Automated Heap Sizing in the Poly/ML Runtime (Position Paper)

David R. White[1], Jeremy Singer[1], Jonathan M. Aitken[2], and David Matthews[3]

[1] School of Computing Science, University of Glasgow
[2] Department of Computer Science, University of York
[3] Prolingua Ltd
{david.r.white,jeremy.singer}@glasgow.ac.uk
jaitken@cs.york.ac.uk
david.matthews@prolingua.co.uk

Abstract. Typical theorem-proving workloads on the Poly/ML runtime may execute for several hours, occupying multi-gigabyte heaps. The runtime heap size may be fixed at execution startup time, or it may be allowed to vary dynamically. To date, runtime heap size growth has been implemented using simple hard-coded heuristics. In this position paper, we argue that a mathematically rigorous approach to heap sizing, based on control theory, is more appropriate.

Keywords: Heap Size, Control Theory, Poly/ML

1 Introduction

The Poly/ML system features automatic memory management, supported by by a runtime garbage collector. It is apparent that the dynamic heap size of a garbage-collected program can have a significant impact on its execution time. If the heap size is too small, then the application cannot make much progress due to frequent GC pauses. If the heap size is too large, then it may exceed the physical memory of the host machine and incur costly page faults. Figure 1 gives a hypothetical illustration of this heap sizing problem.

In this work, we argue that the impact of heap sizing can dominate the execution time for a program, such that it renders the particular choice of garbage collection algorithm relatively insignificant Unfortunately, there is no generally applicable technique to determine, ahead-of-time, the expected impact of a particular heap size on the execution time of a given program. Factors including the the dynamic allocation behaviour of the software and the underlying memory manager in the host OS complicate the relationship between heap size and execution time. Many programs proceed through distinct phases of dynamic allocation behaviour [5, 4, 1], and thus it is important that the heap size *adapts* to accommodate shifts in application allocation characteristics.

A good heap sizing mechanism should minimise the overhead of GC, make efficient use of memory and avoid problems such as paging overhead [6]. Setting a large static heap size is an inefficient use of memory, and thus should be avoided.

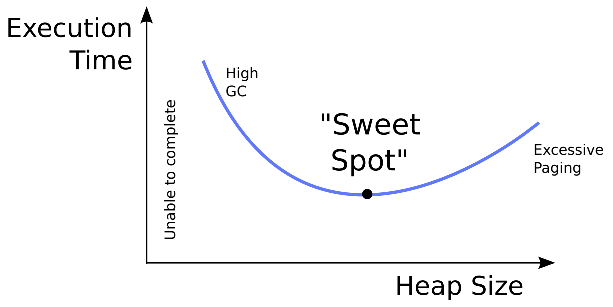

Fig. 1. Heap Size 'Sweet-Spot' for optimal application performance

This paper proposes the use of *control theory* [3] to adjust heap sizes dynamically. In contrast to existing, heuristic-based techniques for heap sizing, control theory provides a principled approach underpinned by mathematical theory. As managed runtime environments become more sophisticated and widespread, a shift from expert-designed, hand-tuned heuristics to rigorous autonomic mechanisms is increasingly appealing.

Heap sizing heuristics are generally opaque to end-users, and may cause confusion and frustration. Below is an email extract[4] from the Poly/ML mailing list, showing how one user struggled with heap sizing.

> *When I build a large session in Isabelle2011-1 (to be precise, an extension of of JinjaThreads in the Archive of Formal Proofs) with PolyML 5.4.1, this takes 1:51h. While running the session, polyml requires 12GB of memory (`VmSize` in `/proc/PID/status`). The final call to `PolyML.share CommonData` before writing the heap image consumes 17GB of memory.*
>
> *I now ran the same session with the SVN version 1352 of PolyML. Then, it takes 2:35h, 16GB for the session and 21GB for sharing common data. What is happening there? Have other people experienced a similar surge in memory usage and runtime?*

This problem was caused by a change in the GC model, invisible to end-users. The heap size was increased to such an extent that it caused the process to incur a large number of page faults.

Contributions of Paper

This paper:

1. provides a characterization of the existing Poly/ML memory management dynamics.

[4] Archived at `http://comments.gmane.org/gmane.comp.lang.ml.polyml.general/510`

2. explains how the adaptive heap sizing model may be modified to incorporate a control system, allowing the user to set an explicit target for GC overhead in the system.

2 Existing Poly/ML Memory Management Behaviour

This section gives a short review of the existing memory management infrastructure in the Poly/ML runtime. This is based on revision 1460 from the sourceforge subversion repository for Poly/ML[5] dated 16th March 2012.

2.1 Garbage Collection

The Poly/ML runtime uses a stop-the-world, generational, parallel GC scheme. There is an *allocation region* (effectively the nursery space) where all memory cells are initially allocated. This heap region is evacuated during *quick* GCs (minor collections). Surviving cells are copied to the mutable and immutable regions (the type-specific mature spaces). The major collector is started when either the mutable or the immutable area is full, usually triggered by a minor GC. The major collector uses a mark/sweep/compact GC scheme.

Parallelism is optional in the runtime. If enabled, a task farm of threads executes each stage of the GC. There is a rendez-vous for all threads at the end of each stage.

2.2 Heap Layout

Figure 2 shows how the different regions of the runtime heap are laid out, and how objects move between regions. Initially, all objects are created in the *allocation space*, which functions as a nursery region for young objects. If objects survive partial GCs in the nursery, then they are tenured to the mature region, which is divided into *mutable* and *immutable* spaces.

In functional languages like ML, most objects are *immutable*, i.e. they cannot be changed once they have been created and initialized. Example immutables are strings, lists, functions and trees. The only mutable objects are refs; these are cells holding addresses, which may be updated.

The distinction between immutable and mutable objects is important for runtime performance: Each minor GC only needs to look at the mutable data (and the other roots) to search for pointers to newly created cells. The immutable space, which contains the overwhelming majority of cells, is not examined at all during a minor collection. Thus the cost of a minor GC is very much less than for a language where the distinction between mutable and immutable data cannot be made.

The initial runtime heap size can be explicitly set by the user on the command line, as well as the relative sizes of the different regions. However if heap size

[5] `https://polyml.svn.sourceforge.net/svnroot/polyml/`

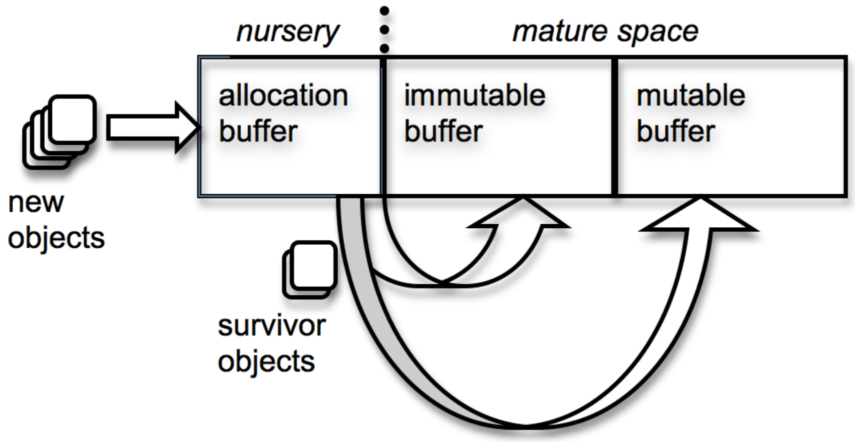

Fig. 2. Schematic Layout of the Poly/ML Runtime Heap

parameters are not supplied, the runtime defaults to allocating half the physical RAM as the initial heap size. This heap is split into two equal parts: one for the allocation space (nursery), and the other for the mature space.

2.3 Runtime Resizing of Heap

At the end of every major GC, the `adjustHeapSize()` method is called. This method varies the heap size dynamically so there is a specific amount of free space available. That is to say, suppose the mature space contains l MB of live data immediately after a GC, then `adjustHeapSize()` varies the mature space size to $K + l$ MB, where K is a precomputed constant amount. K is the value of the `majorGCFree` static variable in `gc.cpp`, which is set to the size of the mature space when the Poly/ML runtime commences execution.

2.4 Typical Behaviour

In this section, we discuss typical heap size dynamics for some standard Poly/ML workloads. Figure 3 shows an example execution of an Isabelle theorem proving session on the Poly/ML runtime. Note the distinct phases in the live data size, as the graph has several live data ramp-up steps, interspersed by relative plateaus. These phases will correspond to particular stages in the theorem proving execution. The execution time is not insignificant. This session runs for over five minutes. The user email quoted in Section 1 mentions execution times of several hours.

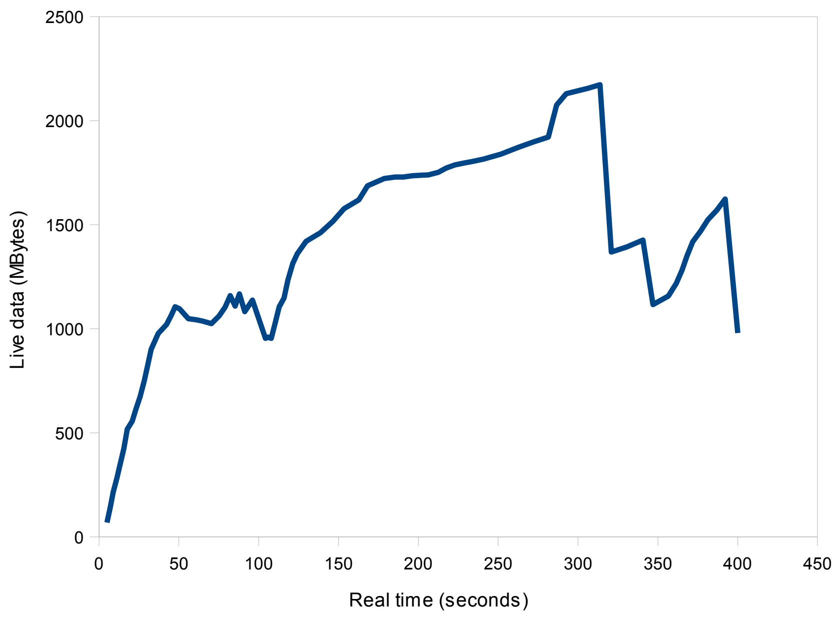

Fig. 3. Graph showing total live data size measured at each major GC against real-time for a single execution of a typical Poly/ML workload

3 Proposed Extension to Poly/ML Runtime

3.1 Control System for Heap Sizing

We propose to use a *control system* to model the problem of heap sizing. Control theory is a well-established branch of engineering that can be used to vary an input control signal to a system in order to extract a desired output. Commonly, feedback is employed in what is known as a *closed-loop* controller. Figure 4 illustrates an abstract control system. The deviation from the desired behaviour (the error of the controlled variable when compared to a reference signal) is used as a feedback signal to adjust the control signal. The controller is chosen to effectively modify the characteristics of the aggregate controller-system pair. We aim to use control theory to create a mathematically elegant and efficient approach to adaptive heap sizing.

Changes in the system, such as a software phase change or increased demand from other applications, can be considered as a change in the optimal heap size, and hence we are attempting to control a dynamic system as may be encountered in the field of control theory. The controller must respond effectively to changes while ignoring spurious noise. The situation is illustrated in Figure 5, where we use the total memory allocated by a program as a proxy for time.

A 'good' controller seeks to minimize both:

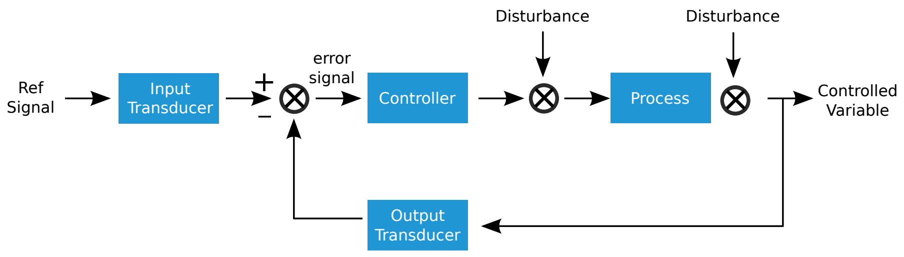

Fig. 4. A closed-loop control system

1. Steady-state error: a system is said to have zero steady state error if it eventually settles to the commanded value. In the case of a mechanical lift system we would wish to have it stop exactly at each floor.
2. Transient response time: i.e. how quickly a system responds to an input. In the case of a lift, if the response time is slow the passengers may grow bored waiting to reach their destination. However, they will feel uncomfortable acceleration if the response time is too fast.

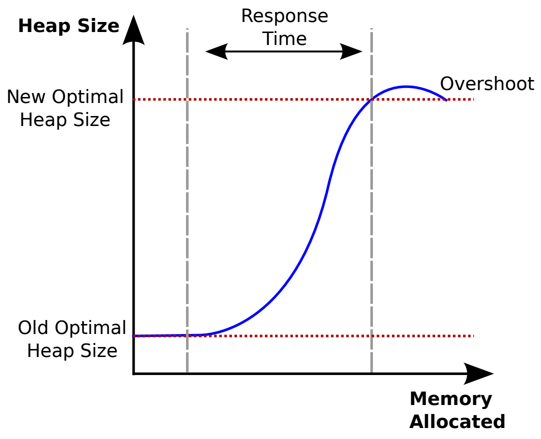

Fig. 5. Viewing heap sizing as a control problem

3.2 Formulating the Problem

To consider heap sizing as a control problem, we must identify a *control variable* that we will use to manipulate the system, and a *measurement variable* that we will use as feedback to modify the control variable. We select the *heap resize ratio* for the control variable; we will focus on the short-term GC overhead as our measurement variable. This decision is partly based our experience with heap resizing functions in Java virtual machines, including Jikes RVM and OpenJDK.

Controller design usually considers the evolution of the system under control in the time domain (before moving into the Laplace domain). However, we adopt memory allocated as a proxy for time, due to the variable nature of execution time across different platforms.

3.3 Deploying a Heap-Size Controller

We elect to treat the system as a *black box*, and apply a popular and robust controller known as a PID (proportional-integral-derivative) controller [3].

We attempt to maintain a given GC overhead g^* set by the user. Thus we do not model the system, e.g. using differential equations. Rather we tune the controller empirically for the system. As with the original Poly/ML memory manager, decisions about resizing the heap only occurs immediately after a garbage collection, in the `adjustHeapSize()` method.

PID Controller PID controllers implement a control technique that builds upon compensator design. It used both proportional plus integral and proportional plus derivative control to achieve improvements in steady-state error and transient response time. A PID controller uses the following time-domain equation:

$$u(t) = K_c \left(\epsilon(t) + \frac{1}{T_i} \int \epsilon(t)\ dt + T_d \frac{d\epsilon(t)}{dt} \right) + b \tag{1}$$

Our control signal, $u(t)$, is the heap resize ratio at time t, i.e.

$$\text{new heap size} = u(t) \times \text{old heap size} \tag{2}$$

Recall that in our equations time actually denotes memory allocated by application threads.

The control signal $u(t)$ is given relative to a setpoint of b, which in our case is a unitary resize ratio. The error measure $\epsilon(t) = g^* - g(t)$ is the deviation at time t from the desired garbage collection overhead g^*. This error is calculated afresh at the end of each major GC.

The constants K_c, T_i and T_d control a proportional, integral and derivative response to the error signal. The balancing of these constants defines the controller behaviour. K_c is referred to as the overall 'gain' of the controller.

Controller Implementation In order to implement a PID controller as outlined above, we need to make the following modifications to the Poly/ML codebase:

1. Extend `MemMgr` class to keep a running count of total bytes allocated on behalf of application threads, to serve as a proxy for time.

2. Extend `gc.cpp` to measure the short-term GC overhead and pass this data to the PID controller, along with the allocation statistics from memmgr.cpp.
3. Adapt `adjustHeapSize()` to use the PID controller when considering a heap resize. This is implemented to respect the maximum and minimum heap sizes, if specified in configuration files or command-line parameters.

Controller Tuning A controller must be tuned for the system in which it is deployed. The tuning process tailors the controller to the characteristics of the underlying system—in our case by setting the parameters in the PID equation. We use Ziegler/Nichols tuning [7].

This approach runs the system in a closed feedback configuration with T_i and T_d both set to 0. The gain K_p is increased from 0 until it reaches a value K_u, at which point the system begins to oscillate with period T_u. The parameters for the PID controller in Equation 1 can then be calculated directly as shown below:

$$\begin{aligned} K_c &= 0.6K_u \\ T_i &= 0.5T_u \\ T_d &= 0.125T_u \end{aligned} \tag{3}$$

This method is not favoured for mechanical systems where there is a risk of damaging the system by applying excess strain in reaching the point of oscillation. However it is ideal for a virtual, software system.

3.4 Evaluation Plan

Once we have completed the development of our control theoretic heap sizing extension to Poly/ML, we will have to evaluate its effectiveness.

The design of a scientific comparison between the old heuristic heap sizing approach and our new controller based approach is non-trivial. Whereas the previous system had no explicit goal or optimization target, the new system attempts to maintain a consistent GC overhead. We could measure how effectively our system meets its target, by plotting actual GC overhead against time for some standard workloads.

A measure of relative efficiency might be the area under the (heap size *vs* time) curve—this measure is often known as *memory space rental.* Ideally the space rental should be reduced for our new system in relation to the old system, without a significant effect on overall runtime.

We might also examine the system's response to sudden spikes in application live size. Whereas we expect the old system to adapt slowly, we should be able to adapt more quickly. It may be necessary to produce some micro-benchmarks tests to verify this assertion.

4 Future Work

At the moment, our proposed control theoretic heap resizing technique is not aware of paging behaviour. We could either impose a hard limit on the maximum heap size, but this presumes that there are no other memory intensive processes concurrently executing with Poly/ML. We might assume that paging will affect GC more than application execution, due to the larger working set size of a full-heap GC. If this assumption is true, it means that GC overhead would increase in the presence of paging, which would make our controller increase the heap size, which would exacerbate the problem. We might need to consider either (i) using a multi-level controller, to consider reducing page faults before controlling GC overhead, or (ii) having an alternative goal for our single controller.

Matthews et al. [2] discuss acceptable levels of GC overhead in Poly/ML workloads. They presume that a 5% overhead is average, although they show that the overhead can rise to 30% in more intensive workloads.

We note that the Haskell runtime system (RTS) also uses simple heuristics to control its heap growth[6]. We expect that if our control theoretic model is successful in Poly/ML, we might also advocate its adoption for Haskell RTS.

References

1. Duesterwald, E., Cascaval, C., Dwarkadas, S.: Characterizing and predicting program behavior and its variability. In: Proceedings of Parallel Architectures and Compilation Techniques (2003), `http://dx.doi.org/10.1109/PACT.2003.1238018`
2. Matthews, D.C., Wenzel, M.: Efficient parallel programming in Poly/ML and Isabelle/ML. In: Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming. pp. 53–62 (2010), `http://doi.acm.org/10.1145/1708046.1708058`
3. Nise, N.S.: Control Systems Engineering. John Wiley & Sons, Inc., 3rd edn. (2000)
4. Reiss, S.P.: Dynamic detection and visualization of software phases. In: Proceedings of the third international workshop on Dynamic analysis (2005), `http://dx.doi.org/10.1145/1083246.1083254`
5. Soman, S., Krintz, C., Bacon, D.F.: Dynamic selection of application-specific garbage collectors. In: Proceedings of the 4th international symposium on Memory management (2004), `http://dx.doi.org/10.1145/1029873.1029880`
6. Zhang, C., Kelsey, K., Shen, X., Ding, C., Hertz, M., Ogihara, M.: Program-level adaptive memory management. In: Proceedings of the 5th international symposium on Memory management (2006), `http://dx.doi.org/10.1145/1133956.1133979`
7. Ziegler, J.G., Nichols, N.B.: Optimum settings for automatic controllers. Transactions of the American Society of Mechanical Engineers 64, 759–768 (1942)

[6] e.g. `http://hackage.haskell.org/trac/ghc/ticket/3061`