Control Theory for Principled Heap Sizing

David R. White Jeremy Singer
School of Computing Science
University of Glasgow
{david.r.white,jeremy.singer}@glasgow.ac.uk

Jonathan M. Aitken
Department of Computer Science
University of York
jonathan.aitken@york.ac.uk

Richard E. Jones
School of Computing
University of Kent
r.e.jones@kent.ac.uk

Abstract

We propose a new, principled approach to adaptive heap sizing based on control theory. We review current state-of-the-art heap sizing mechanisms, as deployed in Jikes RVM and HotSpot. We then formulate heap sizing as a control problem, apply and tune a standard controller algorithm, and evaluate its performance on a set of well-known benchmarks. We find our controller adapts the heap size more responsively than existing mechanisms. This responsiveness allows tighter virtual machine memory footprints while preserving target application throughput, which is ideal for both embedded and utility computing domains. In short, we argue that formal, systematic approaches to memory management should be replacing ad-hoc heuristics as the discipline matures. Control-theoretic heap sizing is one such systematic approach.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Memory management (garbage collection); D.4.2 [*Operating Systems*]: Storage Management—Allocation / deallocation strategies

Keywords Heap Size; Control Theory; Virtual Machines; Jikes RVM; HotSpot; Ergonomics

1. Introduction

The dynamic heap size of a garbage-collected program can have a significant impact on its execution time. We believe that optimization of per-program heap size will become more important with the increasing use of garbage collection (GC) on embedded systems, as well as the growth of utility computing via the cloud. The dominant customer billing model for the latter is likely to be based on CPU cycles and memory space rental [2, 6, 13, 16].

Unfortunately there is no general technique to determine, ahead-of-time, the expected impact of a particular heap size on the execution time of a given program. Factors such as the dynamic allocation behavior of the software, the GC policy of the managed runtime, 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 behavior [15, 26, 28], thus it is important that the heap size *adapts* to accommodate shifts in application allocation characteristics.

A good heap sizing mechanism should minimize the overhead of GC, make efficient use of memory and avoid problems such as paging [36]. Setting a large static heap size is an inefficient use of memory; this should be avoided.

This paper proposes the use of *control theory* [24] to adjust heap sizes dynamically. In contrast to existing, heuristic-based techniques for heap sizing, control theory provides a principled mathematical approach. As virtual machines (VMs) become more sophisticated and widespread, a progression from expert-designed, hand-tuned heuristics to rigorous autonomic mechanisms is increasingly appealing.

We implement a particular controller that monitors short-term GC overhead, and seeks to maintain this at a pre-defined level by adjusting the heap size accordingly. Using this controller, we are able to maintain target levels of application throughput. This is ideal for a high-level quality-of-service agreement, such as might be required in a utility computing context [25].

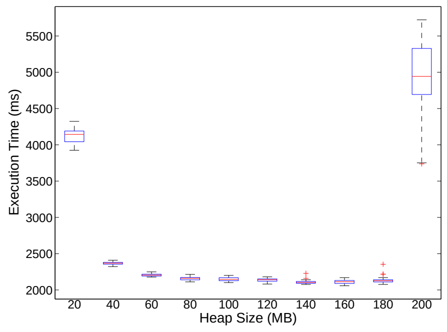
With the exception of HotSpot's *ergonomics* functionality [30] most VMs do not provide users with the facility to specify where an application's execution should lie on the time-space tradeoff curve. Alonso and Appel [3] describe the concept of flexible working sets when employing garbage collection. There is a minimum heap size below which an application cannot execute, and increasing the heap size above this value reduces GC overhead, hence reducing overall application time. Generally the garbage collector is tuned by an expert, to give some average, acceptable level of performance. Unless a fixed heap size is set, the user has no fine grained control over tradeoff between memory and execution time. Whilst HotSpot does provide this functionality, its implementation shares some of the weaknesses of other heap sizing mechanisms, as we discuss later.

This paper makes two main contributions:

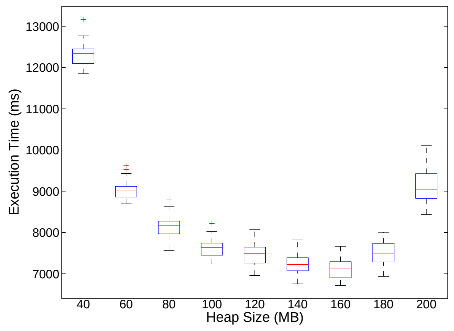
1. It motivates and describes the use of a proportional-integral-derivative (PID) controller for runtime heap sizing, including high-level theory and low-level implementation details in Jikes RVM.
2. It provides an empirical characterization of PID controller heap sizing on a selection of DaCapo benchmark workloads, and compares full-heap GC behavior with Jikes RVM and HotSpot ergonomics heap sizing mechanisms.

2. Heap Size Sweet-Spots

If we consider the large-scale behavior of software, then we may assess the impact of a (fixed) heap size on execution time. Figure 1 illustrates this relationship on a Linux system limited to 300MB RAM by a kernel boot parameter, running Jikes RVM with the antlr and lusearch benchmarks from the DaCapo suite (v2006-10-MR2 [9]) using a full-heap mark/sweep collector, default inputs and 30 repetitions. There is a balance to be struck between a small heap, where GC overhead is high, and a large heap, where paging may occur if limited memory is available. Each graph shows a



(a) DaCapo 2006 antlr



(b) DaCapo 2006 lusearch

Figure 1: Heap size vs execution time for two benchmarks

		Heap Occupancy					
		0.00	**0.10**	**0.30**	**0.60**	**0.80**	**1.00**
GC Overhead	**0.00**	0.90	0.90	0.95	1.00	1.00	1.00
	0.01	0.90	0.90	0.95	1.00	1.00	1.00
	0.02	0.95	0.95	1.00	1.00	1.00	1.00
	0.07	1.00	1.00	1.10	1.15	1.20	1.20
	0.15	1.00	1.00	1.20	1.25	1.35	1.30
	0.40	1.00	1.00	1.25	1.30	1.50	1.50
	1.00	1.00	1.00	1.25	1.30	1.50	1.50

Table 1: Jikes RVM heap sizing look-up table

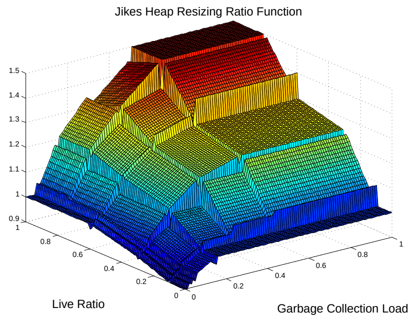


Figure 2: Visualizing the Jikes RVM heap resizing function

'sweet-spot' at a specific heap size, where overall execution time is minimized. This behavior is also reflected in realistically sized, production configurations [10].

This curve will be affected by the VM's environment, particularly the amount of available memory. Issues with paging frequently occur when many applications are executing concurrently on a machine, e.g. a compute node in a cloud data center, or where memory is limited in embedded systems, e.g. an Android device.

While these graphs give us an idea of 'optimal' heap size if we were forced to select a constant value ahead-of-time, in practice the problem is dynamic and this optimum changes at different points during execution. Therefore it is essential that any heap sizing mechanism is *adaptive* and able to respond efficiently to changes in application behavior.

3. Heap Sizing in Deployed VMs

In this section, we survey how two popular production VMs adapt their heap sizes. This is generally a combination of both user-specified thresholds (such as the `-Xmx` maximum heap size parameter to the Java VM) and hand-crafted heuristics encoded within the memory manager.

As a rule, such systems tend to be *improvized*, in that they are not based on any underlying theory. Further, they are usually *conservative* in that they prefer to increase heap size rather than decrease it, and often at a slower rate than may be desired.

3.1 Jikes RVM

Throughout this paper, we use Jikes Research Virtual Machine (RVM) [4, 5], in combination with the Memory Management Toolkit (MMTk) [8] as our experimentation platform. First, we consider how it currently implements heap resizing. After each GC, the `HeapGrowthManager` class is queried to determine a suitable *resize ratio* for the heap based on two variables: these are 'short-term GC overhead', g, and 'current live ratio', l. The resize ratio $r(g, l)$ is a function of these two variables. The function inputs are calculated immediately after each GC has completed, as follows:

$$g = \frac{\text{Time taken for the most recent GC}}{\text{Time since the last GC}} \quad (1)$$

$$l = \frac{\text{Amount of live data on the heap}}{\text{Current heap size}} \quad (2)$$

The resize ratio r is calculated by using these two values as indices into a look-up table. One version of the RVM look-up table is illustrated in Table 1. The two variables are matched to the values in bold, which represent interval boundaries, and the resize ratio is generated based on this look-up table with some interpolation. By calling the RVM source from Matlab, we can visualize this function, as shown in Figure 2. The visualization shows that this function is essentially a discrete valued function using linear interpolation between values. We can also see the discontinuities in the function, due to a small interpolation bug in the source code.

The values in the look-up table are hardcoded in the VM and, according to a private communication [18], were the result of trial-and-error experimentation several years ago. We observe that:

- The system is not goal-oriented; there is no target state that it aims to achieve, for example a particular heap size or value of g.
- The mechanism is *stateless*: i.e. it does not take past behavior into account.
- Trial and error is not an objective approach to determining good coefficients.

- Hand-crafted heuristics are susceptible to programming errors, unanticipated situations and pathological cases (for example, the programming error above is only exposed when the input variables are specific values).
- There is no evidence to believe that the heuristic is still valid. The GC implementation has changed considerably since the table was established. Why should we believe it still works?

Perhaps in view of these limitations, the resize function is clearly conservative. When profiling the behavior of DaCapo benchmarks, we have found this that the heap is rarely shrunk significantly, and heap size growth lags behind application behavior.

3.2 HotSpot

Since version 1.5, the Sun (Oracle) HotSpot JVM features an adaptive heap sizing policy known as *GC ergonomics* [30]. The user can specify ahead-of-time values for three targets: (1) maximum GC pause time goal, (2) application throughput goal (i.e. proportion of overall execution time spent in application code), and (3) minimum heap size. According to Sun's published documentation [30], the ergonomics system applies the following heuristics (in this order). (a) If the GC pause time is greater than the pause time goal then *decrease heap size* to attain the goal. (b) If the pause time goal is being met, then consider the application's throughput goal. If the application's throughput goal is not being met, then *increase heap size* to attain the goal. (c) If both the pause time goal and the throughput goal are being met, then *decrease heap size* to reduce memory footprint.

The heap sizing policy is implemented in the `AdaptiveSizePolicy` class and its subclasses. We have examined code from the OpenJDK v6 open-source release: it consists precisely of the above series of hard-coded, case-based rules. The heap resize ratios are less flexible than for Jikes RVM. In steady state, the heap growth ratio is fixed at 1.2 for increases and at 0.95 for decreases. In the early stages of execution, there is a supplementary value added to the growth ratio for increases, so the first time the heap grows the ratio will be 2.0. However this supplementary value decays towards 0 as further GCs occur. In summary, HotSpot heap sizing is more goal-oriented than Jikes RVM, but still lacks a rigorous mathematical model. Vengerov [33] notes that the ergonomics policy is based on 'some heuristic rules that do not guarantee that the GC throughput [or pause time] will actually be maximized [or minimized] as a result.'

4. Heap Sizing as a Control Problem

We propose that heap sizing should be treated as a *control problem*. 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 obtain a desired output signal. Commonly, feedback is employed in what is known as a *closed-loop* controller. Figure 3 illustrates an abstract control system. The deviation from the desired behavior (the error of the output when compared to a reference signal) is used as a feedback signal to adjust the input control signal. We rely on existing work in control theory to produce a simple, elegant, efficient and well-founded solution to the problem of adaptive heap sizing.

A change 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. In this sense, we are attempting to control a dynamic system such as those that may be encountered in the field of control theory. The controller must respond effectively to changes while ignoring spurious noise. Figure 4 illustrates this situation.

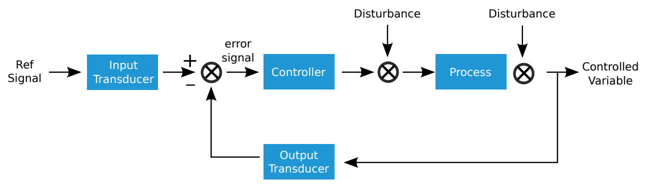


Figure 3: A closed-loop control system

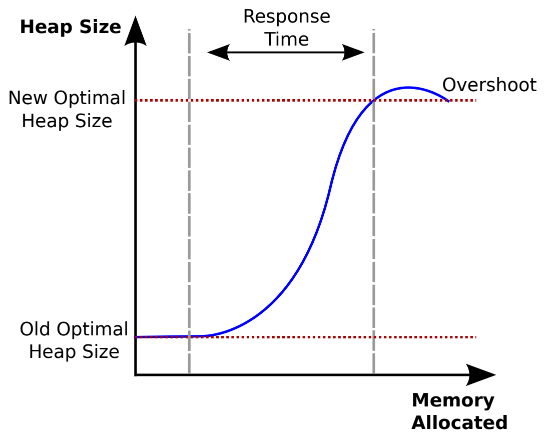


Figure 4: Heap sizing as a control problem

Traditional control theory is concerned with designing effective controllers with regard to the following characteristics:

- Steady-state error: a system is said to have zero steady state error if it eventually settles to the target value. In the case of a mechanical lift system we would wish to have it stop exactly at each floor. In the case of our system, we may expect to settle at the desired value of g (for instance).
- Transient response characteristics: 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. In this terminology, current heap sizing methods can be considered conservative, in that they have a long response time.

4.1 Formulating the Problem

To consider heap sizing as a control problem, we must decide upon the input control variable we will use to manipulate the system, and the output measurement variable we will use as feedback to modify the control variable. Our chosen control variable is the heap resize ratio and in this work we will focus on g, the short-term GC overhead as our measurement variable (alternatives might include mark/cons ratio). This decision is partly based on the use of g in the existing Jikes RVM heap sizing function and in the ergonomics system provided by HotSpot. Any deployed controller is likely to consider at least one other measured variable, but for the moment we focus on g only. Note that we intend to demonstrate the effectiveness of control theory in regulating the system, rather than recommending one variable over another.

We make one modification to the GC overhead variable, i.e. we calculate a *median* average over a sliding window of size five. The sliding window is initialized with the GC overhead target value in all five slots, and then updated with the most recent GC overhead measurement in a FIFO style after each GC. This filtered average was implemented after an empirical evaluation found excessive noise in the GC overhead signal for controller tuning (see later, Section 5.3). The sliding window average dampens this signal somewhat. We denote this average as $\hat{g}$. HotSpot employs a similar smoothing mechanism via its `AdaptiveWeightedAverage` class. It computes average GC overhead x in terms of most recently measured GC overhead g via the recurrence relation $x_{n+1} = \alpha g + (1-\alpha)x_n$. The default α value is 0.5. Such exponentially decaying sample calculations are common for lightweight overhead management in software systems, e.g. QVM [7].

Controller design usually considers the evolution of the controlled system in the time domain (before moving to the Laplace domain). However, we adopt *memory allocated* as a proxy for time, due to the variable nature of execution time across different processors. This is common practice in GC analysis literature ([22] p145).

5. Designing a Heap Size Controller

There is limited existing work on the mathematical characterization of heap sizes for Java applications [32], and it is not clear whether such formulations are generally applicable. It appears that the equations describing heap size are tied to a particular system configuration, for a fixed benchmark/input combination, with no external dynamic variation. Instead we chose to treat the system as a *black box*, and apply a popular and robust controller known as a PID (proportional-integral-derivative) controller [14, 24].

We attempt to achieve and maintain a *target* GC overhead g^* as set by the user. Thus we do not model the system using differential equations (for example), but rather we rely on empirical work to tune the controller to the system. Similarly, we do not consider the application of more sophisticated control methods such as a state-space controller or the formulation of the problem as one of optimal control. These may prove useful avenues for further research.

In our implementation, a decision about resizing the heap only occurs immediately after a GC. This is consistent with the standard Jikes RVM behavior. This ensures that we isolate the effect of the controller as the single change to the system.

5.1 PID Controller

5.1.1 Why a PID Controller is Appropriate

The PID controller is an ideal selection for this problem; it offers a three-term controller with zero-steady state error provided by extra integral action and is proportionate to the time history and predicted progression of the system response. Additionally it requires no model of the system, in effect performing black-box control, albeit one that is tuned rather than designed for the system.

5.1.2 PID Controller Theory

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:

$$u(t) = K_c \left(\epsilon(t) + \frac{1}{T_i} \int_0^t \epsilon(t)\, dt + T_d \frac{d\epsilon(t)}{dt} \right) + b \qquad (3)$$

The PID controller operates on an error signal $\epsilon(t)$, developed from the difference in the system input and output. This error value is minimized by the three terms in the PID equation, which adjust how quickly the controller reacts to a change in input, and therefore a change in the error signal. The proportional term K_c provides a term which is a linear multiplier of the error value at the current time-step. This is a simple traditional gain control block. The value of this gain adjusts the responsiveness of the controller and there is a distinct trade-off that must be made: too small a value and the system will respond slowly to an input, but too large a value and the system will become unstable due to a phase inversion producing positive rather than the desired negative feedback and thus signal amplification rather than reduction.

The integral component, T_i, controls responsiveness to the time history of the error signal. If the error signal is growing, it provides an extra gain relative to the summation of the duration and magnitude of this error. This helps provide an extra boost in reducing the error to zero. As the integrative term only uses time history, a high value may well produce overshoot in the desired value resulting in the system hunting for and oscillating towards a zero error. Due to the introduction of an integral term this reduces any steady-state error to zero. Any controller lacking an extra integral will contain a steady-state error, a constant difference between desired and achieved value when the system is allowed to settle.

The differential component, T_d, controls responsiveness to a predication made about the error signal. This helps add stability and make the system more responsive to changes that diverge from a zero steady state error. The derivative within the calculation makes it very susceptible to measurement noise on the output. The differential could produce a much larger signal that the actual system response.

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

$$\text{new heap size} = u(t) \times \text{old heap size} \qquad (4)$$

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 $\epsilon(t) = g^* - \hat{g}(t)$ is the deviation at time t from the desired garbage collection overhead g^*. This error is calculated at the end of each garbage collection, since g and hence $\hat{g}$ only change after a garbage collection event.

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 behavior. K_c is referred to as the overall 'gain' of the controller.

5.2 Controller Implementation

We instrumented the `MemoryManager` and `HeapGrowthManager` classes from Jikes RVM and MMTk respectively to analyze the behavior of the existing system and to allow us to perform the design and tuning of our controller. In terms of the PID itself, we made the following changes:

1. The `MemoryManager` class was modified to keep a running count of total bytes allocated, to serve as a proxy for time.
2. The `HeapGrowthManager` class was modified to measure, record and transmit values for the short-term GC load to the PID controller, along with the count from `MemoryManager`. This class also maintains a sliding window to calculate average value $\hat{g}$.
3. The `HeapGrowthManager` class was also modified to use the PID controller when considering a heap resize. Any resizing respects the maximum and minimum heap sizes as specified in the boot image and commandline parameters of the VM. When heap size hits the lower or upper value, the PID controller integral term is reset to zero to prevent *integral windup*, which causes errors when signals are clipped at boundary values.
4. A `PID` class was created to maintain the controller operation.

A patch adding this controller to Jikes RVM is available online [1] along with our source code, experimental scripts, analysis scripts and output data.

5.3 Controller Tuning

The tuning process aims to tailor the controller to the characteristics of the underlying system. We use the empirical tuning method by Ziegler and Nichols [38] to determine the constants used in the PID controller. This involves controlling the system with no integral or derivative component. The gain K_c is adjusted until it reaches the ultimate gain K_u, when the output signal begins to oscillate with period T_u. The parameters for the PID controller in Equation 3 can then be calculated as shown below:

$$\begin{aligned} K_c &= 0.6K_u \\ T_i &= 0.5T_u \\ T_d &= 0.125T_u \end{aligned} \tag{5}$$

This method is not favored for mechanical systems where there is a risk of damaging the system by applying excess strain in reaching the point of oscillation. This is especially true of high frequency oscillations, which could damage mechanical components. Also, the onset of the oscillations may prove difficult to identify if they have a long time period. Additionally sharp-fronted input signals may cause excessive strain by exceeding rates of demands on the system, e.g. consider the situation when turning a car into a corner, mechanical limitations will mean that too high a rate of turning force from the driver will result in loss of control.

However this is an ideal approach for a software system, where there is less concern about mechanical strain. Though this method provides no guarantees about whether control of the system will be optimal, rather it provides a good rule of thumb tuning process for a system to be controllable without knowledge of the underlying system model. The trade-off with developing the control this way means that there is no guaranteed response, as we have not developed the controller with any particular desired dynamics in mind. Additionally the non-linearities in the underlying system will change the dynamics of the controller response across the operational range meaning the controller will behave with different responses in different regions of operation.

6. Evaluation

6.1 Setup

In all these experiments, we execute the `FastAdaptiveMarkSweep` configuration of Jikes RVM hg tip of 25th March 2013 with the GNU Classpath library. We used the simple mark-sweep GC in order to best expose the behavior of the PID controller (and other) heap expansion managers. In particular, at this stage we wanted to avoid any complexities of separately controlling the size of more than one space (as would be necessary for a generational GC). As above, the test machine is lightly loaded (although the load should not affect our results provided that paging of the heap does not occur), Mac OS X 10.8.2, 2GHz quad-core Intel Core i7, 4GB 1333MHz DDR3. In the interest of repeatability, we provide all our code, scripts and data online [1].

We first run experiments using individual benchmarks from the DaCapo suites with `large` inputs, and then create a 'phased' benchmark for further evaluation in Section 6.5. We use 3 benchmarks from the DaCapo 9.12 suite and 8 benchmarks from the DaCapo 2006-10-MR2 suite; thus we used 11 benchmarks from the 25 in the suites. Of the 14 excluded, 10 were not used because the Jikes RVM or the supporting Classpath library were incapable of executing the benchmarks. A further 3 were not memory intensive enough to provide sufficient data, and one was too slow and memory intensive to use efficiently. Full details are available online [1].

DaCapo v.	**Benchmark**	**Target** g	K_c	K_c/T_i	K_cT_d
2009	pmd	0.09	6.6	0.01	1300
	sunflow	0.03	9.0	0.03	750
	xalan	0.04	8.4	0.02	1100
2006-10-MR2	bloat	0.11	6.0	0.02	620
	eclipse	0.14	5.4	0.01	810
	fop	0.06	7.8	0.01	1300
	jython	0.21	4.8	0.00	1200
	luindex	0.09	7.2	0.06	230
	lusearch	0.05	5.4	0.01	980
	pmd	0.15	5.4	0.01	950
	xalan	0.04	8.4	0.01	1820

Table 2: Target GC Overhead Values and Tuned PID Parameters for each Benchmark

6.2 Experiment A: Establishing Realistic Overhead Targets

We ran each benchmark on the standard RVM for an unlimited number of iterations, until 100 garbage collections had been completed, with the heap size limited to the range $[50, 250\text{MB}]$. The average GC load in these runs was calculated, and subsequently used to provide a realistic target value for the PID in controller in Experiment C. Table 2 shows the resulting target values. Each value is the median of our 'sliding window' GC overhead $\hat{g}$, hence each value in the table is actually a median of medians.

6.3 Experiment B: Tuning the PID Controller

Next, we enabled the PID controller and ran the same benchmarks in order to follow the Ziegler-Nichols method of PID tuning as described in Section 5.3. The heap size was limited to $[50, 500\text{MB}]$, and we increased the gain K_c until the system began oscillating around the goal values derived in Experiment A. Figure 5 gives an example of an oscillating system; the quality of the oscillations achieved varied between benchmarks, and the noise in the signal made tuning a subjective and imperfect process.

We took three measurements of the period for each benchmark (example shown in Figure 5) and took the median period as our final measurement. This allowed us to calculate coefficients for the PID equation on a per-benchmark basis. Note that if we were to deploy the PID controller generally, we would choose an average or other summary statistic of these values, but here we were interested in (i) a limit study of the optimal application of the technique, and (ii) whether tuning varies with the application.

6.4 Experiment C: Evaluation

We then enabled the PID controller with the coefficients derived from Experiment B; we set the goal value g^* of the controller to be the target values from Experiment A. The results for the eleven benchmarks are given in Figure 6.

There is a pair of graphs for each benchmark: the top graph of a pair shows how the garbage collection overhead varies with time, as the PID controller attempts to achieve the designated target. The bottom graph of a pair shows how the heap size is changed to achieve this goal. Each point represents a single garbage collection. Note that the graphs have different scales.

The PID controller adjusts the size of the heap in response to any deviations from the target GC overhead. Over time, we would expect adjustments to decrease provided that the software does not exhibit large variations in memory consumption. Thus, a smoother graph on the left should be reflected in a converging heap size on the right; this is what we see.

For the 2009 and 2006 xalan benchmarks the PID controller rapidly reduces the error and converges the heap size to a reason-

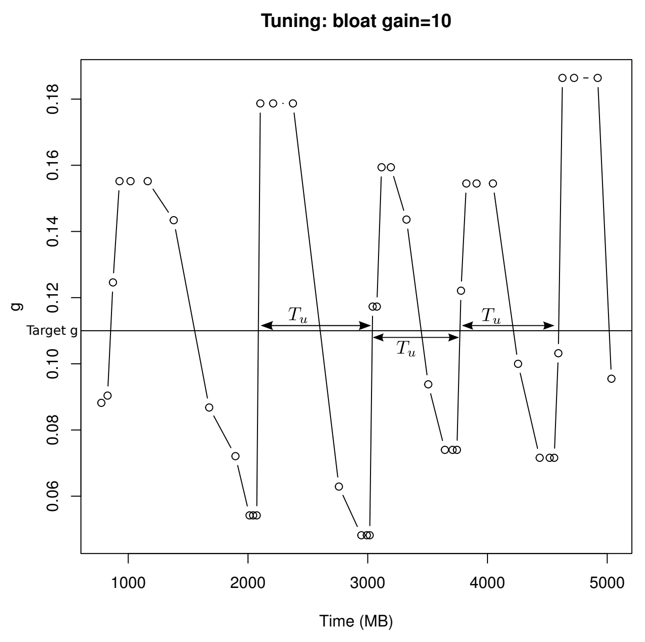


Figure 5: Experiment B: Example of an Oscillating System During Controller Tuning

able value. Note the decreasing amplitude as time progresses for both xalan graphs. There is an initial start-up period, which varies in length between the two. Similarly, the 2009 pmd and 2006 fop benchmarks, for example, settle well and the PID maintains a low error.

In contrast, the 2006 bloat and 2006 jython benchmarks are less stable and result in dramatic changes in heap size; the PID controller struggles to reduce the error that results from large temporal variations in GC overhead. The PID has a natural responsiveness defined by its parameters; in this case the PID provides very light damping which results in the poor, oscillating, response.

In these experiments, we allowed the PID to resize the heap almost without restraint, and we apply the calculated resize at every possible opportunity (i.e. immediately after every GC event). However, the main purpose of the PID is to provide a better calculation of what the heap size should be, rather than determining how frequently we resize the heap. Hence, it is possible to imagine subsampling the PID output, i.e. only executing a subset of the resizes indicated by the lower graphs. This would lead to a more stable heap size, at a potential cost of reducing the responsiveness of the PID and increasing deviation from the target g^* value. Another way to reduce the variation in heap size would be to further smooth the measurement of GC overhead, by further filtering of g.

To summarize, the PID works well at controlling the median GC overhead, but when g is subject to a large amount of variation, heap size will also vary greatly. This is likely due to high frequency content in the input signal causing some instability in the controller, which could be rectified by using a slightly smaller gain at the potential expense of some responsiveness. Alternatively it may be necessary to incorporate further control logic surrounding the PID or more intensive filtering, to find a balance the stability of heap size versus the efficiency gains of using a more responsive controller.

6.5 Experiment D: Comparative Evaluation on Phased Benchmark

In this study, we compose two DaCapo 2009 benchmarks to induce phased behavior. The artificial workload is two iterations of xalan followed by two iterations of sunflow, both with large inputs, The sequence is repeated many times, within the same VM instance. This behavioral profile may be similar to a Java application server which runs diverse jobs. Our objective is to provide an empirical comparison of different heap sizing mechanisms: default RVM/MMTk, Ergonomics, and PID controller.

We run all the phased benchmark experiments with the modified Jikes RVM build outlined above, using a full-heap mark/sweep GC. We set the initial and minimum heap size to 50MB, and the maximum heap size to 500 MB. We run each phased benchmark test for 500 full-heap GCs, which is always enough to change phase from xalan to sunflow several times.

For the RVM/MMTk default heap sizing policy (as outlined in Section 3.1) there are no parameters to set apart from the minimum and maximum heap size.

For the Ergonomics policy, we implement a simple case-based ergonomics scheme (as outlined in Section 3.2) in Jikes RVM. We use the same hard-wired parameters as in HotSpot, The full source code for this cut-down ergonomics reimplementation is available in our online repository [1]. Our ergonomics system does not support a GC pause time goal since we have no nursery generation to resize, but specifies an application throughput goal (from which we derive a GC overhead target) and a minimum and maximum heap size.

For the PID controller, we initially used the mean value of the corresponding PID parameter settings for xalan and sunflow as reported in Table 2. However we have reduced the proportional controller gain K_c to prevent the system clipping; this aids stability. We have reduced K_c to 0.75 times the mean value of xalan and sunflow gains from Ziegler-Nichols tuning, to iterate towards more desirable behavior. This is a common process in controller design, especially for non-linear systems where some manual tuning is often necessary to produce improved responses.

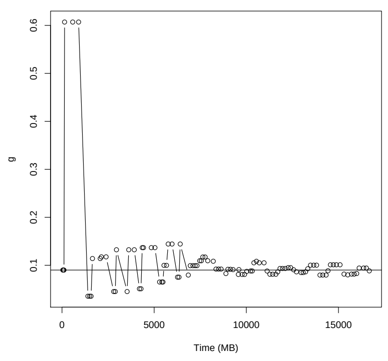
We set the GC overhead target g^* to 0.05, which is a representative value. We use this g^* value for both the PID controller and the ergonomics mechanism.

Figure 7 shows how the heap size changes over time with each policy. Workload phases are clearly marked on the graph. From the graph, we see that Ergonomics and PID heap sizing are more responsive than Jikes RVM. This is particularly noticeable at the beginning of a phase. Further, we see that Ergonomics is more conservative than PID in its heap size decrease actions (PID generally decreases earlier and further). Finally, there is a memory leak in this phased benchmark since the overall trend for all policies is to converge on the maximum heap size. This is caused by repeated classloader and recompilation activity bloating the immortal data region.

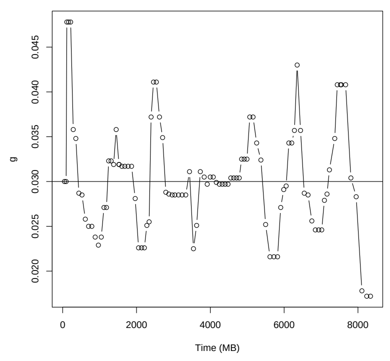
6.6 Discussion

One might ask why we did not use a *replay compilation* methodology when generating these heap sizing graphs. Our answer is that we want to demonstrate that our PID controller can be deployed in realistic (i.e. adaptive compilation) scenarios, rather than constrained experimental environments. Rather than steady state behavior, we are interested in the dynamic unstable behavior of initialization and phase change. In real use, the adaptive compiler operates and affects heap expansion. We did not want to ignore this.

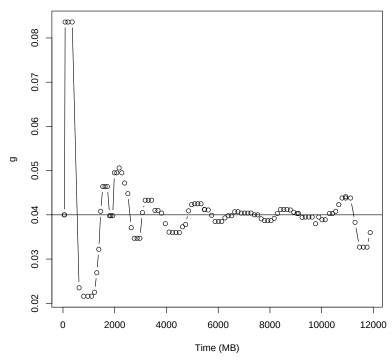
Similarly, one might ask why the results for each heap sizing policy are not drawn from multiple runs, and displayed with *confidence intervals*. The difficulty is that, in Jikes RVM, GC does not occur deterministically in relation to memory allocation (even when replay compilation is enabled). So each run of a benchmark



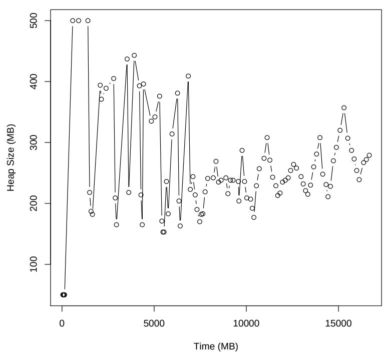
(a) GC Overhead for DaCapo 2009 pmd



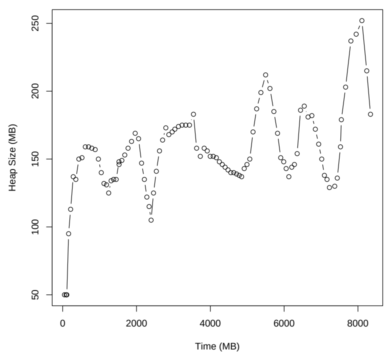
(b) GC Overhead for DaCapo 2009 sunflow



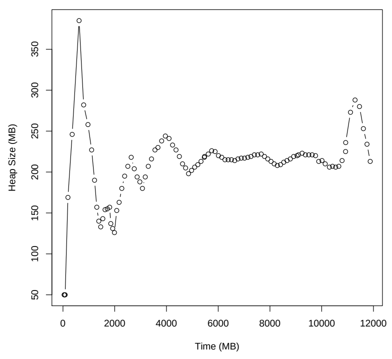
(c) GC Overhead for DaCapo 2009 xalan



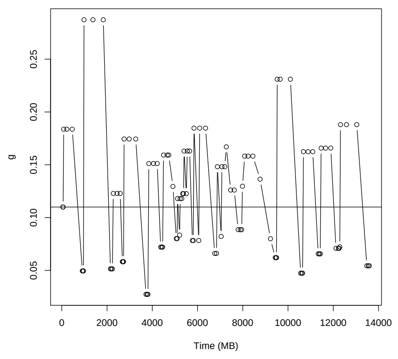
(d) Heap Size for DaCapo 2009 pmd



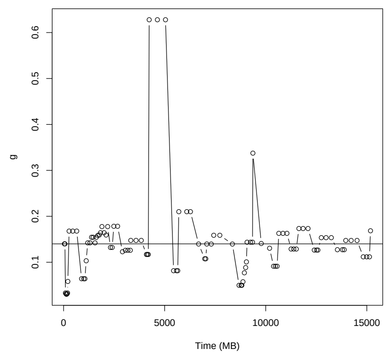
(e) Heap Size for DaCapo 2009 sunflow



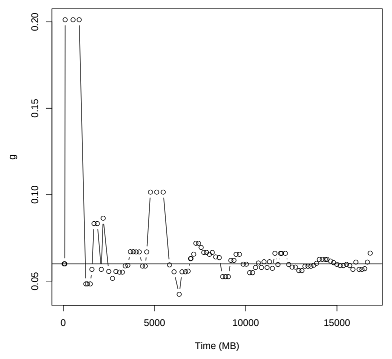
(f) Heap Size for DaCapo 2009 xalan



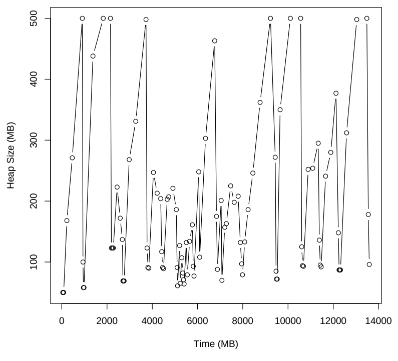
(g) GC Overhead for DaCapo 2006 bloat



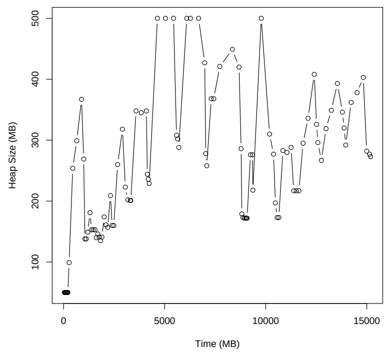
(h) GC Overhead for DaCapo 2006 eclipse



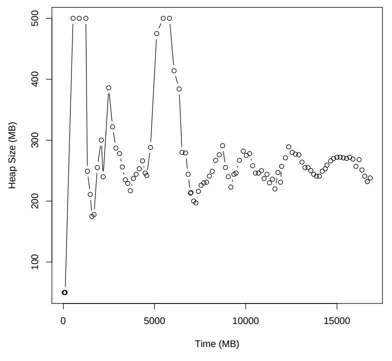
(i) GC Overhead for DaCapo 2006 fop



(j) Heap Size for DaCapo 2006 bloat

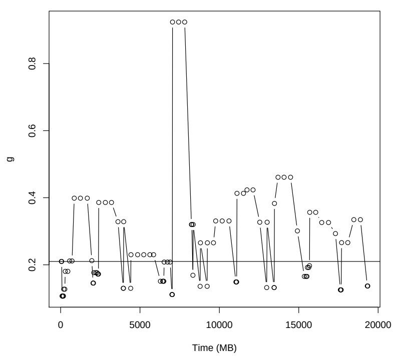


(k) Heap Size for DaCapo 2006 eclipse

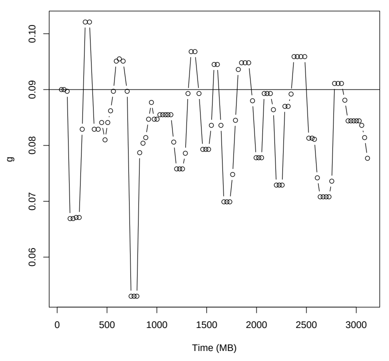


(l) Heap Size for DaCapo 2006 fop

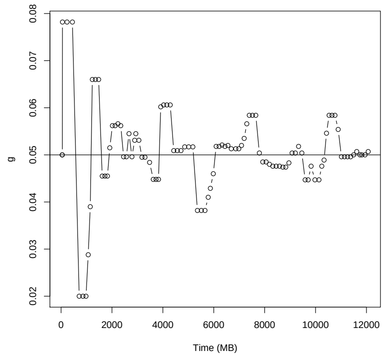
Figure 6: Evaluating the PID Controller on the DaCapo Benchmarks



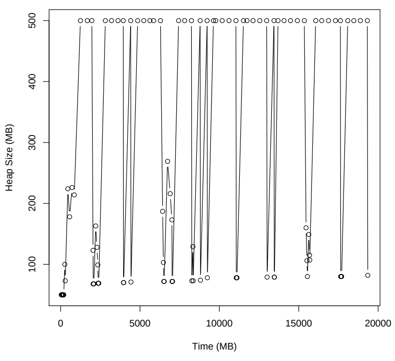
(m) GC Overhead for DaCapo 2006 jython



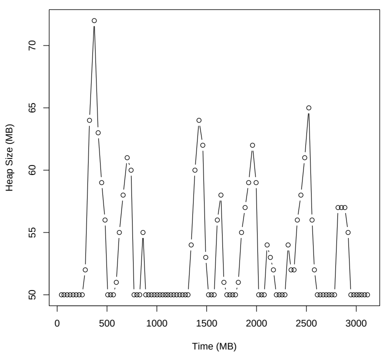
(n) GC Overhead for DaCapo 2006 luindex



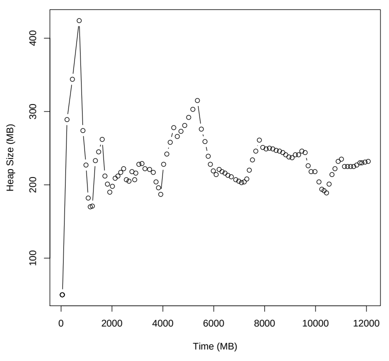
(o) GC Overhead for DaCapo 2006 lusearch



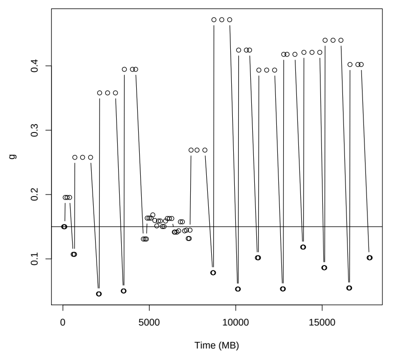
(p) Heap Size for DaCapo 2006 jython



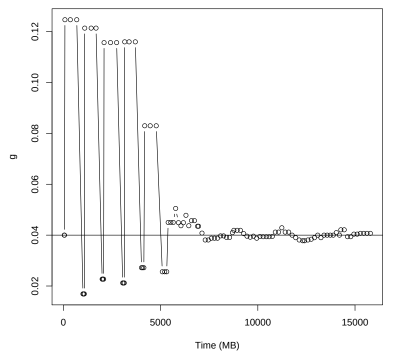
(q) Heap Size for DaCapo 2006 luindex



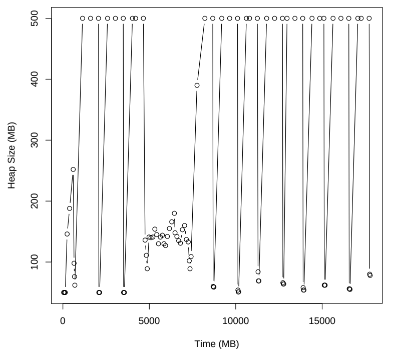
(r) Heap Size for DaCapo 2006 lusearch



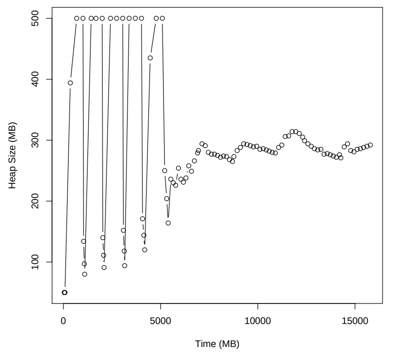
(s) GC Overhead for DaCapo 2006 pmd



(t) GC Overhead for DaCapo 2006 xalan



(u) Heap Size for DaCapo 2006 pmd



(v) Heap Size for DaCapo 2006 xalan

Figure 6: Evaluating the PID Controller on the DaCapo Benchmarks (continued)

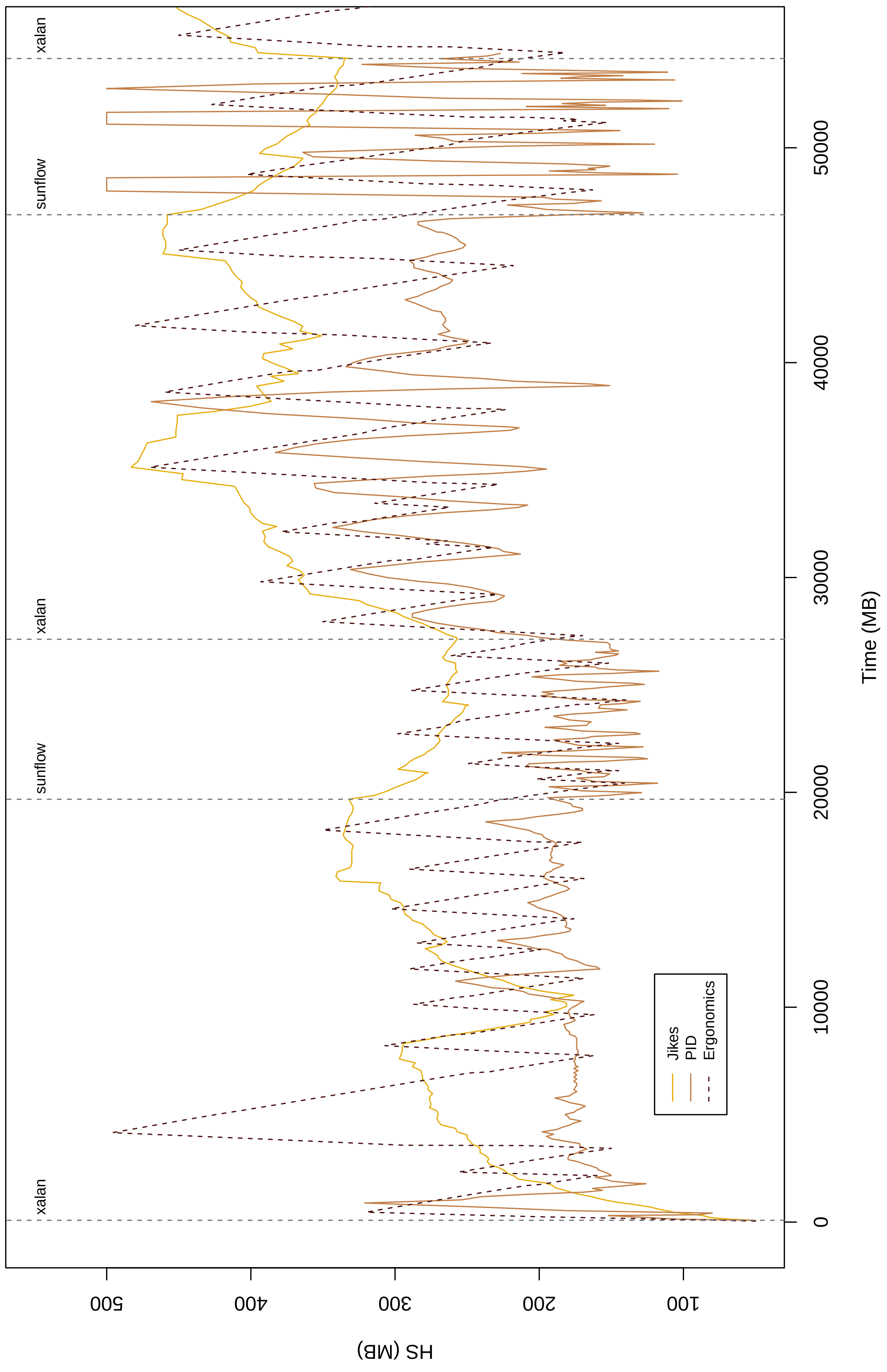


Figure 7: Comparison of three heap sizing policies on a phased benchmark

with a particular heap growth policy will give a different curve, with points at different x values.

We have deliberately omitted any evaluation of benchmark execution times with the three heap sizing policies. In truth, they are all largely similar. The key point is that, to reduce execution time, the goal-oriented policies (PID, Ergonomics) enable the user to specify a lower GC overhead target which will result in a correspondingly larger heap size. Jikes RVM on the other hand has no such facility, apart from the coarse-grained initial and maximum heap size parameters.

We measured the overhead of the heap resizing calculation and found it to be negligible. In a micro-benchmarking test, we ran 10 million iterations of each heap resizing method, supplying randomized input. The Ergonomics and PID resizing methods each took around 1 second to complete 10 million iterations. The default Jikes RVM resizing method took around 2 seconds to complete 10 million iterations. Jikes RVM takes longer because it does multiple array lookups for bounding, followed by multiple floating-point operations for interpolation. However the overhead of a single resize calculation (which happens once per GC) is minimal in comparison to the total cost of a GC.

7. Related Work

Heap sizing is a well-studied problem, and many researchers have attempted to provide better mechanisms. Most of these techniques have not been adopted by commercial VMs to our knowledge. (Note that we have already reviewed existing VM techniques in Section 3.)

7.1 Heuristic Approaches to Heap Management

Brecht et al. [12] discuss the concept of a 'sweet-spot' in heap size, similar to our own discussion in Section 2. They introduce a novel, heuristic-based, heap sizing mechanism for the Boehm GC. The heap grows by different amounts, depending on its current size in relation to a set of threshold values. The threshold values are hand-tuned for each system configuration, with the aim of reducing GC overhead while avoiding paging. Note that the heap size is never reduced; this is a restriction of the Boehm GC [11].

Most recent work on dynamic heap sizing has the explicit goal of avoiding paging (e.g. [19, 21, 34, 35]). Yang et al. [34, 35] employ reuse distance histograms and a simple linear model of the heap required; their approach requires modifications to the underlying OS. The Page-level Adaptive Memory Manager (PAMM) attempts to discover the optimal heap size for a number of applications running on a machine, taking advantage of software phase behavior [36]. The Isla Vista system uses allocation stalls as a warning of impending GC-induced paging [19] and resizes the heap accordingly. Isla Vista uses an additive increase / multiplicative decrease heap sizing policy, based on TCP congestion control. The heap grows linearly when there are no allocation stalls, and shrinks aggressively as soon as allocation stall activity is detected. Hertz et al. [21] use a region of shared scratchpad memory to allow concurrently executing VMs to exchange page fault and resident set size information in order to coordinate collections and heap sizes. The cooperative aspects of the memory manager are encoded using a fixed set of rules, known as Poor Richard's memory manager.

Alonso and Appel [3] describe a user-level advice service, that concurrent garbage-collected ML applications can query to determine whether to grow or shrink their runtime heaps. When an application requests advice, it passes the parameters describing its current state with respect to GC (e.g. current GC overhead and the proportion of live data on its heap). The advisor returns a ΔS value that specifies a heap size change for the application. The advisor uses a hard-coded, hand-tuned equation to determine values for ΔS based on this application's CPU time and memory size, in relation to the other concurrent applications.

7.2 Mathematical Models for Heap Management

Although it may seem like a somewhat subjective distinction, the following papers deal with heap resizing using principled mathematical models, rather than arbitrary heuristics.

Sun et al. [31] consider the problem of a single Java application server that has isolated, per-application heaplets within a single JVM. Each heaplet's size can be set independently. They introduce a simple analytical model, which adapts the size of all heaplets in order to equalize the GC frequency over all applications.

Tay and Zong [32] demonstrate how to derive a page fault equation that relates the number of page faults to actual heap size and resident heap size. Such an equation characterizes the behavior of a single program run with a specific input, given fixed GC and OS policies. Tay and Zong introduce a heap sizing rule (an equation that uses the same parameters as the page fault equation) and provide a coherent interpretation for its formulation. When the heap sizing rule is applied, the number of page faults during execution is minimized. The weakness in this approach is that the equations are not readily transferable to another situation, i.e. a change in application input, or transient system load would necessitate retuning to generate new parameter values for the page fault equation and heap sizing rule.

Vengerov [33] derives a mathematical equation to characterize the throughput (proportion of time spent in application execution rather than GC, i.e. $1 - g$ in our notation) of the generational collector in HotSpot. Based on this model, Vengerov then develops a mechanism for tuning the GC parameters so as to optimize the throughput. The major difficulty in estimating and optimizing throughput in HotSpot arises from its multiple generation spaces and decoupled GC policies in each space. Vengerov's work is relevant to ours, in that it controls heap size parameters and seeks to minimize GC time. However he tackles the problem in an entirely different way, using a *white-box* approach. As an expert GC analyst, he constructs a mathematical model of the whole system, then designs a custom tuning algorithm. We feel that our *black-box* approach is simpler. However we have only demonstrated it on a full-heap collector (i.e. no young generational spaces) where all throughput measurements are precise.

Singer et al. [27] use microeconomic supply and demand theory to characterize GC behavior. They apply the concept of elasticity to heap size, and devise a new elasticity-based approach to heap expansion. In the reported experiments, heap growth is rapid and difficult to control. Damping is required: this is not envisaged in their crude microeconomic framework, but would be implicitly provided in a PID controller. Another shortcoming of their heap sizing approach is that a target elasticity value is not an intuitive parameter for a user or system administrator to set. On the other hand, a controller target GC overhead is much easier to understand.

7.3 Control Theory for Heap Management

As far as we are aware, there are only two other instances of control-theoretic approaches to memory resource allocation [17, 29]. These are application-specific optimizations, rather than general VM mechanisms.

Storm et al. [29] deal with *autonomic database* configuration. They use control theory to implement a self-tuning memory manager that handles adaptive heap sizing for databases. A typical enterprise database has distinct heaps for various memory-intensive features. e.g. compiled SQL cache, buffer pool, sort memory. The solution proposed by Storm et al. uses a cost/benefit estimation model for resizing individual heaps, with an overall tuning objective of equalizing the cost/benefit metrics for all heaps. The tuning is accomplished using a multi-input multi-output (MIMO) controller with an integral control law (cf. the I component of a PID controller). They enumerate the advantages of a controller-based tuning approach as (i) fast convergence, (ii) rapid adaption, and (iii) stable response to noise.

Gandhi et al. [17] use control theory to improve performance of the *Apache web server*. The two high-level system outputs that their controller attempts to optimize are CPU and memory utilization metrics. The system administrator must set desired values for CPU and memory utilization. The two controller inputs are Apache tuning parameters for (i) the maximum number of simultaneous clients, and (ii) the pause time on an http client connection before it is closed. The web server is modeled as a black box, and characterized using experimental data. The controller is a simple MIMO proportional integral controller (cf. the P and I components in a PID controller). Controller parameter values are tuned using pole placement and linear quadratic regulator techniques. There is limited system performance evaluation in the paper.

We note that the application of control theory to computer systems, and particularly cloud-based resource sharing virtualized systems, is a growing area of research activity [20, 23, 25, 37].

8. Conclusion

8.1 Summary

In this paper, we have proposed the use of control theory for dynamic heap sizing of garbage-collected applications. We have described the deployment of a PID controller in the Jikes RVM memory management system. We have characterized the behavior of this heap size controller on a set of standard Java benchmarks, and compared it with two existing, heuristic-based heap sizing mechanisms.

Our goal in employing control theory is as much to *provide a rigorous approach* as it is to provide a near-optimal solution. So long as our controller is robust and competitive with hand-crafted alternatives, then we propose that its solid foundation should prove a compelling argument for its adoption.

8.2 Discussion of Limitations

In one sense, it is difficult to make a fair comparison between our PID controller and the existing heap sizing mechanism in Jikes RVM, since it is not clear what the current system is trying to optimize, whereas the PID controller has an explicit goal. We hope eventually to design a controller that frees the user of the need to specify a target GC overhead, which will enable a more straightforward comparison. This may require the application of *optimal control theory*.

An obvious limitation is the need to *tune* a PID controller for a specific scenario. As we discuss in Section 6.3, the parameter values are fairly similar across the range of DaCapo benchmark workloads. For clearly distinct workloads, one can use *gain scheduling* to swap in a new set of parameters. In general, most heuristic-based approaches require some amount of tuning effort, so this is a common weakness. All tuning was performed manually in our experiments, however automated PID tuning packages are widely used in industrial settings.

All the experiments reported in this paper use a full-heap, mark/sweep GC. This seems to be a useful base case to demonstrate our new control-theoretic technique. We have not examined generational copying collection at all. We expect that the same techniques should be applicable to generational GC, but that the process of interacting with the controller will be more complex. We note that both Jikes RVM and HotSpot use their heuristic heap sizing mechanism for both generational and non-generational GC, possibly with different growth ratio parameters. We also note that Vengerov's work [33] on computing overall GC overhead from nursery GC overhead may be applicable.

Another potential concern is the possibility of address space fragmentation caused by excessively frequent heap size changes, particularly with non-moving collectors. In defense of our scheme, we observe that all production VMs support adaptive heap sizing, and there is a general complaint from users that VMs are 'not ramping up the heap size quickly enough.' Further, modern memory managers like MMTk support the mapping of logical heap spaces onto discontiguous region in virtual address space.

So far, our controller does not support paging avoidance, i.e. it does not account for the right-hand half of the sweet-spot curves in Section 2. The currently deployed Jikes RVM heap resizing mechanism is also oblivious to paging. For future work, we hope to incorporate a second controller (using a subsumption model) that will reduce the heap size if it detects paging activity. Such a compound controller would drive an application's heap size towards the sweet-spot region of execution automatically and adaptively.

8.3 Future Work

In addition to extending our control-theoretic system to handle generational collectors and paging avoidance, we have a more ambitious objective.

We envisage a set of VM instances, executing concurrently on a manycore server. Each VM has its own low-level heap resizing mechanism, similar to the PID controller described in this paper. However a higher-level meta-controller is needed at the system level, to ensure that all the VMs co-operate fairly, or in a manner that satisfies (possibly diverse) client policies. We imagine this meta-controller will drive the target variables of the underlying controllers, using some kind of statistical, economic or game theory model.

Acknowledgments

We are grateful to Tony Printezis and Mario Wolczko for confirming our understanding of the HotSpot heap resizing mechanism. We thank David Matthews and Sophia Drossopoulou for commenting on an early draft of this paper. We also acknowledge the help of the anonymous referees.

This research was partly supported by the London Mathematical Society and the Scottish Funding Council through the SICSA project.

References

[1] Experimental resources. `http://sf.net/p/jikesrvm/research-archive/40`.

[2] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. The resource-as-a-service (RaaS) cloud. In *USENIX Conference on Hot Topics in Cloud Computing*, 2012.

[3] R. Alonso and A. W. Appel. An advisor for flexible working sets. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1990.

[4] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1), 2000.

[5] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, V. Sarkar, and M. Trapp. The Jikes research virtual machine project: Building an open source research community. *IBM Systems Journal*, 44(2):1–19, 2005.

[6] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Communications of the ACM*, 53:50–58, 2010.

[7] M. Arnold, M. Vechev, and E. Yahav. QVM: an efficient runtime for detecting defects in deployed systems. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 143–162. ACM, 2008.

[8] S. M. Blackburn, Perry Cheng, and K. S. McKinley. Oil and water? high performance garbage collection in Java with MMTk. In *Proceedings of the 26th International Conference on Software Engineering*, pages 137–146. ACM, 2004.

[9] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM, 2006.

[10] S. M. Blackburn, R. E. Jones, K. S. McKinley, and J. E. B. Moss. Beltway: Getting around garbage collection gridlock. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 153–164. ACM, 2002.

[11] H.J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software: Practice and Experience*, 18(9):807–820, 1988.

[12] T. Brecht, E. Arjomandi, C. Li, and H. Pham. Controlling garbage collection and heap growth to reduce the execution time of Java applications. *ACM Transactions on Programming Languages and Systems*, 28:908–941, 2006.

[13] R. Buyya, S. Y. Chee, and S. Venugopal. Market-oriented cloud computing: Vision, hype, and reality for delivering IT services as computing utilities. In *Proceedings of High Performance Computing and Communications*, pages 5–13, 2008.

[14] A. Datta, M.T. Ho, and S.P. Bhattacharyya. *Structure and synthesis of PID controllers*. Springer, 2000.

[15] E. Duesterwald, C. Cascaval, and Sandhya Dwarkadas. Characterizing and predicting program behavior and its variability. In *Proceedings of Parallel Architectures and Compilation Techniques*, 2003.

[16] D. Durkee. Why cloud computing will never be free. *Queue*, 8:20:20–20:29.

[17] N. Gandhi, D.M. Tilbury, Y. Diao, J. Hellerstein, and S. Parekh. MIMO control of an Apache web server: modeling and controller design. In *Proceedings of the American Control Conference*, 2002.

[18] David Grove. Private Communication, 2011.

[19] C. Grzegorczyk, S. Soman, C. Krintz, and R. Wolski. Isla vista heap sizing: Using feedback to avoid paging. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2007.

[20] J. Hellerstein, S. Singhal, and Qian Wang. Research challenges in control engineering of computing systems. *IEEE Transactions on Network and Service Management*, 6(4):206–211, 2009.

[21] M. Hertz, S. Kane, E. Keudel, T. Bai, C. Ding, X. Gu, and J. E. Bard. Waste not, want not: resource-based garbage collection in a shared environment. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Memory Management (ISMM)*.

[22] R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall, 2012.

[23] C. Karamanolis, M. Karlsson, and X. Zhu. Designing controllable computer systems. In *Proceedings of the 10th conference on Hot Topics in Operating Systems*, 2005.

[24] N. S. Nise. *Control Systems Engineering*. John Wiley & Sons, Inc., 3rd edition, 2000.

[25] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*. ACM, 2007.

[26] S. P. Reiss. Dynamic detection and visualization of software phases. In *Proceedings of the 3rd International Workshop on Dynamic Analysis*. ACM, 2005.

[27] J. Singer, R. E. Jones, G. Brown, and M. Luján. The economics of garbage collection. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Memory Management (ISMM)*. ACM, 2010.

[28] S. Soman, C. Krintz, and D. F. Bacon. Dynamic selection of application-specific garbage collectors. In *Proceedings of the 4th ACM SIGPLAN International Symposium on Memory Management (ISMM)*. ACM, 2004.

[29] A. J. Storm, C. Garcia-Arellano, S. S. Lightstone, Y. Diao, and M. Surendra. Adaptive self-tuning memory in DB2. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, 2006.

[30] Sun. Garbage collector ergonomics. `http://docs.oracle.com/javase/1.5.0/docs/guide/vm/gc-ergonomics.html`.

[31] K. Sun, Y. Li, M. Hogstrom, and Y. Chen. Sizing multi-space in heap for application isolation. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM, 2006.

[32] Y.C. Tay and X.R. Zong. A page fault equation for dynamic heap sizing. In *Proceedings of the first joint WOSP/SIPEW International Conference on Performance Engineering*, 2010.

[33] D. Vengerov. Modeling, analysis and throughput optimization of a generational garbage collector. In *Proceedings of the 9th ACM SIGPLAN International Symposium on Memory Management (ISMM)*. ACM, 2009.

[34] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. Cramm: virtual memory support for garbage-collected applications. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. ACM, 2006.

[35] T. Yang, E.D. Berger, M. Hertz, S.F. Kaplan, and J.E.B. Moss. Automatic heap sizing: Taking real memory into account. In *Proceedings of the 4th ACM SIGPLAN International Symposium on Memory Management (ISMM)*. ACM, 2004.

[36] C. Zhang, K. Kelsey, X. Shen, C. Ding, M. Hertz, and M. Ogihara. Program-level adaptive memory management. In *Proceedings of the 5th ACM SIGPLAN International Symposium on Memory Management (ISMM)*. ACM, 2006.

[37] X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, P. Padala, and K. Shin. What does control theory bring to systems research? *SIGOPS Operating Systems Review*, 43:62–69, 2009.

[38] J. G. Ziegler and N. B. Nichols. Optimum settings for automatic controllers. *Transactions of the American Society of Mechanical Engineers*, 64:759–768, 1942.