allocation area* is the memory area in the heap where new objects are allocated. It is part of the young generation. When the nursery is full, a *minor GC* is triggered, which only affects the young generation. During GC, the surviving objects are promoted to the next generation. Sometimes,

, .

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

objects from the nursery are promoted to the young generation prior to being promoted further, in order to avoid a problem known as premature promotion. When the occupancy of older generations surpass a given limit, a GC for that generation and all of the younger ones is triggered. A *major GC* is a GC where all of the generations are collected.

2.1 GHC

The Glasgow Haskell Compiler is a state-of-the-art compiler and parallel runtime system for the pure lazy functional language Haskell [**?**]. It achieves great flexibility by using a lightweight thread model, where multiple logical Haskell threads are mapped into one single OS thread which runs concurrently with others.

2.2 Garbage Collection in GHC

GHC currently uses a generational, stop-the-world garbage collector [**?**]. There have been attempts to introduce concurrent and per-core garbage collectors [**?**], but they were abandoned.

By default, GHC uses two generations, a fixed allocation area size and a dynamic heap size. This behaviour can be changed via command-line flags, by specifying the number of generations, minimum size of the allocation area, minimum and maximum heap size and automatic heap sizing. If there is more than one generation and no heap size hints are provided, the size of the allocation area will remain fixed. On the other hand, in the case of automatic heap sizing or providing a size hint, the allocation area is dynamically adjusted after every GC. If the user specifies the maximum heap size with the `-H` runtime flag, then the nursery is resized according to the following formula:

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

where N is calculated as the amount of memory needed for the next GC, i.e. twice the amount of the current live data, and p is the percentage of live data allocated since the last GC and is also used to approximate its value in the future. Essentially, this formula results in the nursery that is as large as possible, with a boundary in the calculated total amount of needed memory, which is provided by the user with the `-H` parameter, or calculated at each major GC as twice the amount of live data. In practice, resizing the nursery in this way may be bad, because it does not take into account cache behaviour which requires a small nursery. Also, the nursery size may be increased even when the amount of copied data is small (meaning the the GC cost may be insignificant).

Our aim in this paper is to improve the formula for the new allocation area size after GC, so that it takes into account not only information about previous GCs in the same program run, but also information about the memory profile of the same program in previous runs. In this way, we can choose the best nursery size accounting for the future (rather than previous) behaviour of the program.

2.3 A Motivating Example

In order to test our hypothesis, we have written an example program that has two phases with very different memory behaviours. In the first phase, the survival rate of newly created objects is high, which means that the amount of live data is also high. In the second phase, the completely opposite memory behaviour occurs –most of the data becomes garbage soon after it is created, which means there is very little live data. The code of our program is shown below

```
long :: Int -> Int -> Int
long n m = sum (foldl1 (zipWith' (+)) l)
  where
    l = take n (repeat [1..m])
```

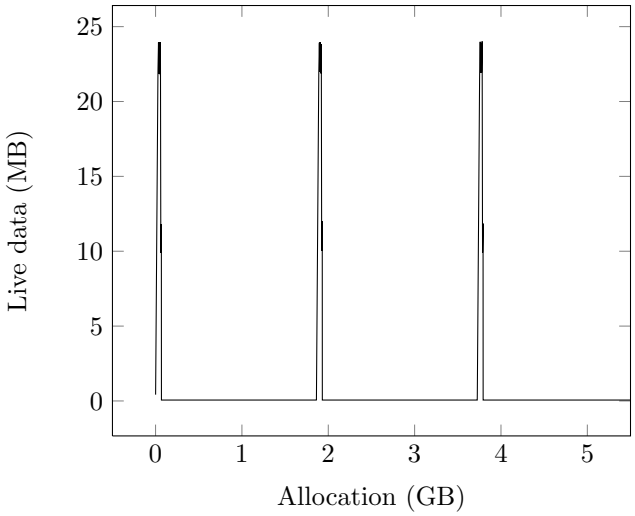

Figure 1. Object survival in an artificial example program.

```
f x = long 5 (mem*10000) + long (time*1000000) 1

main = print $ show $ take 10 (map f [1..])
```

This code runs 10 iterations of two calls to the function `long`. This function, for n lists of m elements $[x_{1i}, x_{2i}, \ldots, x_{mi}], 1 \leq i \leq n$ computes a new list $[y_1, y_2, \ldots, y_m]$, where $y_k = x_{k1} + x_{k2} + \cdots + x_{kn}$, and then sums this list. In order to achieve the memory behaviour we want, we force the evaluation of every sum before computing the final addition. In this way, the two lists being summed at any one time are always kept in memory until the addition of the resulting list. The function `f` calls the function `long` two times: the first time for a small number of very long lists, and the second time for a large number of small lists. `mem` and `time` are constants used to tune, respectivelly, the size and the amount of lists being summed.

Figure 1 shows the amount of live data against the total amount of allocated data for the first three iterations of function `f`. Note that the x axis in the figure is not the time from the beginning of the program execution, therefore it is not true that the program spends much less time in the phases that have a large amount of live data (large spikes in the figure). Actually, the amount of time that the program spends in each of the two phases is much bigger, because of the GC time.

To check the influence that the nursery size has on the program execution time, we have used the `ghc-gc-tune` tool to run our program with different nursery and heap sizes. If a nursery size is provided (`-A` GHC runtime flag), then the nursery has a fixed size for the whole program execution. On the other hand, if a minimum heap size (`-H` GHC runtime flag) is provided, the size of the nursery changes dynamically as described in Section 2.2. Figure 2 shows the execution times with different combinations of nursery and heap sizes. From the figure we can observe that program execution time is highly dependent on the nursery. We can also see that setting the nursery size to small values results in bad execution time, mainly due to too frequent GCs. Similarly, setting it to values too high, the reduced GC time cannot compensate the increase in mutator time, due to increasing cache misses. We tried using the automatic nursery size but it wasn't close to the numbers in the lower part of the graph.

Figure 3 was constructed to compare the runtime behaviour of different nursery sizes, and check the hypothesis that a dynamic

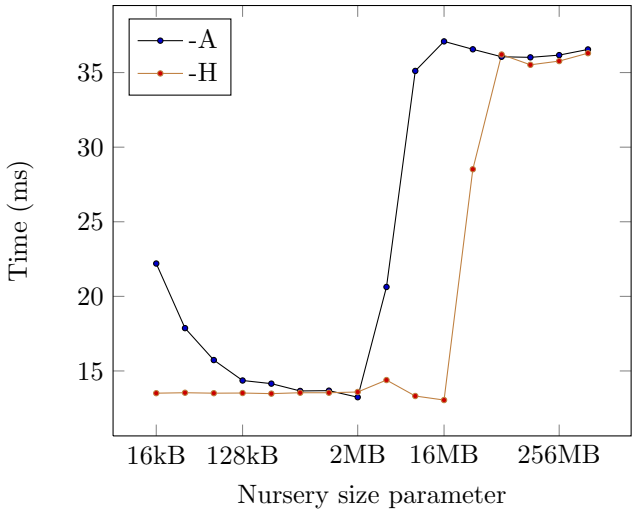

Figure 2. Effect of nursery size parameters.

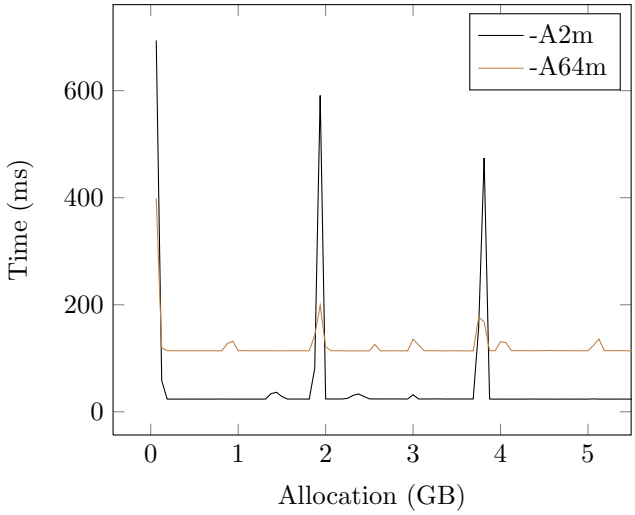

Figure 3. Comparison of different nursery sizes.

nursery size is needed for different stages of a program's execution. It shows the elapsed time of each segment of execution between GCs (GC time included) using the best case size, the black line, and one of the worst ones, `-A64m`, shown in brown. What we want to show in this figure is that, although we discovered that the globally optimal nursery size was `2MB`, in some phases of the program execution, it would be beneficial to have a nursery of a different size, as is clear because of the lower elapsed runtime of the `64MB` version at the spikes in the figure.

Lastly, Figure 4 shows a zoomed view of the automatic nursery sizing algorithm in one of the phases with a big amount of live data. We plotted the amount of copied data on top of it, in order to find out the relation between the amount of live data and the chosen nursery size.

As we described in Section 2.2, the dynamic nursery resizing algorithm in GHC uses a formula that, given a fixed heap size, estimates the amount of space that will be needed in the next GC and then uses the rest of the heap size for the nursery. This means that the nursery size is inversely proportional to the amount of data that survives each GC. We can see that in the first change

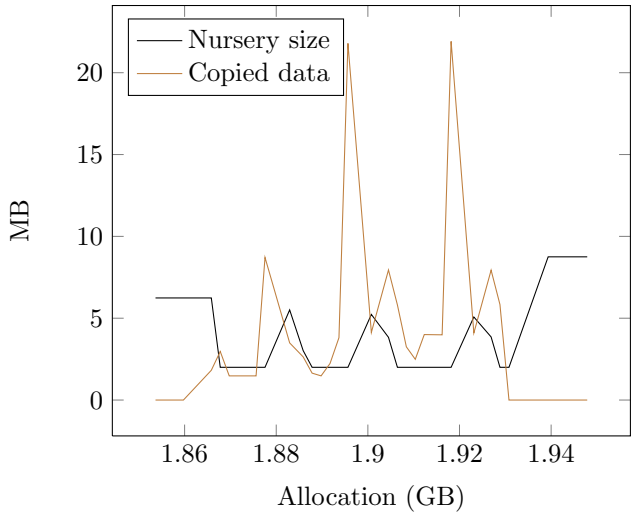

Figure 4. Automatic resizing of the nursery, run with `-H`.

that happens after `1.86GB`, where an increase in the amount of copied data made the nursery size smaller. The opposite behaviour happens at the end of the plot. What occurs in between, is that three spikes in copied memory force major collections, updating the heap suggested size. This inmediately increases the nursery size, which then keeps decreasing because of a significant rate of survivors in the nursery, which accumulate in the total live data.

Summarising this, the main problem with the current behaviour of the nursery resizing policy is that it tries to maximise the nursery size instead of making it proportional to the rate of survivors in each GC. Additionally, it uses a target heap size which is only updated at each major collection, rendering its results useless after some iterations of minor collections.

As a first step to fix this problem, we developed a tool which analyses the liveness information of a previous run of the program to find out different stages in the program in which to use different nursery sizes. When used with the example with presented here, it detects the stages of the program with a high percentage of survivors and writes a profile with a nursery size suggestion which avoids repeated collections on live data by accumulating the allocated memory in that phases so that a GC is triggered once the stage is over and there is little amount of live data. Using this technique, we were able to get an speedup of 22.74% against the best execution presented above.

3. Algorithm for Dynamically Changing Allocation Area Size

4. Evaluation

5. Related Work

Most generational garbage collectors use two generations: nursery and mature space. Appel garbage collector [**?**] dynamically tunes the size of the nursery area to be the size of the free space in heap, and divides the nursery into the allocation area and the reserved area. Only nursery GCs are performed until the nursery becomes smaller than some predeterminated value, in which case the GC is performed on the whole heap. Velasco et al. [**?**] aim to improve this strategy by giving more space from the nursery to the allocation area than to the reserved area. After each collection, they use one of the two strategies to decide on the proportion of nursery space to give to the allocation area – *average*, where average ratio between

the data copied and the allocation area size over last collections is given to the allocation area, and the *worst*, where the worst ratio is given to the allocation area.

Guan et al. [**?**] consider three different policies for dynamic nursery resizing in a HotSpot generational garbage collector: GC Ergonomic Policy (measuring GC pause time and throughput in each collection and adjusting the nursery size accordingly), Fixed Ratio Policy (keeping the same ratio between the nursery and the mature space) and Heap Availability Policy (similar to the policy considered by Velasco). Their conclusion is that in most of the cases, Heap Availability Policy brings the best results. Anderson [**?**] describes another method of dynamic resizing (in the context of functional languages), where they measure the time it takes to do GC on a nusersy (normalised by nursery size) in each collection, and half the nursery size if this time becomes too significant. White et al. [**?**] propose a strategy for adaptive resizing of the whole heap based on control theory.

6. Conclusions and Future Work

References