University of Glasgow | School of Computing Science

Control Theory for Dynamic Heap Resizing

Blair Archibald (1002105)
1002105a@student.gla.ac.uk

ESE Project (70 Page Limit)

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Level 4 Project — March 28, 2014

Abstract

Obtaining high performance from garbage collected runtime systems is hard and increasing the performance of a runtime system generally requires manual parameter tuning. This is not cost effective; in many cases requiring program profiling, expert knowledge and trial and error techniques. We propose that the runtime system itself should be responsible for tuning its parameters to give high performance. With focus placed on the garbage collection system we present one such method of automatic tuning. We show how a memory management system can be viewed as a control system and using ideas from control theory implement a proof of concept controller system for the HotSpot Java virtual machine. From an evaluation of the system we show how even a very simple controller design can have performance comparable to that of the current state of the art heap resizing algorithms; in many cases we also show a significant improvement in heap memory usage making our system suitable for constrained and multi-tasking devices.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: _____ Signature: _____

Contents

Chapter 1

Introduction

Automatic memory management in the form of garbage collection is becoming increasingly popular within modern programming language implementations. The main idea is to allow programmers to write programs without worrying about explicit management of the heap. Removing explicit memory management attempts to make writing code simpler and reduce the chances of *memory leaks* and other memory related issues prevalent in many *hand allocated* programs.

Garbage collected runtime systems have been around since the *Lisp* programming language in 1959, however it has not been until recently with the increased popularity of the *Java* programming language that garbage collection has become mainstream. Many other programming languages, both functional and imperative, are following suit making this a particularly relevant and interesting topic for study.

There are various space and time time complexities present when using garbage collection. These complexities require careful management in order to maximise runtime system performance and, in turn, program performance. In the general case this requires the tuning of various garbage collection parameters such as generation sizes, amount of parallelism and garbage collection algorithm used. In many cases, in order to gain maximum optimisation applications need to be memory profiled and performance tuned by hand.

Tuning is a difficult task requiring expert knowledge and significant effort dedicated to program profiling. In many cases there is an element of trial and error while tuning the system, this is very costly in terms of resources (both human and computer). To further motivate the issues with garbage collection tuning, an example[13] of some standard Java virtual machine tuning parameters follows:

```
java -Xmx12g -XX:MaxPermSize=64M -XX:PermSize=32M  -XX:MaxNewSize=2g
-XX:NewSize=1g -XX:SurvivorRatio=128 -XX:+UseParNewGC
-XX:+UseConcMarkSweepGC -XX:MaxTenuringThreshold=0
-XX:CMSInitiationOccupancyFraction=60 -XX:+CMSParallelRemarkEnabled
-XX:+UseCMSInitatingOccupancyOnly -XX:ParallelGCThreads=12
-XXlargePageSizeInBytes=256m ...
```

By requiring specialised tuning, garbage collected systems do not introduce the simplicity they claim. They have instead moved memory management considerations from the code implementation phase into a separate optimisation stage. We wish to remove the necessity for manual performance tuning and instead provide automated methods for maximising performance gains.

It is difficult to determine ahead of time the amount of memory which a program will require throughout its lifetime. Consider a program using a fixed-size heap, if the running program (mutator) has a high allocation

rate then garbage collection will occur often. A high number of garbage collections can be very detrimental to performance as a program may spend more time performing garbage collections than doing useful work. This is similar to paging within a virtual memory system and can be particularly noticeable in applications involving user interaction.

It is possible however to reduce the time spent performing garbage collection by giving the program more memory. We wish to be conservative in memory use as it is still a limited resource; although memory is becoming cheaper there are many devices that use 4 gigabytes or less of memory[1]. This becomes more of an issue when you consider the multi-tasking nature of modern systems with all running programs sharing the same physical memory[2].

A method is required for finding the optimum memory size which reduces the average time spent in garbage collections whilst being as conservative with memory as possible. This is becoming particularly important for distributed systems. Many industrial applications are moving to virtualised services in the cloud. By ensuring conservative memory usage and high throughput we can allow many applications to run on one hypervisor with both high performance and a reduced chance of paging.

1.1 Contributions

This report makes several key contributions:

- Provides a survey of current state of the art heap resizing algorithms.
- Provides a mapping between the fields of control theory and memory management.
- Shows how the use of a mathematical model in the creation of a goal-based system can simplify heap resizing algorithms.
- Via a proof of concept system shows how a simple, control-theory-based controller design can be used to achieve similar performance to the current state of the art heap resizing algorithm present in the OpenJDK (version 7u).

1.2 Outline

The report has the following structure:

Chapter 2 Provides an introduction to garbage collection techniques giving key definitions of terms used throughput this report.

Chapter 3 Presents a survey of modern runtime systems and their heap sizing policies.

Chapter 4 Introduces control theory as an approach to designing goal based systems. This motivates the ideas behind viewing dynamic heap resizing as a control theory problem.

Chapter 5 Focuses on related attempts to use control theory to automate performance tuning within a computing system.

[1] Many mobile or embedded devices contain much less memory than a conventional desktop PC.
[2] We do not account for non uniform memory architectures here.

Chapter 6 Discusses the implementation and design of dynamic heap resizing algorithms for the *HotSpot* (OpenJDK 7) Java runtime system.

Chapter 7 Presents a mathematical system model and uses it to derive various controller designs. These designs are analysed, evaluated and compared.

Chapter 8 Summarises the limitations of our controller design and provides pointers to future work.

Chapter 9 Ends this report with some concluding thoughts and personal reflection on the project as a whole.

Chapter 2

Introduction to Garbage Collection Techniques

This chapter provides a high level overview of numerous garbage collection techniques including key definitions which shall be used throughout this report. For a fuller introduction to garbage collection techniques see Wilson[18] who presents a comprehensive survey paper of the basic techniques. Many of the basic memory system elements are shown graphically in Figure 2.1. This figure is intended to be used as a reference when reading this section.

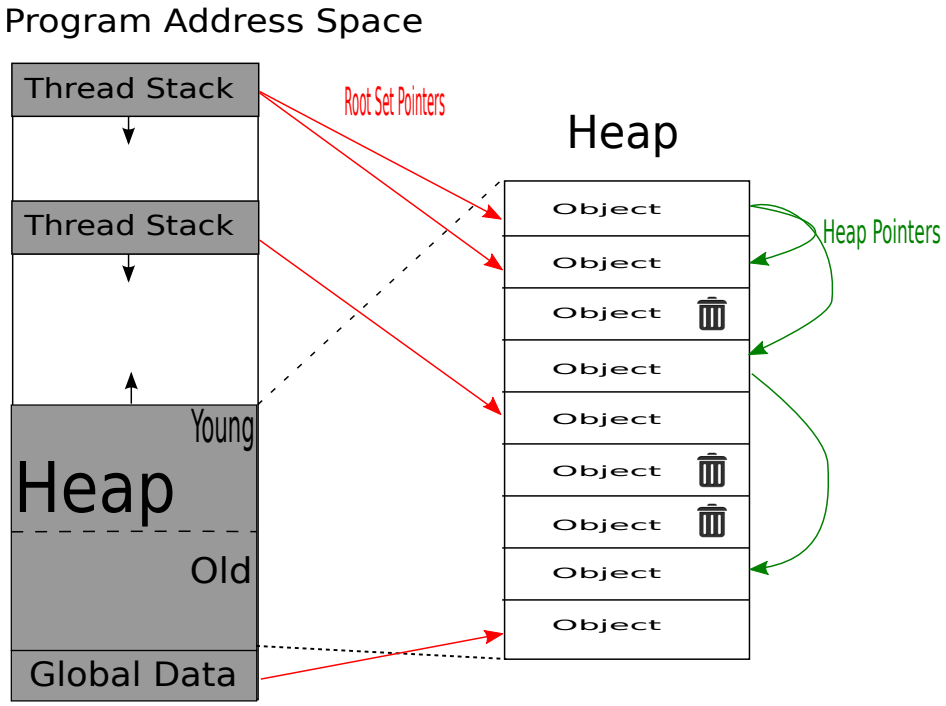


Figure 2.1: Schematic diagram of a basic memory system

Computer programs typically divide their data memory into two areas known as stack and heap memory. Within a multi-threaded system each thread is given their own stack area of memory (from the global stack area) which is used for maintaining function call stacks and performing local variable allocations. On the other hand heap memory is generally used for dynamic allocation (when the amount of memory needed is unknown at compile time) and typically stores objects which are either shared between multiple threads or too large to effectively fit within the constrained stack size – an example object may be a large array spanning several megabytes.

Objects on the heap are generally accessed by reference to a particular memory address.

Management of stack and heap memory is performed in two different ways. The stack automatically removes locally allocated variables when a function returns, this is very fast as we simply need to move our stack pointer to the previous frame. The heap is generally managed explicitly by the programmer who performs allocation and deallocation through the use of library functions (malloc()/new and free()/delete are common). This technique is error prone; if a programmer forgets to deallocate memory then this memory is never reclaimed and cannot be used for the rest of the program run – this is known as a memory leak. There is also an associated overhead for each memory deallocation while the library manages its internal heap data structures (typically some form of buddy heap system is used).

Garbage collection (GC) reduces the risk of programmer error by performing automatic heap memory management. A garbage collected system does not depend on the programmer explicitly deallocating memory, but instead automates this process by detecting when allocated memory is no longer accessible by a program and therefore must be available to use again.

Garbage collection takes many forms. Common garbage collection systems traverse the heap to determine which objects[1] are reachable. There are two main steps in this process. Firstly a *root set* is constructed. The root set contains all variables which are currently accessible (by any thread) including global variables. We then follow any pointer variables into the heap and continue to follow pointers in a depth-first or breadth-first fashion in order to locate all reachable objects. Reachable objects are known as the *live* objects. An object which is not reachable may be considered as *garbage* and the memory can be reused. The algorithms differ in what happens when a live object is reached. A *mark-sweep* collector uses a special mark bit within an object header to denote liveness, once all objects are marked the algorithm searches for unmarked objects and returns these to the allocation pool (in most cases some form of free list). A *copying* collector on the other hand copies any live objects into a known free area, this has the effect of compacting the memory and allowing contiguous allocation at the cost of data copying overheads.

An alternative to heap traversal garbage collection algorithms is *reference counting*. With this system we keep track of how many references point to each heap object. During any pointer updates to a particular object we add or subtract to the object's reference count as necessary. Should the reference count reach zero we can free the object as it is no longer reachable. Reference counting has increased computational complexity during pointer updates but operates concurrently with the running program, avoiding the need for a separate garbage collection phase.

A garbage collector which uses heap traversal is usually invoked when the system has ran out of memory and requires space to be freed in order to continue operation. It is possible however to perform our marking phase at two different times. In the first case we wait for an out of memory condition, all program threads are paused, and then we proceed with heap traversal. This is known as a *stop the world* collection. As an alternative, we can perform heap marking incrementally in the background while the program continues to run. When an out of memory condition is reached we only need to perform a sweep phase as the objects are ready marked. This approach is complicated by the fact the root set and references may change once an object has already been marked. These updates are tracked by the garbage collector such that the marking is consistent. Notice that we can afford to mark a garbage object as alive but *never* afford to mark a live object as garbage if we wish to maintain memory consistency.

The nature of allocations found in most applications implies that newly allocated objects will only be live for a short period of time; this is known as the *weak generational hypothesis* [16]. As an example consider the code sample shown below:

[1] An object refers to any heap structure rather than the object orientated term.

```
1     for (int i = 0; i < 5000; i++) {
2         Object o = new Object();
3         doSomething(o);
4     }
```

Listing 2.1: Weak generational hypothesis example

In this program we allocate 5000 objects onto the heap, however each object is only used once. Assuming that *doSomething()* does not add global references to these objects we see that the lifetime of an object is a single loop cycle; this is a very short lifetime. Some garbage collection systems exploit the weak generational hypothesis by dividing the heap into several sub heaps known as *generations*. This is known as generational garbage collection. This new heap structure allows each sub heap to be collected independently of the others reducing the time required to perform each garbage collection (pause time). The main drawback of generation garbage collection is how to handle references which cross generation boundaries – an object may be kept live by a reference from another generation. The solution is to track cross generation references through via *read/write barrier*, the details of which we omit here for simplicity.

Many generational garbage collection systems split the heap into two main generations known as *old* and *young*. Objects are allocated into the young generation which is collected during a *minor* collection. If an object survives a minor collection (or possibly several) it is promoted to the old generation on the principle that it is likely to be a long living object. A *major* collection collects both the young and old generations, this process takes significantly longer than a minor collection as the whole heap must be traversed.

We return to garbage collection algorithms in Chapter 6 where we shall discuss garbage collection details for the HotSpot Java Virtual Machine.

Chapter 3

Runtime Environments

This chapter explores three runtime systems, the Glasgow Haskell Compiler, the Jikes Research Virtual Machine and the OpenJDK HotSpot Java Virtual Machine in order to address their suitability for research and explore the current state of the art dynamic heap sizing implementations.

3.1 Glasgow Haskell Compiler

The *Glasgow Haskell Compiler*[4] (GHC) provides both a compiler and a runtime system for use with the Haskell programming language.

The Haskell programming language is a pure and lazy functional language. Lazy evaluation is an evaluation strategy in which a function is only evaluated fully if its result is needed. Contrast this to strict evaluation in which functions are always evaluated when they are encountered. For example $f(g(x))$ will always evaluate $g(x)$ whether f uses its argument or not. The use of lazy evaluation impacts directly on the memory system. Programs typically require large amounts of memory to allow the storage of unevaluated functions known as *thunks*. Garbage collection is used within GHC to ensure that memory becomes available once an evaluation is eventually forced. The increased memory requirements makes it essential we recycle memory as quickly as possible to avoid paging.

GHC uses a *block structured heap*[6] in which memory is split into fixed size blocks which may be chained together to form a block group (generation). This approach is beneficial when developing dynamic resizing algorithms for the following reasons:

- By simply grouping blocks together we can create as many generations as required.
- Resizing a generation is a simple case of adding or removing blocks from the group descriptor. This also ensures that we maintain heap alignment invariants.

3.1.1 Heap Sizing Policy

The GHC runtime attempts to resize all generations after a major garbage collection (all generations collected). It does this using the following heuristic:

> "make all generations except zero the same size. We have to stay within the maximum heap size, and leave a certain percentage of the maximum heap size available to allocate into"[1].

The sizing policy for a non-generational collector is different, we focus solely on the generational collector here as this is in common use.

The size to which GHC attempts to scale all generations is based on a live estimate. To determine the oldest live estimate GHC sums the number of marked bits within the oldest generation. The formula to calculate the generation size is as follows, where Old Gen Factor defaults to 2.

$$\text{Live} = \frac{\text{Oldest Live Estimate} + \text{Block Size} - 1}{\text{Block Size}} + \text{Number of Large Obj Blocks in Oldest} \tag{3.1}$$

$$\text{Generation Size} = \text{Live} \times \text{Old Gen Factor} \tag{3.2}$$

If the combined generation sizes using this calculated size goes over the maximum heap size then size adjustments are made such that the maximum heap size invariant is maintained. The algorithm used to adjust the sizes is not reproduced here for simplicity.

There is no indication given as to why twice the current live data amount would make a good generations size. It appears, like many resizing algorithms, to be purely based on a heuristic.

The sizing policy for the *nursery* (youngest) generation is different due to this being the area where allocations occur. To calculate the nursery size we calculate how much space will be needed to collect all older generations assuming all data remains live (this is important since GHC uses a copy collector and *always* needs free space to copy into). We also calculate an estimate of the *young live data*. From this the blocks needed are determined as follows, where Suggested Heap Size is provided by the user or the size calculated during a major GC via the method above.

$$\text{Blocks} = \frac{\text{Suggested Heap Size} - \text{Space Needed To Collect All Older Generations} \times 100}{100 \times \text{Percentage Young Live}} \tag{3.3}$$

The ease of implementing new generation sizing policies within GHC makes it a potential candidate for dynamic heap resizing research. However the nature of lazily evaluated languages has a significant effect on the allocation and collection rates of a program. The controller ideas developed within this report rely on tracking program allocation rates and for this reason GHC was discarded as a potential research system.

3.2 Jikes Research Virtual Machine

The *Jikes Research Virtual Machine*[14] (Jikes RVM) is an open source runtime system implementation for the Java programming language written in Java. It is designed primarily for the purpose of performing runtime system research and as such flaunts a very clean and modular code base. The *Memory Management Toolkit* provides a collection of classes and interfaces designed to allow for easy experimentation with memory systems. A combination of these features make Jikes a viable system for use in memory management research. Jikes however, being focused on research rather than production, does not fully emulate all the features expected from a production Java virtual machine, there are also known stability issues. Not having a fully implemented Java virtual machine specification causes difficulty when attempting to run certain benchmarks supporting only a

		Live Ratio					
	0.00	**0.00**	**0.10**	**0.30**	**0.60**	**0.80**	**1.00**
	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
GC Load	**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 3.1: Jikes *HeapGrowthManager* scaling function (generational)

fraction of the latest Dacapo 9.12 benchmark suite; the most recently fully supported benchmark suite being Dacapo 2006-MR-2 released in 2007[15].

These factors combined led to a production Java virtual machine being chosen over Jikes for this research.

One of the key features of the Jikes (RVM) is the *Memory Management Toolkit* (MMTk). The MMTk allows for easy experimentation with new garbage collection techniques and as such Jikes is compatible with a variety of garbage collection algorithms. The MMTk contains a basic dynamic heap sizing algorithm which is queried after each garbage collection to find a suitable heap size. This is much less advanced than the Ergonomics system found within HotSpot.

3.2.1 Heap Sizing Policy

Jikes uses two parameters to determine a suitable heap scaling ratio after each garbage collection. These parameters are fed into a lookup-table-based *scaling function* which is shown in Table 3.1. There are two lookup-table variants within Jikes; one for non-generational collectors (with a single heap section) and another for generational collectors (where the heap is split into multiple sections). We reproduce the generational function here. Apart from the look-up function itself, all other details are the same in both the generational and non-generational case.

The two parameters used to index the lookup-table are the *live ratio* and *garbage collection load*. Methods for calculating these parameters are given below:

$$\text{Live Ratio} = \frac{\text{Maximum Heap Size}}{\text{Current Heap Size}} \tag{3.4}$$

$$\text{GC Load} = \frac{\text{Time Taken By GC}}{\text{Time Since Last GC}} \tag{3.5}$$

In the case that the live ratio or garbage collection load do not match a index value exactly then a point interpolation is performed. Using point interpolation we can view the lookup-table as a continuous function of scaling ratios, rather than an indexed table of discrete scale amounts.

The live ratio is a measure of how much memory we are currently using. Jikes is attempting to minimise memory usage. GC load is a ratio of the pause times (time performing GC) to the separation of garbage collections. This can been seen as the inverse of a throughput measure, which we wish to minimise.

There are several disadvantages to the approach taken by Jikes. Although the table appears to be scaling in the correct direction based on what we know of the parameters, there is no system model presented as to why

these values particular values should be chosen. There is also an associated overhead to both maintain the lookup table and perform the interpolation functions. In general we wish to minimise the size and computation costs of a runtime system where ever possible. We note however that this approach does have the benefit of simplicity.

3.3 HotSpot Java Virtual Machine

HotSpot[8] is a runtime system for the Java programming language. It is the runtime system deployed within the *Open Source Java Development Kit* (OpenJDK). OpenJDK is a very popular Java implementation for both business production systems and home use. The maturity and industrial backers (mainly Sun/Oracle) for HotSpot makes it a very stable system capable of running most benchmarks. Although the learning curve is steeper than that of Jikes it is still a valid virtual machine to perform research on.

HotSpot is a large codebase of approximately 500,000 lines written mainly in the C++ programming language. Although this presents possible implementation and comprehension challenges, the object orientated nature provides a clear separation of concerns between components of the system. HotSpot also provides multiple different garbage collection algorithms for experimentation – at the time of writing there are 4 collectors[1], however in some cases you can use multiple at once to collect different generations.

HotSpot contains a dynamic heap parameter tuning system known as *Ergonomics*, which allows the runtime system to perform dynamic heap resizes to try and match a set of user-specified goals. The Ergonomics system is the main competitor to our controller design and comparisons will be be drawn between the two.

The multiple garbage collection algorithms available, maturity, stability and the availability of a state of the art heap sizing algorithm, make HotSpot an ideal candidate for the investigation of new dynamic heap sizing designs and is used as the basis for this research.

3.3.1 Heap Sizing Policy – HotSpot Ergonomics

Ergonomics allows a user to specify required performance parameters and let the system automate the rest of the process (if possible) without the need for further parameter tuning. This mirrors the ideas presented within our controller designs. Internally to HotSpot, Ergonomics is simply referred to as an *AdaptiveSizePolicy*.

Ergonomics is goal-based system which allows the user to specify the following goals[9]:

- Maximum pause time goal.
- Throughput goal.
- Minimum footprint goal.

That is, the system will use as much memory as needed (up until the maximum heap size) in order to achieve (or come close to) the required performance.

These goals have a priority ordering. A flowchart of the system priorities is given in Figure 3.1. The *GC Cost* metric for determining whether or not to adjust for throughput is calculated in a similar manner to the *GC Load* metric used within Jikes, which is calculated as shown in Equation 3.5. The size adjustments, shown in brackets, give the general scale method used by Ergonomics to move towards the required goal. However, Ergonomics may choose a different scale method if it has not yet gathered enough statistics.

[1] Soon to be 5 with the addition of the Shenandoah garbage collector.

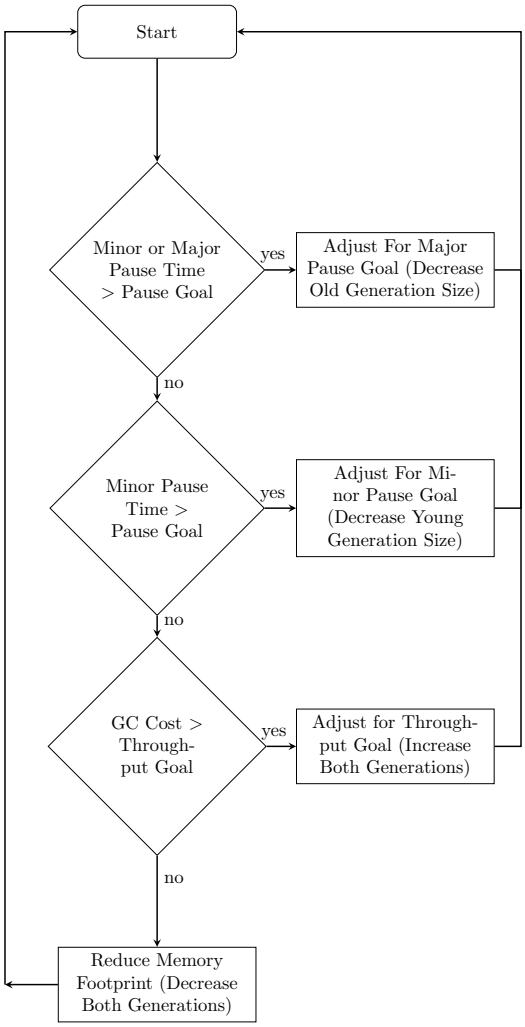


Figure 3.1: Ergonomics Goal Priorities

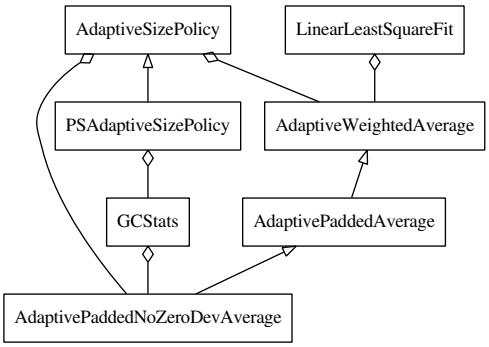


Figure 3.2: Ergonomics class diagram

Ergonomics collects statistical information about each garbage collection in order to *predict* how the system is operating and whether or not its goals are being met. The statistical techniques are centralised, however a garbage collection algorithm may wish to implement its own specialised versions of the heap size policy in order to optimise for the different heap arrangement and track algorithm specific variables. For the purposes of our discussion we focus on the current adaptive heap sizing policy for the parallel scavenge garbage collector.

A simplified UML class diagram focusing on the key parts of the ergonomics system is shown in figure 3.2. The following two objects are used heavily in Ergonomics statistics gathering:

The *AdaptiveWeightedAverage* (and padded variants) use exponential weighted averaging to provide a good approximation to the average of the value it is sampling.

An exponential weighted average takes the following form:

$$Avg = (1 - w) * (prev\_avg) + w * new\_sample \tag{3.6}$$

Where w is an integer weighting factor representing a percentage between 0 and 100 percent. Weighted averaging allows for new values to have a larger effect on the average while not disregarding older values. It operates as a form of *smoothing* function. This function has the advantage of being recursive such that it takes constant storage and updates in constant time. This allows for high performance averaging.

A linear least squares fit attempts to fit a straight line through the samples in order to calculate an estimated rate of change. A motivating example would be estimating the major pause time against the size of the old generation; if the gradient is positive the system can expect a reduction of the old generation size to reduce the major pause time.

Various statistics, approximations and estimates are gathered and updated during each garbage collection run. In most cases Ergonomics strives to use recursive algorithms, like that of the weighted average function introduced above to maximise performance.

Using collected approximations and estimates the *AdaptiveSizePolicy* checks the user-defined goals and adjusts the heap parameters by a predefined delta amount which is, in general, a percentage of the current heap size. It is then up to the heap to perform the actual resize. The size policy will only give the required resize amount to avoid violating the single responsibility principle.

Chapter 4

Memory Management as a Control Theory Problem

This chapter provides a basic introduction to the field of control theory. We also show how a possible mapping between the fields of control theory and memory management may be constructed.

4.1 Control Theory

Control theory encompasses a wide array of engineering ideas and mathematical techniques focused on modelling and controlling the behaviour of dynamic systems – systems in which the output varies with the input. In most cases we are trying to manipulate the system input(s) in such a way that the system output(s) matches a given (system specific) reference value. To ensure the reference is accurately met a control system must also be able to cope with errors and external disturbances.

The canonical example of an everyday control system is the central heating system present in many homes. The user sets a temperature goal on their thermostat and the power supplied to the heater(s) is varied in such a way that the house temperature meets the temperature goal as accurately as possible. If a door is opened and the temperature of the home drops, the control system recognises this and acts accordingly. A heating system without any form of control would either be on (at maximum power) or off with no regards to meeting a specified temperature.

A first step in the design of a control system is to form a system model. In many cases the modelling a dynamic system requires the use of differential equations as this allows us to mathematically show how output values change as the inputs are varied. Many physical phenomena are readily described in this form; as an example consider Newton's famous equation $F = m\frac{d^2s}{dt^2}$ which shows how acceleration is related to force. Computer systems unfortunately do not have the privilege of strong physical models to work from. This is undoubtedly the key issue to be addressed when mapping concepts of control theory into an abstract domain.

Although the creation of an *exact* mathematical model is very difficult, the availability of memory cells and processing power allow us to form approximate system models. It is important to note that such approximations require time to allow enough data to be captured before an accurate model can be formed. To reduce this *warm-up* effect we can form an approximate model via mathematical techniques and/or heuristic methods and allow captured data to tune our model – this forms the basis for adaptive control and is out of the scope of this report.

Although many control theory techniques are based on physical systems, if we consider the underlying form

of a control system we can attempt to apply this elsewhere. For our purposes we consider control theory applied to the design of dynamic heap controllers.

We start our journey into the world of control theory by considering a typical control system such as that shown in Figure 4.1. This block diagram encompasses the key elements of (classical) control theory and it is these we intend to generalise and use for our purposes. Brief definitions of these elements are given below.

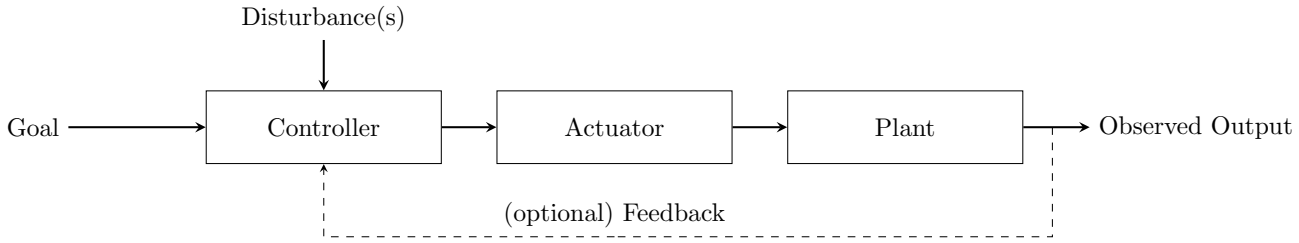


Figure 4.1: Basic Control System

Reference Signal This specifies the *goal* of the system – what the plant output would be in the ideal case. In physical terms this may be a required temperature, speed or rotation etc[1].

Controller Generates a control signal based on the current system properties. These properties may include system disturbances, accumulated errors and feedback from the observed output. The control signal informs the actuator how it needs to react to tend the output closer to the specified goal. It is useful to think of the controller as a decision unit. Note that not all of the system properties need to be used in the decision making process.

For more complex controller designs it is possible to define controllers which provide multiple control signals. This is useful in a system which consists of multiple actuators working together. We generally call this *multiple input, multiple output* control.

Actuator The device which directly affects the plant. A common physical actuator is a motor which affects the rotation a mechanical system.

Plant The plant is the system we are trying to control. If we were designing a controller to control a car's acceleration then the plant would be a car itself. The plant provides an output signal, in this cause the *actual* acceleration of the car as measured by an accelerometer.

In general we wish to to match the output signal as closely as possible to the reference signal.

Disturbance A disturbance can take many forms depending on the system. In a heating system one possible disturbance is ambient temperature which causes an error to the perceived output. A control system will generally try to minimise disturbances by measuring the magnitude external disturbances and using this value to remove disturbances during the decision making process.

4.2 Relationship To Memory Management

Given these basic elements we can start to determine elements of our memory management system which share similar properties. This leads us to the system model shown in Figure 4.2. We can define the components of our new system model as follows:

[1] In practice control theorists deal with electrical representations of these parameters hence *signal*.

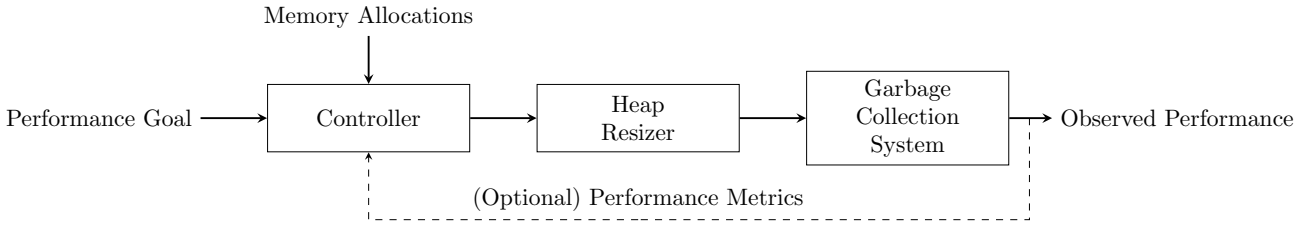


Figure 4.2: Memory Control System

Goals There are a number of reasonable goals for a memory management system. Various examples are:

- Throughput goal: This specifies the amount of time we wish to be spent doing garbage collections per time period. This is the main goal used throughout this report.
- Pause time goal: This specifies the maximum amount of time we wish to spend in any garbage collection. Note this is not the same as throughput as we are unconcerned with the time between garbage collections simply the garbage collection time itself.

Notice that these goals are very similar to those used by the ergonomics system described in Chapter 3.

Heap Resizer The heap resizer acts as the *actuator* for our memory management system. Ignoring special cases such as *system.GC()* – where the program can issue a function call to request an early garbage collection – garbage collections only occur when the system has used all available memory in the allocation space. By varying the amount of memory available to allocate into we can directly control garbage collection performance properties.

Garbage Collection System The choice of garbage collector affects the output properties. As an example consider the choice between a stop-the-world collector and a incremental collector. Within our abstract model we treat the garbage collection system as a black-box. When designing a concrete controller we must take into account the properties of the concrete garbage collection system. This is a limitation of our mapping and makes it very difficult to create a generic controller design. We discuss this limitation further in Chapter 8.

Disturbances Within a garbage collection system the main disturbance is memory allocations. A system that allocates no memory maintains a steady state and no garbage collections are required which allows for maximum performance gains.

4.3 Types of Control

Control theory is a mature field and as such there are a wide array of controller designs available. Common control systems tend to fit loosely into one of two categories:

Open loop control In these control systems the controller does not monitor the system output. These systems respond to a change in either the reference signal or an external disturbance. For example, in a heating system if the user increased the required temperature an open loop system would vary the power to the heaters without regards for the current temperature. These systems are less common than closed loop systems but are still useful. One of our controller designs is based upon this method.

Closed loop control A closed loop system monitors the system output and uses it to calculate an error term. Using this error term the controller can determine the best course of action to counteract the error. In the same heating example as above the controller would check how far from the user specified temperature the system was before varying the power.

4.3.1 Controller Properties

The choice of controller design can affect many system properties. Most notable are as follows:

Response time How quickly the system can respond to a change in reference signal or an external disturbance. This is particularly important for our controller designs and is discussed further in Section 7.3.2 when we investigate at how our system responds to a sudden change in the running applications allocation profile.

Stability An unstable system is one in which the output increases indefinitely as a response to an input signal. For our purposes this is like a controller which maximises memory even when the system does not require the maximum memory to meet its performance goal. Stability analysis is made difficult due to the abstract nature of our system and not discussed further.

4.3.2 Feedforward and Feedback Control

The two controller types most relevant for our purposes are *feedforward control* and *feedback control*. We briefly summarise these types of control and there relationship to our controller designs as follows:

Feedforward Control In a feedforward control system the controller predicts the effect that a system disturbance will have on the plant output and applies a proportional signal to work against it.

Feedforward control has the advantage of simplicity and allows for very quick response since we do not need to wait for an output value to propagate. In our controller designs the program allocation rate is always fed forward into our performance calculations.

Feedback Control A feedback control system relies on plant output information and is an example of a closed loop system. Care must be taken when using feedback control to ensure that the error gain is not too large. If the feedback gain is too large then we risk system stability. To see this more clearly consider a system with some error value e, the system will apply a control signal of $K_e e$ to counteract this error (where K_e) is the system gain. Should K_e be too large then the control signal will introduce another error e_2, and thus the system oscillates indefinitely. If we set the error gain too low then the system response time will increase.

The controllers presented within this report do not place much emphasis on feedback control. The various performance metrics which are monitored are averaged (with a fixed gain of 1) to avoid causing system instability.

We return to control theory again in Chapter 7 where we derive a system model and controller designs for our memory management system.

Chapter 5

Related Work

5.1 Directly Related Work

This work is derived largely from that of White et al.[17] who propose control theory can be used as a principled mechanism for controlling heap resize mechanics. Likewise to our runtime system survey given in Chapter 3 the authors note how current heap sizing mechanisms depend on heuristics rather than a strong mathematical system grounding.

The main difference between their work and that presented in this report is the choice of controller design. They propose to use a proportional, derivative, integral (PID) controller whereas we follow a much simpler controller design while still maintaining a mathematically principled approach. PID control is a form a feedback control and very common within control systems. The controller has the following form where $u(t)$ is the control signal to the actuator and $e(t)$ is the error signal ($required\ output - actual\ output$):

$$u(t) = K_p e(t) + K_i \int_0^t e(t) + K_d \frac{de(t)}{dt} \tag{5.1}$$

The ideas behind each of the PID components can be briefly summarised as follows:

Proportional: How much of the (negated) error is feedback to reduce the error. This determines the speed of response but can also lead to overshoot if the gain (K_p) is set too high.

Integral: Accumulates errors overtime such that when the system reaches a steady state the required output matches the actual output exactly.

Derivative: Ensure a *smooth* response to a rapidly changing error signal. This slows the response to make sure the system does not oscillate or overshoot unnecessarily.

One downside to PID controllers is the necessity to tune the controller to match specific applications. This is done by choosing the values of $K_p\ K_i\ and\ K_d$. There are several approaches to determining these parameters which are out with the scope of this discussion. White et al. perform application benchmarking and manual system tuning, which is a sever limitation of their system.

Their controller implementation is design to work only on full heap collectors which means we cannot take advantage of generational collector properties. Again this is a limitation on their design. It is possible that several

generations could maintain their own controller. In this case the tuning and bookkeeping overheads may become an issue.

To relieve the burden of the programmer needing to manually profile and benchmark their application it is possible to foresee a PID controller system which performs adaptive system control tuning the system at runtime. This however is unsuitable for short running applications which may not have the time required to construct a profile. To avoid running into similar issues, application agnosticism is a key goal of our system design.

5.2 Applications of Control Theory To Computer Science

The application of control theory to other areas of computer science is not a new idea, yet one that may be underutilised. Storm et al.[12] propose an adaptive self tuning controller design applied to manage memory requirements for the DB2 database system.

The authors present a strong argument towards having systems which can dynamically change their parameters at runtime based on the current system load. They place emphasis on the time and expertise a database administrator must have to tune for performance. The work described in this report mirrors this argument. As discussed in Chapter 1 garbage collection tuning is a complex, time expensive and expertise based topic. We likewise develop a control system to remove this burden from the programmer.

The *Self-Tuning Memory Manager* design is significantly more complex than our controller designs. Their controller manages multiple sections of memory allowing dynamic alteration of buffer sizes, locking memory size, and software cache size. A further benefit of their controller that it allows for the control of multiple communicating database instances simultaneously.

To achieve these ambitious goals, Storm et al. use a multiple input, multiple output (MIMO) controller design. In such a design the controller gathers information from many sources. This could take the form of external disturbances, feedback from several sub components and even, as in this case, multiple plants (database instances). This highlights the benefits of a software approach over a traditional *analogue* based control system. We can allow our controllers to perform very complex decision logic which would be extremely difficult with discrete components.

The main point of interest within the paper is the *Self-Tuning* aspect of the design. To build a self-tuning controller a cost/benefit model is first developed. At runtime this cost/benefit model can is tuned by the controller based the *actual* observed effects of the control outputs. This approach works well for long running applications as these have plentiful time to self-tune.

It is possible that several of these idea are applicable to the design of dynamic heap controllers. The management of several memory areas is closely related to that of a generational garbage collector. The possibility of embedded complex decision logic with software is of great benefit, however we must be careful not to slow down the virtual machine unnecessarily. Controller complexity also leads to a significant maintenance issue; although we can remove the burden of system performance tuning, requiring expert control theory knowledge to fix internal system issues becomes a problem.

The idea of controlling multiple system instances running together is very interesting. A similar approach is being investigated by Singer and Cameron[5] who are attempting to apply economic theory (which also requires a cost/benefit model) to managing multiple virtual machine instances in parallel.

Another application of control theory to computer science was made by Gandhi et al.[3] who propose a multiple input, multiple output controller for the Apache web server. The motivating factor is again to reduce the amount of performance tuning necessary by a system administrator.

The authors highlight several of the issues encountered when attempting to map from the well defined electrodynamical control theory into the less defined world of computer science. Given a systems no physical characteristics it is easier to treat the system as a *black box*, a system which we have no knowledge of the internal workings, and model its dynamic characteristics analytically. We form a similar argument in Chapter 4.

The two parameters which they attempt to control are the *MaxClients* and the *KeepAliveTimeout*. A large MaxClients value increases the size of the work threadpool, this increases utilisation at the cost of CPU and memory usage. A large KeepAliveTimeout causes the system to become underutilised while waiting on connection to respond. To allow the control of multiple parameters the authors, like Storm et al. require a MIMO controller.

The MaxClients parameter is similar to the *eden size* parameter within our system. A large heap implies high throughput at the cost of increased memory usage. A small heap causes more time to be spent performing garbage collection but has reduced memory cost.

Core to their design, and that of any control system, is the formation of strong system model. Unlike the work presented within this report, the authors use an empirical analysis to determine the gain parameters of the controller. In Section 7.2 we derive our system model, not by empirical analysis, but by comparison to a physical system. Developing an empirical model may give our controller better performance and this is an area of future work.

Highly relevant to our system is their model evaluation. They show that even a one-step prediction, predicting the value a time $k + 1$ based solely on the value at k time, can in fact give very low values of variance. Our controller designs use a similar one step prediction and these results give us some confidence that this course of action is acceptable.

The controller designs we present in Chapter 7 are all dependant on modelling the program allocation rate as a system *disturbance*. Zhu et al.[19] implement a control system to manage data centres with focus placed on determining whether or not control theory is a good fit for systems research. They model changing workloads as a system disturbance which is very similar to how we model changing workloads (mutator programs) and gives us confidence in this method.

Zhu et al. also highlight limitations of a control theory based approach. The limitation most relevant to our system is:

> "Most classical control problems are formulated as tracking problems. i.e, the controller maintains the outputs at certain reference values."

They proceed in commenting how this tracking problem based approach is not suitable for all systems. Our system has a similar problem. Instead of optimising such that we get maximum performance with as little memory usage as possible, we must set a concrete goal value. This adds in the necessity for user specified goals (or sensible defaults) which leads to programmers needing expert knowledge in order to tune for performance.

We finish this section by considering how our controller implementation fits into the grand scheme of computer science from a control theoretic approach. Patikirikorala et al.[10] present a survey of control theory literature applied to computer science applications.

The authors survey both the types of controller and the performance metric under control. In 37.8% of cases the control system is interested in controlling the system response time. We briefly touch on this in Chapter 7, however our focus is placed on throughput and memory usage. These metrics are the main goal in 6.5% and 2% of cases respectively.

More interesting is the type of control system. In only 1.2% of cases is feedfoward control used on its own and in 10.6% of cases is it coupled with feedback control. We gain good results when using these basic forms of control which leads us to believe they may be underutilised, particularly in systems with a dynamic workload such as virtual machines.

Chapter 6

Implementation Details

This section introduces the internals of the HotSpot runtime system and describes the necessary runtime changes required for the creation of dynamic heap controllers. The motivation behind many of these required changes is from the system model which is derived in Section 7.2

6.1 Garbage Collection in HotSpot

A high level overview of garbage collection techniques was given in Chapter 2. We now relate these techniques to real world garbage collection implementations within the HotSpot Java Virtual Machine.

HotSpot is a very full featured virtual machine equipped with several different forms of garbage collector, each suitable in different situations. For example the *Garbage First* (G1) collector is designed with large memories ($>$ 100GB) in mind. There are also several smaller collectors suitable for everyday desktop and web use.

The *Parallel Scavenge* garbage collector is used as the basis of our proof of concept system. It is important to note that the parallel scavenge collector is designed for collecting the young generation with the user allowed to choose a different form of old collector, for our purposes we focus only on the young generation. We can briefly summarise the main qualities of the parallel scavenge garbage collector as follows:

Stop The World Before garbage collection starts all program threads are fully paused. Note that the *parallel* refers to a parallel garbage collection phase not an incremental collector (one that runs at the same time as the program).

Survivor Spaces The heap is divided into an *eden-space* for allocations and a two semi-spaces known as *to-space* and *from-space*. During each garbage collection, live objects from both eden-space and from-space, will be copied into to-space. Once an object has moved between to-space and from-space more than n (user set or defaults to 7) times it is promoted to the old generation. This process is known as *ageing*. The survivor spaces therefore prevent early promotion. After each garbage collection the spaces are switched (via a fast pointer switch) such that the to-space becomes from-space. This heap layout is shown diagrammatically in Section 6.2.

Copying Objects are always copied from one space to another. This stops the young generation from becoming fragmented.

Fast Allocation Since eden is always fully emptied at each garbage collection we can perform *bump pointer allocation*. With bump pointer allocation we simply move the allocation pointer forward n bytes to allocate

n bytes of memory. This is significantly faster than a free-list based approach where space must be searched for.

Parallel As the name suggests the parallel scavenge garbage collector is designed to exploit the multi-threaded/multi-processor nature of modern devices. The parallelism comes form assigning each garbage collection thread a different set of root elements (from the master root-set) allowing the heap to be traversed in parallel.

6.2 Implementation Details

6.2.1 Heap Layout

The heap layout for the parallel scavenge garbage collector is shown in Figure 6.1. Notice that the heap may be expanded into the *virtual space* on the right by requesting additional (virtual memory) pages from the host operating system. This is key to allowing dynamic heap resizing to happen.

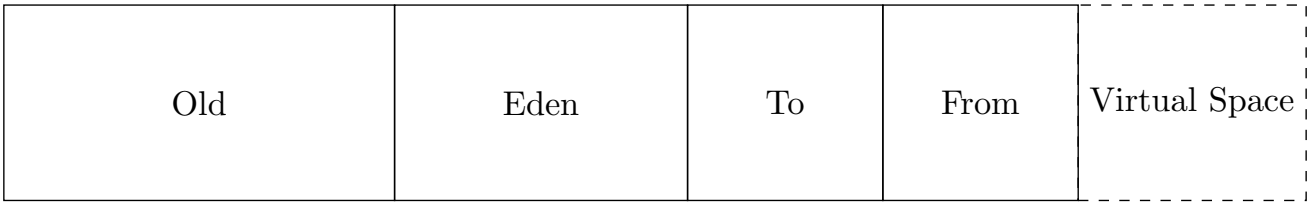


Figure 6.1: Parallel scavenge default heap layout

The nature of the heap layout means that it is much easier to scale the survivor spaces than it is to scale the eden-space. This is acceptable for the Ergonomics heap resizing algorithm which only scales the survivor spaces even at the cost of reducing eden-space. One important thing to note is that we can only scale the survivors when to-space is on the far right because the from-space currently holds data at its low memory addresses. This is a limitation of this scale mechanic.

The system model that will be derived in Section 7.2 focuses largely on the allocation rate of a running program. As such we are interested in controlling the size of the eden-space since this is where all allocations occur. Varying the size of the eden-space determines both *when* a garbage collection will occur and also how much data we can expect to be live at a garbage collection. We adopt a new heap layout to allow easier scaling of the eden space.

Our new heap layout is shown graphically in Figure 6.2. This layout places eden-space to the right of all other generations, right next to the virtual space. Given this, eden-size adjustments are as simple as requesting or freeing operating system pages.

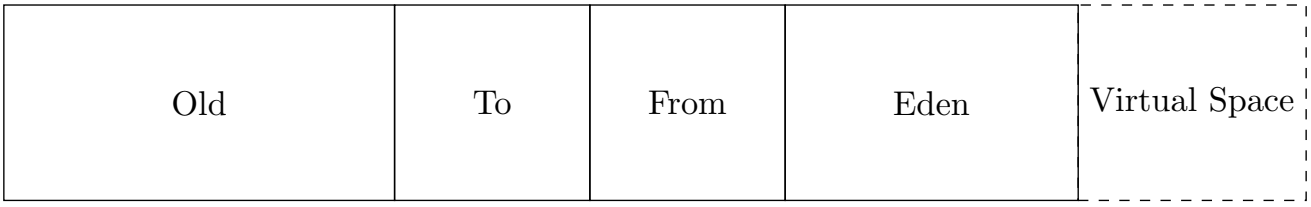


Figure 6.2: Modified Parallel Scavenge Heap Layout

This new heap layout has several benefits. Firstly we can scale the allocation area as much as necessary without affecting the survivor spaces. If eden was still on the left of the survivor spaces then we would be able to scale when to space was directly to the right of eden. This follows from the previous argument of from-space

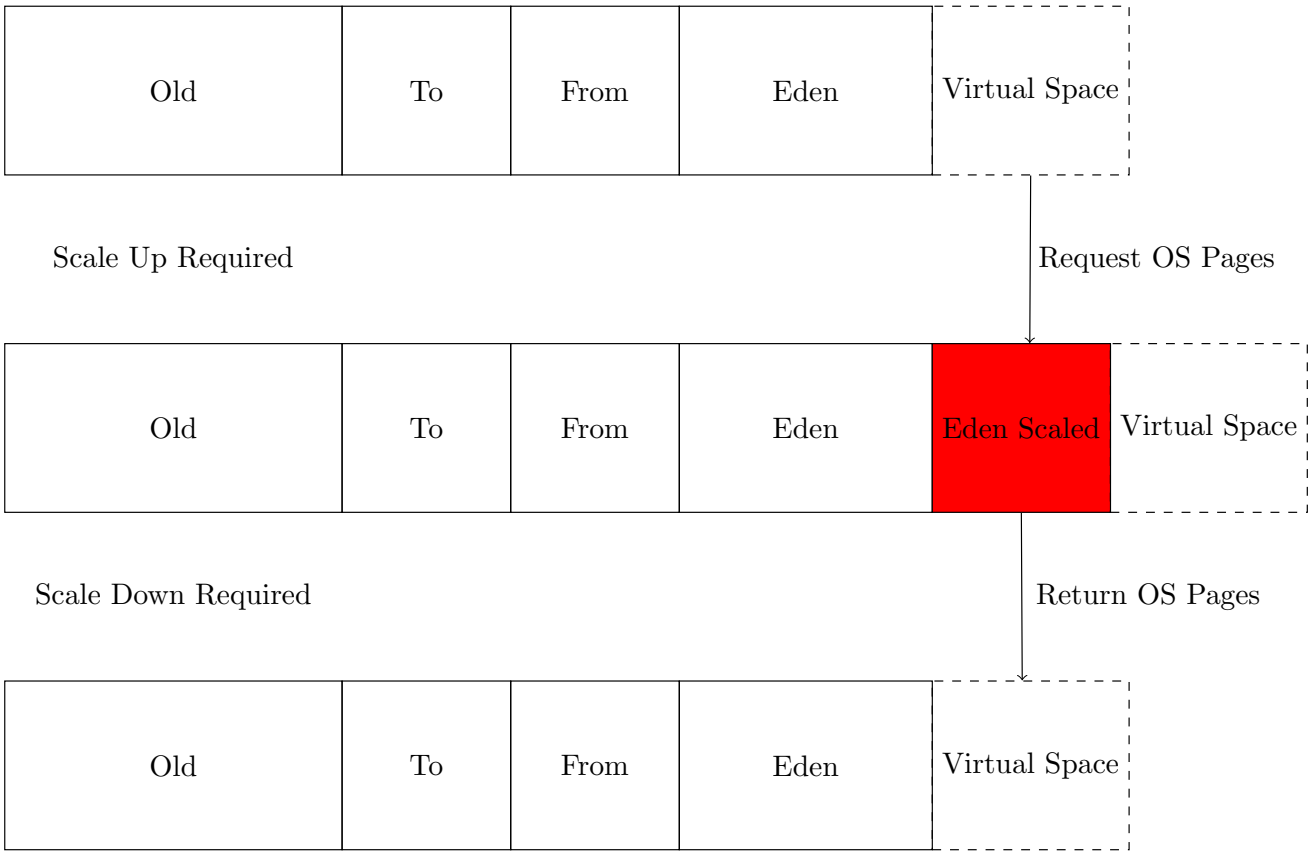


Figure 6.3: Heap Resize Mechanics

containing live data. This is not practical for a controller which is designed to be fast (since we must wait for two garbage collections in the worst case). We have also kept eden space contiguous, this allows us to continue using bump pointer allocation without resorting to a dual allocation area approach where we have an eden at the left and the right of the young generation. Bump pointer allocation is also much simpler to implement.

Using our new heap layout dynamic resizing of eden-space becomes simplified. We show the resize process graphically in Figure 6.3. Notice that since resizing involves requesting operating system pages that we can only scale to page aligned boundaries. This is not an issue as it is unlikely that in practice we need finer grained alignments.

Once the eden-space has been moved to the right hand side of the heap we can hook into the current heap resizing code designed for the Ergonomics system. This code handles the request of operating system pages and also maintains any alignment variants required for the object model. There are also some minor adjustments needed to ensure that the eden-space scales and not the survivor-spaces as ergonomics was originally designed for.

We mention briefly one complication which took a long time to debug. Moving the eden-space to the right hand side of the heap caused countless virtual machine crashes of many different forms. It was difficult to track down the dependence on the eden-space being on the far left of the heap, the code was designed in an object orientated manner and any code segment looking for heap information could simply ask the heap where its semi-spaces were. It was discovered that some code was relying on the garbage collector knowing where the heap's young generation started rather than asking the heap directly. We found this to be bad object orientated design which fully highlights the difficulties involved when working with a large, mature system – particularly in such a hostile environment as a virtual machine.

6.3 Calculating the Allocation Rate

The controller designs presented in Section 7.2 require the ability to track the allocation rate of a running programs and use this to model a system disturbance. In this section we highlight how allocation rate tracking may be achieved.

Two approaches to allocation rate tracking have been explored. The first approach tracks all memory allocations *as they occur*. The other method calculates the allocation rate *during a garbage collection* phase. These approaches offer various advantages and disadvantages which are discussed below.

6.3.1 Per-Allocation Allocation Rate Detection

In the first approach the allocation rate calculation is performed on the critical allocation path, more precisely whenever a thread requires a new thread local allocation buffer (TLAB). The thread local allocation buffers are an allocation method where each thread is allocated some *scratch* space on the heap. This allows for parallel allocations into their own buffers without the overheads of shared memory locking for *every* allocation. Note we still need shared memory protection when allocating the buffers themselves.

The main allocation path only tracks thread local buffer allocations. By measuring the rate at which these allocations happen we get a good prediction of the overall program allocation. This is only a prediction since we do not know if a thread has used all its allowed buffer space.

Because multiple threads may be requesting new buffers allocations simultaneously it is necessary to protect the allocation rate calculation code with a critical region. This ensures that we do not miss an allocation due to a race condition; this will cause erroneous statistics which may severely limit our controller's effectiveness. Normal thread synchronisation is handled by a low level compare-and-swap (CAS) instruction, this is designed to be as fast as possible. The addition of an critical region (via a mutex lock) within the allocation path may have a significant performance overhead as, unlike the compare-and-swap, it is not designed explicitly for fast memory allocation.

To track the allocation rate we perform the following operation:

$$\text{Allocation Rate} = \frac{\text{TLAB Allocation Size}}{\text{Time Since Last Allocation}} \tag{6.1}$$

It is likely that the *time since last allocation* (in seconds) will be small which gives rise to very high allocation rate values. From a practical perspective it is useful to average the calculated rates over n allocations to avoid the allocation rates appearing like noise.

In order to track the allocation rate we require a timer be read (and reset) on each allocation and a (floating-point) division to be performed. Although these operations are both relatively small, due to the number of allocations within a program it is possible this overhead would prove too costly. This analysis has not been performed and is left as future work. See Chapter 8 for more details.

The key advantage to tracking the allocation rate on each buffer allocation is that we can request a garbage collection *before* we run out of allocation space (which is the usual garbage collection trigger). Consider the situation in which the program allocation rate rapidly increases. If we are calculating allocation rates only at garbage collections then we need to wait until the next garbage collection to detect this rapid increase. If we track each allocation then once we notice the rapid rate increase we can request an early garbage collection and

scale the eden-space. This may lead to better overall system performance. No analysis has been performed on this subject and like the overhead analysis above, we leave this as future work.

6.3.2 Garbage Collection Allocation Rate Detection

The second approach calculates the current allocation rate when a garbage collection occurs. This is the final approach used when implementing the controller designs.

Given that all allocations occur into eden space and that eden is always fully evacuated after each garbage collection we can calculate the allocation rate, for this particular garbage collection interval, as follows:

$$\text{Allocation Rate} = \frac{\text{Current Eden Size}}{\text{Time Since Last Collection}} \tag{6.2}$$

The simplicity of this approach is advantageous. We gain an average allocation rate over one garbage collection period using a single division and timer read, this is a significant reduction in the number of cycles used compared to the per allocation method above. When performing the calculations in practice we assume that there was a garbage collection at time $t = 0$, this avoids a one garbage collection period warm up.

The use of *Current Eden Size* in the numerator of Equation 6.3 is an optimisation based on the fact eden space is always fully evacuated at a garbage collection. Should we wish to extend our controllers to multiple generations, as shall be discussed Chapter 8 then we can generalise this equation to the following form:

$$\text{Allocation/Promotion Rate} = \frac{\text{Current Objects Live - Objects Live At Previous GC}}{\text{Time Since Last Collection}} \tag{6.3}$$

This process of finding the allocation at each garbage collection is shown graphically in figure 6.4. Note that this image does not show changing heap sizes, in which the maximum eden line is no longer static and will vary over time.

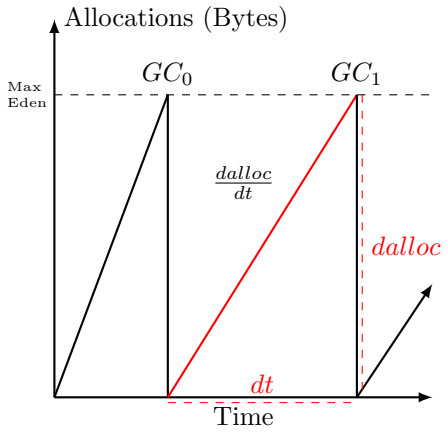


Figure 6.4: Allocation Rate Determination Mechanism

6.4 Software Engineering

We deviate briefly from the details of our controller design and implementation to focus on the software engineering practices involved in a project of this kind.

The controller implementation itself is key to this project and proper care was taken to ensure software engineering practises were applied. The modifications to HotSpot followed the form of other HotSpot modules; this is important should we wish to feed this work back into the larger community. The code was designed to be as simple yet extensible as possible. This is important to allow our results to be verified and to provide a strong platform for future work and modifications by others.

Code is reused where possible, which is particularly useful when dealing with the resizing code since Ergonomics has much of this already. However, in some circumstances code reuse can be difficult. The HotSpot virtual machine is populated with many assertions to aid system debugging (which is notoriously difficult within virtual machines). It was not uncommon to break an assertion when attempting to implement new functionality. For example there are assertions to ensure eden-space is below to-space, this is non-ideal for our purposes. In many of these cases the functionality was replicated with new assertions added as appropriate.

Two main controller designs are discussed in Chapter 7. Both controller designs have been implemented as two independent pieces of code rather than using an interface based approach; in which we allow the different controller designs chosen between at runtime. The issue with having a single, loosely coupled, interface based design is that different controller designs require access to different performance metrics. This requires the addition of many controller specific *hooks* into the abstract model. For example one controller design requires the ability to time how long each garbage collection takes. This requires additional code within the garbage collector which controllers who do not track garbage collection times have no need for. It may be interesting to determine a suitable interface for a generic controller implementation, however as this is primarily a research based project we leave this as future work.

Although not part of the controller implementation itself a dedicated *gclog file* parsing program was created. The log parser was created using the Haskell programming language and a parser combinator library known as *Parsec*. The parsing program is simple and designed to be ran as a command line program which takes a gclog file as an input and outputs a custom comma separated value (csv) formatted file. Given that the parser is written in Haskell it is foreseeable that we could extend this base program to implement further, and more detailed analysis before outputting the csv files. We intend to make this available to the wider community.

Chapter 7

Control Theory for Dynamic Heap Resizing

In this chapter we form a basic mathematical model of our memory management system. Using this model and the implementation ideas from Chapter 6 we design, implement and evaluate four controller designs.

7.1 Experimental Setup

Throughout this chapter we discuss various controller designs and evaluate them in turn. For all the evaluations we follow the same experimental procedure which we discuss here.

All the controllers have been implemented within the Hotspot Java Virtual Machine using the methods highlighted in Chapter 6. Benchmarks were run using a *debug* version of OpenJDK (version 7u) compiled from source. The debug variant of OpenJDK includes extra assertions and runtime checks and does not make use of any compiler optimisations. The lack of optimisations causes a significant increase in the programs expected runtime but allows for additional assertion checks which are invaluable during implementation and controller testing. As future work we intend to perform our analysis again on an optimised version of the OpenJDK.

To make comparisons between our controller implementations and the current heap sizing implementations offered by HotSpot we make use of (a subset of) the *Dacapo* benchmarking suite[2]. This suite provides a wide array of benchmarks with various allocation profiles and is heavily used within the Java memory management community as a standard benchmarking suite. All benchmarks were ran using the *default* work size and were ran for 20 iterations.

All benchmarking was performed on a lightly-loaded machine[1] equipped with 24 *Intel Xeon E5645* processors which use a hyper-threaded 6 core design. These processors run at 2.4GHz with a 12MB cache and there is approximately 100GB of shared main memory available within the system[2]. The machine makes use of the Linux kernel version 2.6.32 as its operating system. Although this machine allows for parallel processing, all benchmarks were ran in singly threaded mode to ensure multiple runs maintained similar allocation profiles.

[1] *Savage* within the dcs network

[2] Our benchmarks use a very small fraction of this computing power and memory.

7.2 Mathematical Model of Our System

As a first step in controller design we wish to formulate a mathematical model for our system. As mentioned in Chapter 4 deriving mathematical models of non-physical systems is difficult due to the lack of well defined physical laws such as Newtonian mechanics. To overcome this issue we draw parallels of our system to that of a physical system and using this we can derive an approximate system model.

Our controllers focus on the response of the garbage collection system when it is influenced by memory allocations (our system disturbance). As we only concern ourselves with a program's allocation rate the main focus is on modelling the eden-space and the characteristics of minor collections. We therefore ignore the response of major garbage collections. Should a major garbage collection occur it is simply treated as a minor garbage collection which happened to take a longer than average time.

To motivate our mathematical model we draw a parallel between our system and a physical system of similar properties. Consider the system in Figure 7.1. The tap fills a fixed size container with water at an average rate of r litres per second. Once the jug is filled fully, we stop the system, *fully* empty the jug and then restart the system. To avoid spillages the tap must be turned off while the container is being emptied. From inspection of the system it can be seen that the time our system is not operating is dependant on the container size, rate of water flow and the time to empty the container.

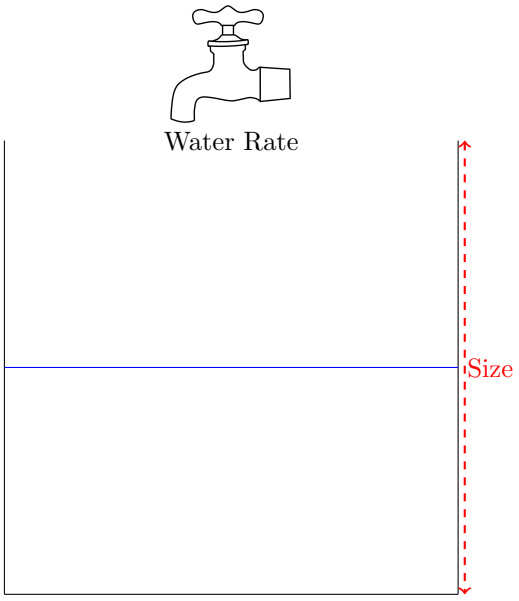


Figure 7.1: Physical Analogy

This is a useful analogy to keep in mind. However we must remember that when our container is emptied we do not throw away the water, we instead keep a non-deterministic amount in another container. This is analogous the live data being kept.

In our memory management system the rate of waterflow is analogous to the program allocation rate, the size of the container is analogous to the size of the eden-space and the time it takes us to empty the container may be viewed as the garbage collection time.

We define the system *throughput* for time interval T as the percentage of time which the program is running within the interval T. To look at this another way, it is the time which we do not spend performing garbage collection in an interval T.

Given this definition and the insights gained from our physical system, we derive a mathematical system

model for garbage collection system throughput as shown below. In this derivation $f(T)$ represents the value of f for a discrete time interval T, for example if $T = 2$ seconds then Time Not In GC(T) represents the time that we were not performing garbage collections for the last two second interval.

$$\text{Time Not In GC}(T) = \text{System Throughput}(T) = T - \text{Time In GC} \tag{7.1}$$

Assume there are N collections in a time interval T

$$\text{Time In GC}(T) = \sum_{i=1}^{N} \text{GC Time}_i \tag{7.2}$$

$$\text{where } N = \frac{\text{Total Allocated(T)}}{\text{Eden Size}} \tag{7.3}$$

Assuming an average time per garbage collection over N collections

$$\text{Time In GC}(T) = \frac{\text{Total Allocated(T)}}{\text{Eden Size}} \times \text{Average GC Time} \tag{7.4}$$

$$\text{Total Allocated}(T) = \frac{\text{Allocation Rate(T)}}{T} \tag{7.5}$$

$$\implies \text{Time In GC}(T) = \frac{\text{Allocation Rate(T)}}{\text{Eden Size} \times T} \times \text{Average GC Time} \tag{7.6}$$

Normalising to one second intervals ($T = 1$)

$$\text{Time In GC Per Second} = \frac{\text{Allocation Rate Per Second}}{\text{Eden Size}} \times \text{Average GC Time} \tag{7.7}$$

$$\implies 1 - \text{System Throughput Per Second} = \frac{\text{Allocation Rate Per Second}}{\text{Eden Size}} \times \text{Average GC Time} \tag{7.8}$$

Relating this back to our control system diagram in Section 4.2 we can deduce the following about the parameters of our model:

System Throughput Per Second This is the *goal* of our control system. We can also express this as a time in garbage collection goal. For example a 95% throughput goal implies 0.05s (50ms) should be spent performing garbage collections each second.

Allocation Rate This is a system *disturbance* that we wish to work against. In general, a higher allocation rate warrants a bigger heap size.

Eden Size The size of eden is the system *actuator*, by changing this we change the system dynamic. It is the only parameter present in our system which the controller can change (assuming goals are static).

Average GC Time This takes the form of *feedback* which feeds our controller information about the current garbage collection properties. This is different from the standard control model where it is much more likely the *actual throughput* would be feedback to the controller in order to construct an error term.

Using this system model we now investigate two main forms of controller design. In the first form uses no system feedback and relies fully on the allocation rate to guide the controller response. The second controller adds feedback by gathering information about the garbage collector properties as it runs. Both controllers give rise to two further forms of controller by utilising the two resize mechanisms presented in Section 6.2.

7.3 Fixed Spacing Controller

The system model presented in Equation 7.8 is constructed of two main components: time between garbage collections and the (average) time it takes to perform a single garbage collection. To derive our first controller design we look at these parameters in detail.

As mentioned in Section 6.1 the parallel scavenge garbage collection algorithm uses heap traversal to find all currently live objects. It is important to note, since we only follow pointers to live objects, the time taken to perform a garbage collection is determined by the amount of live objects and not the amount of garbage. If we assuming the weak generational hypothesis is correct[3] and most objects do indeed die young, then we can approximate the time taken to perform a garbage collection as a small constant value.

Given these assumptions we form the following simplified model assuming GC_t is small and constant:

$$1 - \text{System Throughput Per Second} = GC_t \times \frac{\text{Allocation Rate}}{\text{Eden Size}} \tag{7.9}$$

$$1 - \text{System Throughput Per Second} = GC_t \times \frac{1}{\text{Spacing Goal}} \tag{7.10}$$

Since we assume GC_t is constant, we now specify our throughput goal as a spacing goal instead. We can do this since our model implies that throughout is proportional to spacing.

We now design our first controller which we shall refer to as a *fixed spacing* controller. The fixed spacing controller uses, as parameters, a user specified garbage collection spacing goal and the current program allocation rate (calculated at each garbage collection). The controller can calculate the required eden size to meet its goal as follows:

$$\frac{1}{\text{Spacing Goal}} = \frac{\text{Allocation Rate}}{\text{Eden Size}} \tag{7.11}$$

$$\implies \text{Eden Size} = \text{Allocation Rate} \times \text{Spacing Time} \tag{7.12}$$

Notice how the controller uses the current allocation rate as a prediction of future allocation rate. From experimentation this is an acceptable prediction. A discussion of this limitation may be found in Sections 8.1.

7.3.1 Controller Types

We can further divide this controller idea into two smaller controllers by considering the heap resizing mechanism. There is a fast response and a slow response method which are discussed and evaluated in turn below.

Fast Response

In the case of a *fast response* controller we implement the system model shown in Equation 7.12 and use the calculated value as the absolute scale amount.

[3] There are some circumstances where this is not true.

The heap resize implementation will then scale the heap to the closest alignment boundary *as fast as possible* by requesting or freeing as many pages as it needs.

Notice that if the spacing time is made one second then it is possible to remove an additional multiplication from our controller. It is unlikely that this optimisation will lead to any performance improvements however by fixing the controller spacing at one second we completely remove the need (and the ability) for a user to specify this goal. Manual goal setting is only beneficial in a limited set of cases. As mentioned in Chapter 1 we generally wish a responsive runtime system without the need for any tuning or expert knowledge of the underlying system. The current controller implementation includes manual goal setting for the purposes of evaluation.

Slow Response

We can create a *slow response* version of the spacing controller by, instead of considering eden size calculated by Equation 7.12 as an absolute value, treating the calculated size as a threshold value. Based on the current threshold value we then scale one alignment size (page size) at a time such that we tend slowly towards the allocation rate.

Currently the scale amount is the system page size. It has not been determined whether or not there exist better parameters for deciding on the best scale amount and this is left as future work.

At first it may seem strange to want a slow response controller. There are however benefits to this approach. Consider the situation where the controller detects a sharp allocation rate increase. The fast controller will try to aggressively respond and ask the operating system for as many pages as required. Should the allocation rate drop again then this operating system work, which may include page replacement, is wasted. With the slow response controller we do not place as much strain on the operating system. This is ideal in circumstances where the system is heavily loaded.

The main downside of such an approach is the increased response times when a running program changes its allocation profile. This may be caused, for example, when switching from a user input intensive process to batch process which generally causes large changes in the allocation rate. A slow controller takes longer than the fast controller to react to this change and re-optimise the heap size. For non-batch processes there is the possibility of long *warmup* times (time taken to tend toward the average allocation rate) should the starting heap size is set too low.

7.3.2 Evaluation

Spacing Goal

We begin our controller evaluation by considering how well our model holds in practise. To do this we ran the *tomcat* Dacapo benchmark specifying a spacing goal in increments of 1 second for all spacings between 1 and 10 seconds. The time between garbage collections is found simply by taking the difference between the two closest garbage collection occurrence times[4].

Histograms of the garbage collection spacing times are shown in Figures 7.2 and 7.3. These graphically show the density of garbage collection spacings within our tomcat benchmark run.

While gathering our spacing data the maximum heap size was increased to 1GB to ensure we had enough space to scale when we request high spacing amounts. In the histograms shown, data which is within 0.5 second

[4]This data is available from the *gclog* files

(in either direction) of an integer step is counted as a match for that step.

In most cases the controller meets the specified spacing goal. This gives us confidence in our system model. As the spacing increases less garbage collections are performed. Since we only sample at garbage collections this leads to decreased allocation rate data for the controller to base its scale calculations from and accounts for the anomalies shown in Figures 7.3(c) and 7.3(d).

It is also worth noting the tightly spaced garbage collections (those near 0 seconds) present in all the figures. These appear to be caused by the just in time compiler (JIT) warm up times. During this warm up period we have a bad prediction of future allocation rates since the code has not yet been optimised. Once the optimisations have completed its not uncommon for the allocation rate to double. This is discussed later in this section.

The figures presented above have a large error spacing at 0.5 seconds. To get a clearer idea of the actual controller spacing densities we present a zoomed in version of the two second controller case in Figure 7.4. In this case each box has a width spacing of approximately *0.1 seconds*.

This figure shows significant variation in garbage collection spacing time with the mean and standard deviation values are calculated as *2.15 seconds* and *0.67 seconds* respectively. These values get worse as we tend toward the higher spacing goals. This is, like the anomalies discussed above, caused due to the controller not resizing or gathering data for long periods of time.

From these statistics we can conclude that the controller on average comes close to its garbage collection goal. Unfortunately the high value of standard deviation shows great variation in our garbage collection spacing with lows of *1.5 seconds* being likely. We suspect this is due to the current allocation rate being a bad indicator of future allocation rates. To remedy this in future controller designs we propose keeping track of an average allocation rate over time and using this as a potentially better indicator of future performance. This is discussed further in Section 8.1.

In general, performance is the main goal of our controller. To this end it may be more suitable to overestimate the size of eden such that our mean spacing value is increased. By doing this we lower the likelihood that we miss our spacing goal at the expense of additional memory usage. This offset may be considered an *error* within our control system model. For the rest of the report we consider the system as is; without overestimation being implemented.

Fast Fixed Spacing Controller – Ergonomics Comparison

Now that we have improved confidence in our system model we can compare our controller to both the HotSpot Ergonomics heap resizing algorithm and the no resizing case.

For our comparison the metrics we are most interested in are the system *throughput* and the *average memory usage*. All metrics are averaged over 20 benchmark iterations. To calculate the throughput percentage the following equation is used:

$$\text{Throughput} = \frac{\text{Total Time In GC}}{\text{Total Runtime}} \times 100 \tag{7.13}$$

Which gives the percentage of time we were not doing garbage collection for the complete run. This is used over calculating the throughput at one second intervals as we only get statistics at garbage collection phases, which are non periodic.

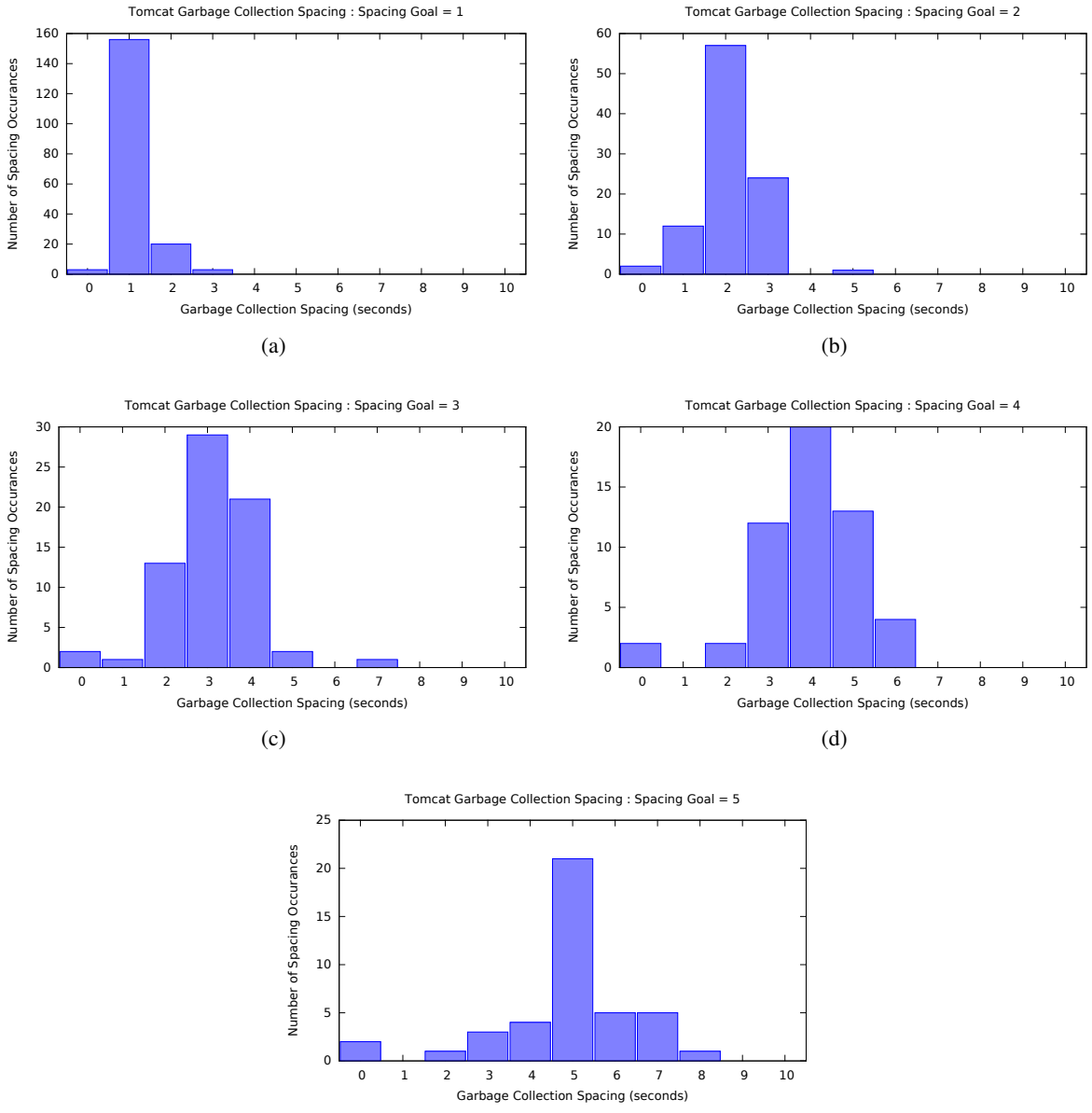


Figure 7.2: Fast fixed spacing controller – GC spacing for tomcat

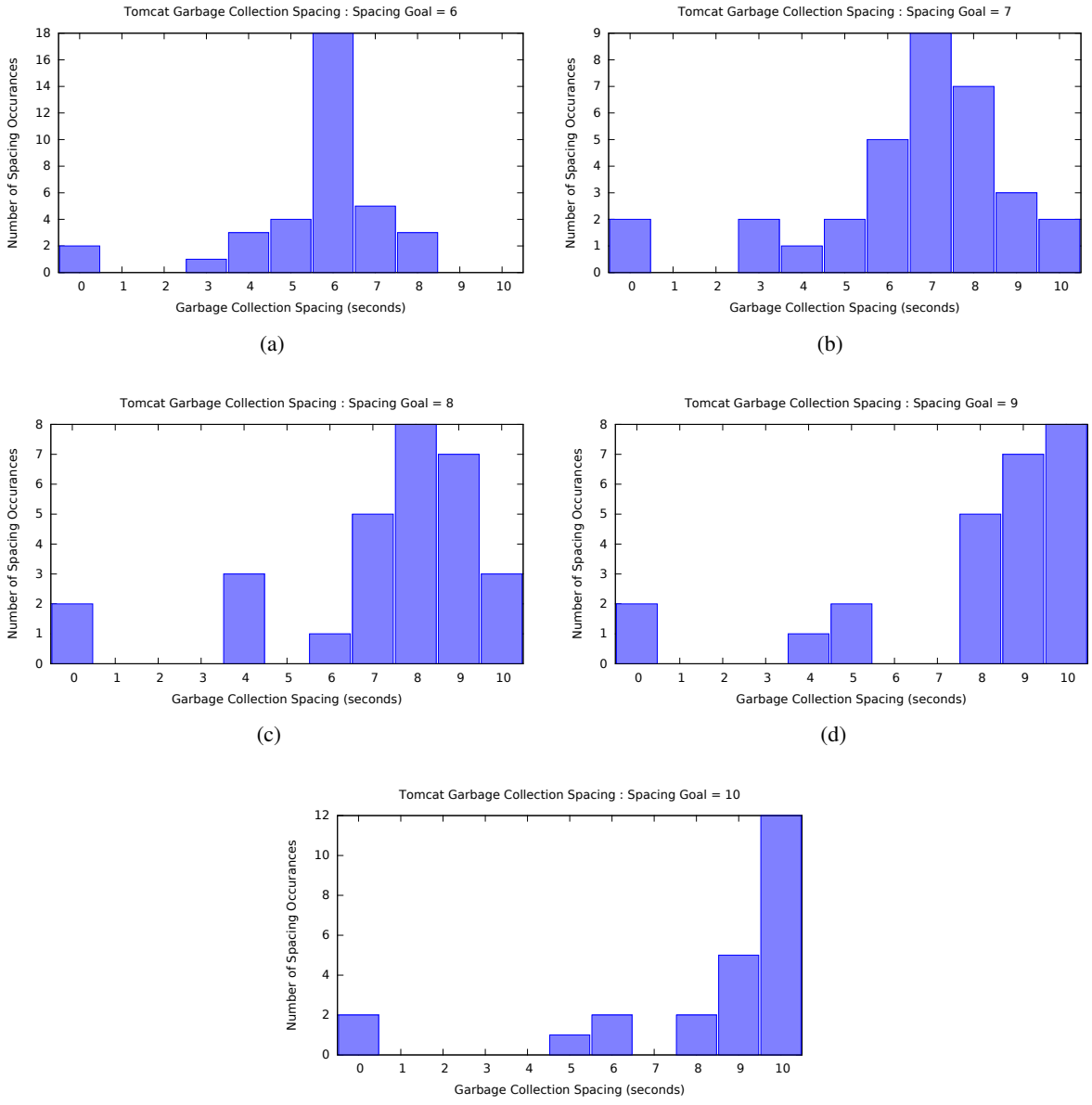


Figure 7.3: Fast fixed spacing controller – GC spacing for tomcat

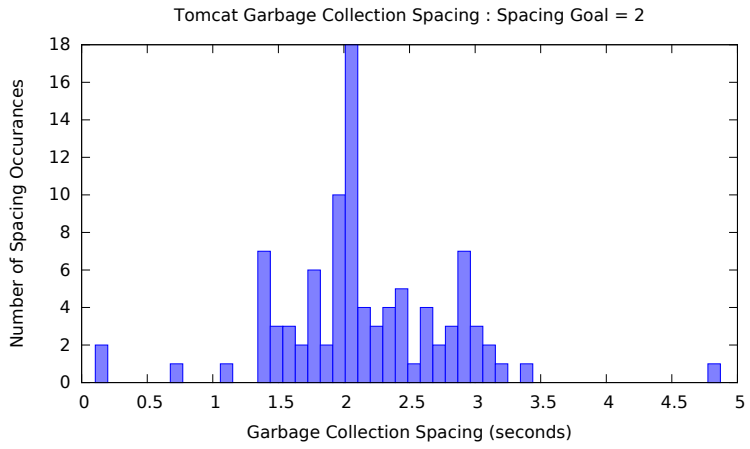
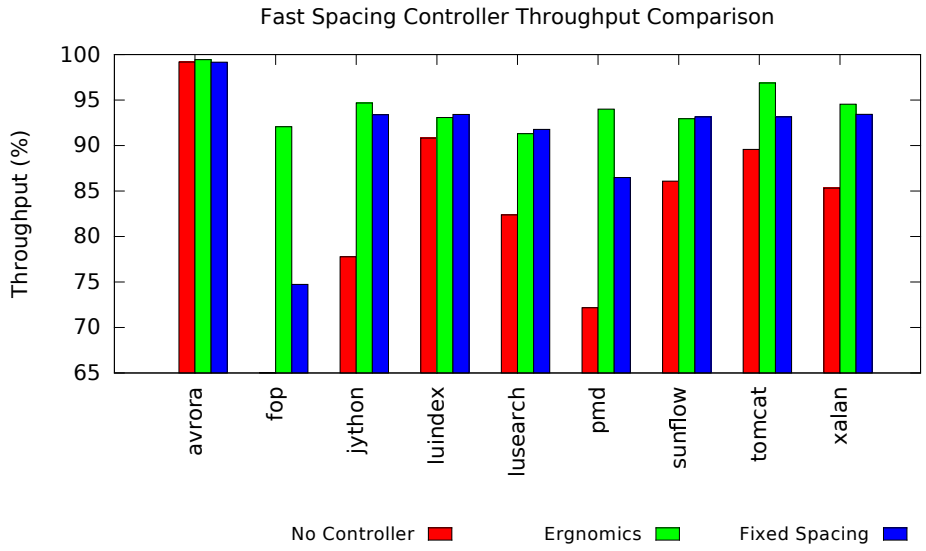
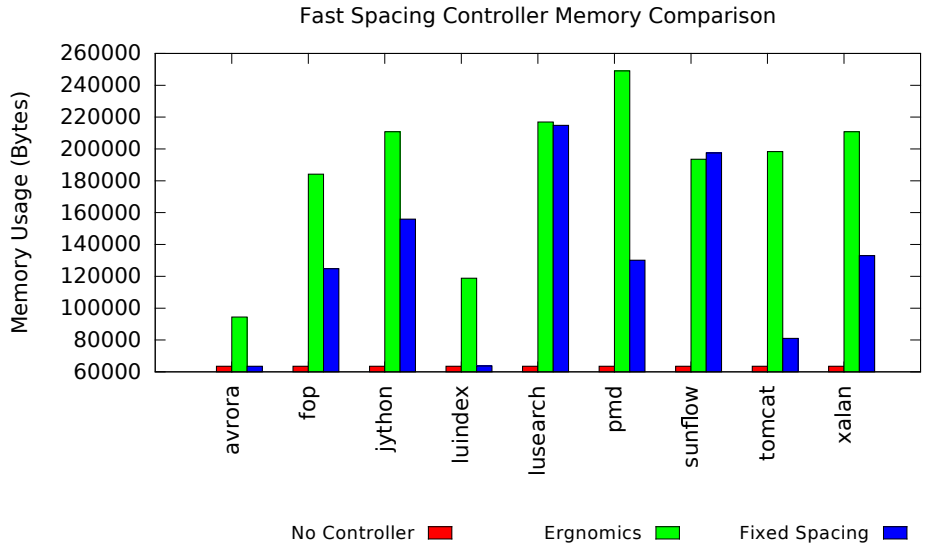


Figure 7.4: Two second fast fixed spacing controller – GC spacing for tomcat

To investigate these metrics we ran several Dacapo benchmarks on the system introduced in Section 7.1. The controller used a spacing goal of one second. Throughput and memory usage results are shown graphically in Figure 7.5. For the full table of results please see Appendix A.



(a)



(b)

Figure 7.5: Fast fixed spacing controller comparison

These results show that Ergonomics consistently gives throughputs of greater than 90%. Our controller performs likewise in 7 out of 9 cases. In all cases the fixed spacing controller always outperforms the no controller case.

What is particularly interesting about the fixed spacing controller is that it provides this high throughput whilst, in many cases, using significantly less memory than Ergonomics. The best case benchmark is *Tomcat* in which we use 1.5 times less memory than Ergonomics whilst only being at a 4% throughput disadvantage.

The two cases in which we do not breach the 90% throughput mark are the *fop* and *pmd* benchmarks. This is due to the allocation rate characteristics of these two programs which are shown in Figure 7.6. For comparison we show the allocation rate data for *xalan* – one of the performant benchmarks.

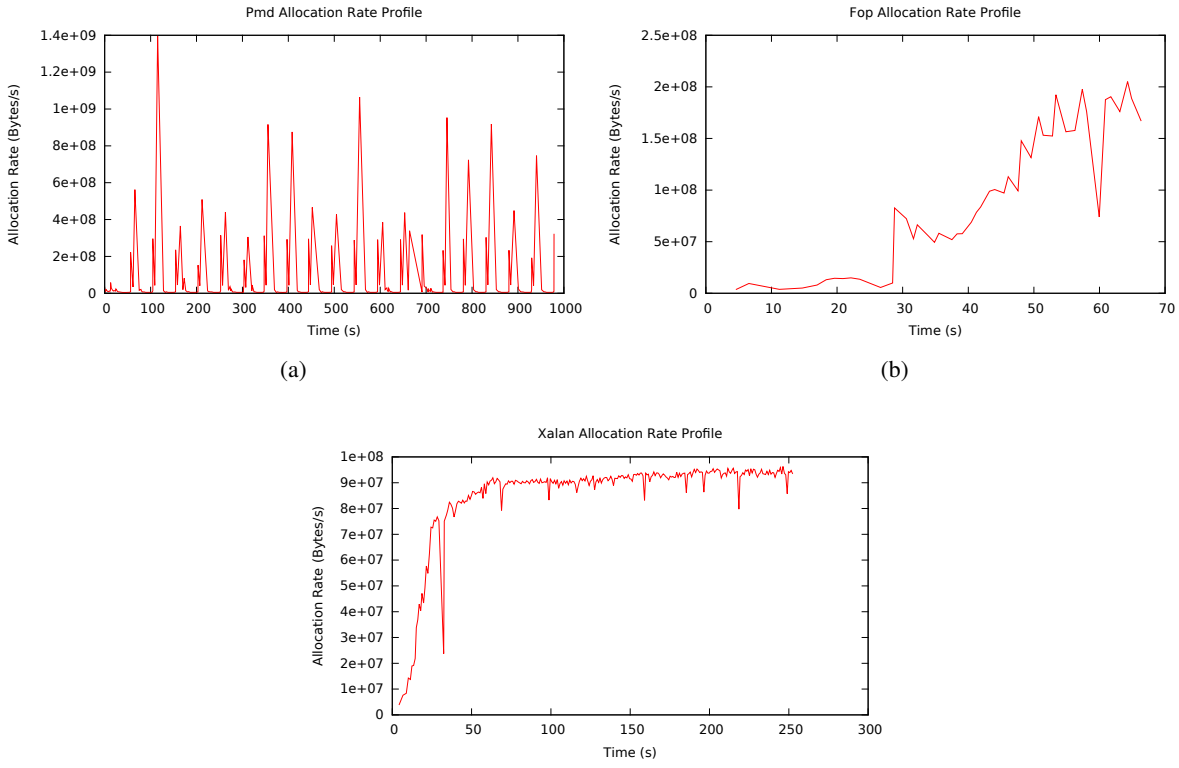


Figure 7.6: Allocation rate profiles for *Pmd*, *Fop* and *Xalan* benchmarks

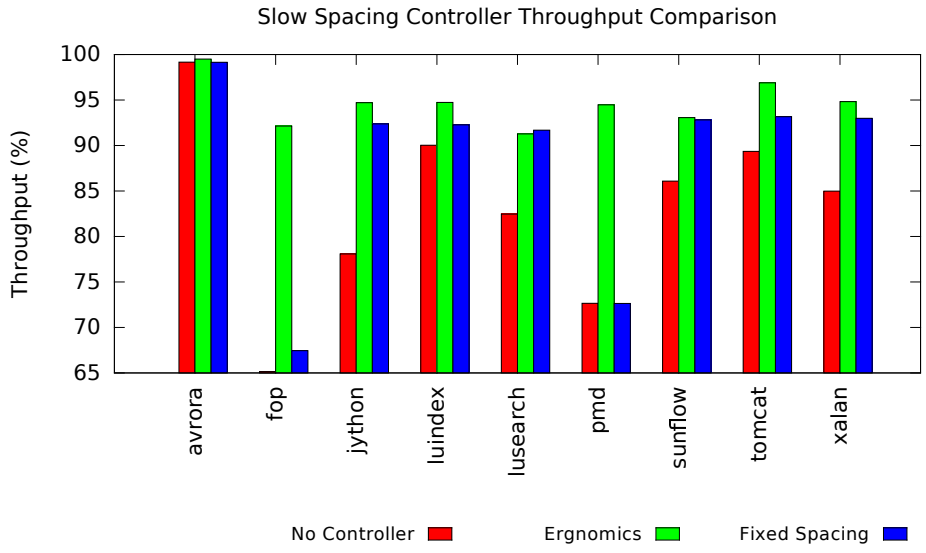
Xalan, like many of the high throughput benchmarks, tends towards an average allocation rate and oscillates around this point. Our controllers are designed to track this rate such that our heap size takes the form $K * Average\ Allocation\ Rate$ and we easily track with little variation. In the case of pmd the allocation rate does not tend towards an average value rather it goes through several burst allocation periods, this becomes difficult to track successfully leading to the reduced performance.

The fop benchmark has a linearly increasing allocation rate and does not tend to an average value for the duration of our run (20 iterations). This shows a limitation of our controller design in that we cannot predict ahead of time what the average allocation rate might be. In general this is not an issue for long running programs but can present an issue for those that are short lived. A discussion of this limitation and possible solutions is presented in Section 8.1.

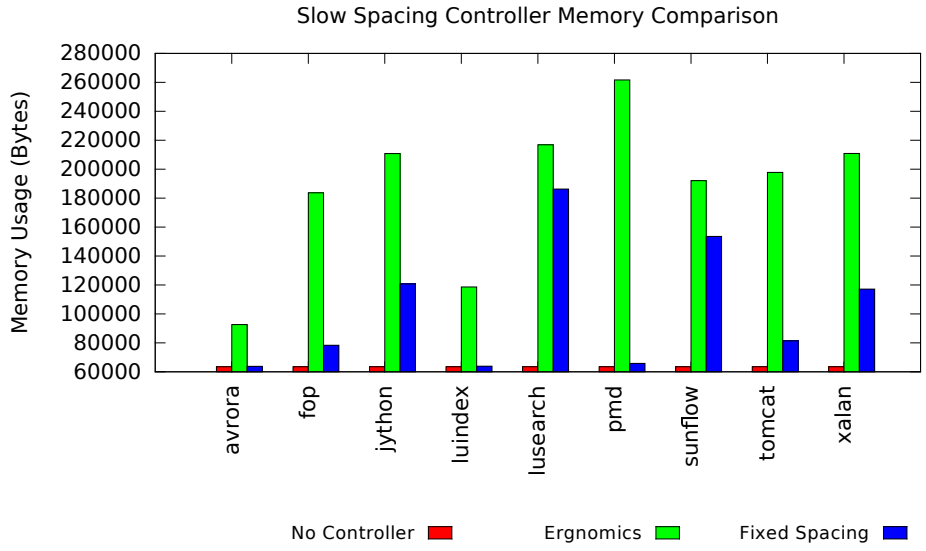
Slow Fixed Spacing Controller – Ergonomics Comparison

The performance characteristics of the slow response, fixed spacing controller are shown in Figure 7.12. The full table of results can be found in Appendix A.

Like the fast response version of the controller, the slow response version also performs well in most cases; achieving high throughputs of over 90% in many cases. For the same reasons as in the fast controller case, the controller does not respond well to the pmd or fop benchmarks. The burst allocation nature of pmd causes very minimal scaling to occur which gives it similar performance to that of the no heap resizing case.



(a)



(b)

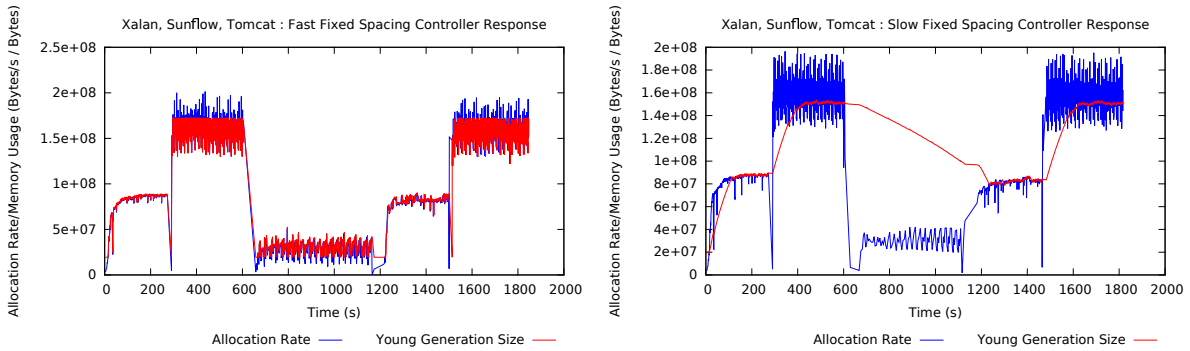
Figure 7.7: Slow fixed spacing controller comparison

Fixed Spacing Controller Response

As is shown for the Xalan benchmark in Figure 7.6, for many of the benchmarks the allocation rate rises until an average level is met, however we argue that this is not always true within production systems. Consider a system which performs *multiple* forms of batch processing for example generating database reports and handing transactions. These applications have differing memory requirements yet will still be ran on the same virtual machine. It is therefore interesting to view how our controller design responds to the system running new batch tasks. In control theory terms we are interested in the controller response to changing *disturbances*.

To test the response time of our controller several Dacapo benchmarks (of differing average allocation rates) were ran back to back on the *same* virtual machine. A run consisting of Xalan, Sunflow and Tomcat [5]is shown in Figure 7.8. We show the (controller calculated) allocation rate in blue and the young generation size in red.

[5]The run performs a full cycle and then ends on the second Tomcat



(a) Fast Controller

(b) Slow Controller

Figure 7.8: Response Characteristics Of The Spacing Controller

The fast controller responds very well to a dynamically changing system. Like in the individual benchmark case this allows for high performance without overusing unneeded memory.

There is however an issue with the fast controller. Consider a heavily loaded system such as a shared cloud system. Scaling quickly requires the virtual space to expand and the operating system to commit extra pages to our process. Fast allocation can force many page replacements at once leading to thrashing in the virtual memory system. The slow controller eliminates this issue by only ever requesting a small amount of pages at once. A second advantage of the slow controller is that we do not return pages only to request them a short while later which is advantageous within batch jobs that follow well defined processing loops such as the one above.

As a final step in our controller response analysis we briefly analyse the response characteristics of the Ergonomics system. The response of Ergonomics to the same back to back benchmarks as above is shown in Figure 7.9. Ergonomics does not provide allocation rate data therefore, for comparison, we have included the allocation rate data from the fast spacing controller runs; this affects the time offsets but we can still see the general trend. This time offset issue is discussed further in Section 7.6.

This figure shows Ergonomics moving very quickly towards maximum memory usage and staying there, varying only very slightly when new batch processes start. This appears to be wasteful of memory particularly when you monitor the underlying allocation rate characteristics. Ergonomics hardly adjusts for the running program at all. Increasing the heap size to the maximum early does however provide high throughput at the cost of memory.

Fixed Spacing Controller – Conclusions

Although the implementation of a fixed spacing controller is very simple, in many cases it has similar performance and reduced memory usage compared to the to the HotSpot Ergonomics system.

We have considered two rescaling mechanisms both of which give differing characteristics which are particularly useful depending on system characteristics, in particular virtual memory and system load. In future we envisage a controller implementation which allows dynamic switching between these two scaling methods. This is discussed further in Chapter 8.

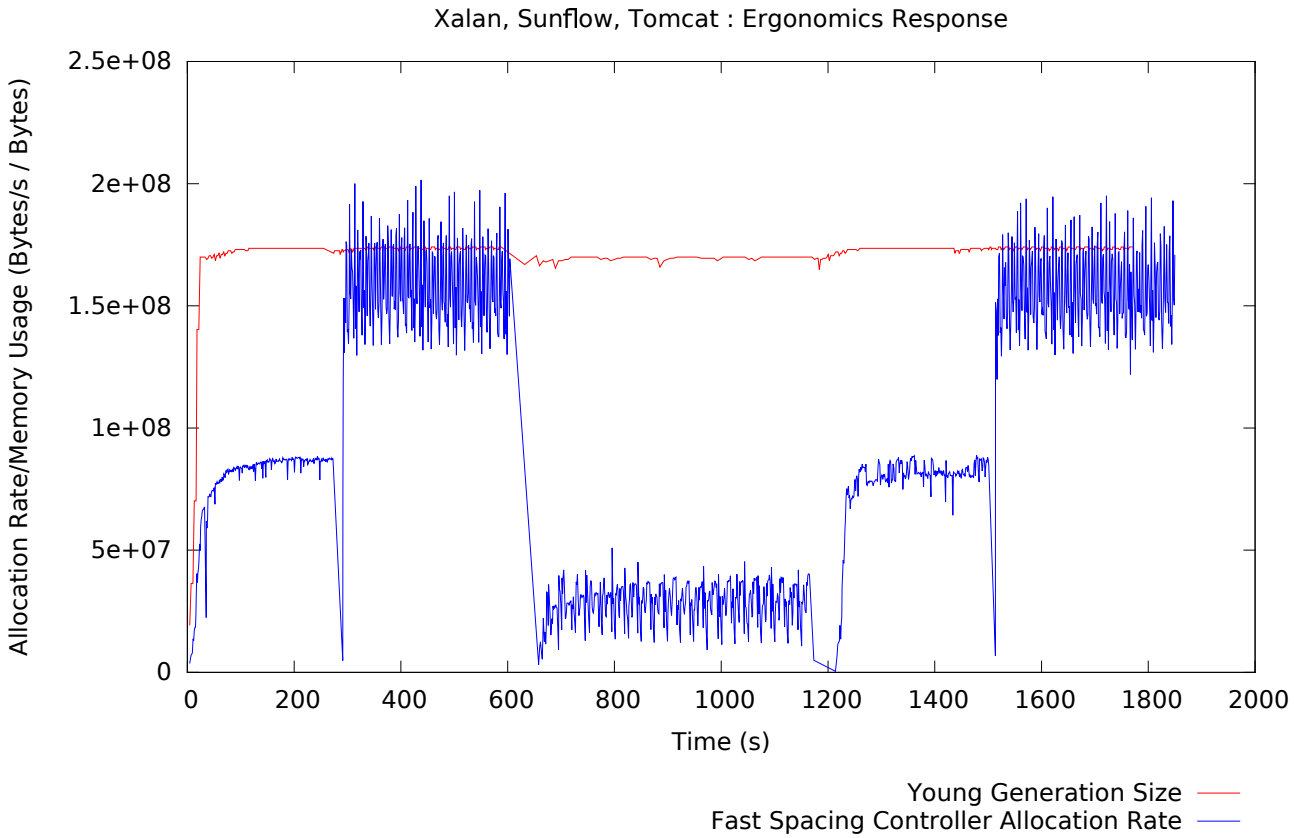


Figure 7.9: Response characteristics of Ergonomics

7.4 Throughput Controller

The controller presented in section 7.3 was designed on the assumption that the time taken to perform a garbage collection was small and constant. For a more accurate controller design we can use both the allocation rate and average garbage collection time as control parameters. Given this, we can allow a user to specify a *throughput goal* and resize the heap such that this goal is met. From the mathematical model we can derive the equation for the eden size as follows:

$$\text{Eden Size} = \frac{\text{Average GC Time} \times \text{Allocation Rate}}{\text{Time In GC Per Second Goal}} \tag{7.14}$$

This controller introduces an element of *feedback* into the system. In this case instead of monitoring the throughput value (our system's *output*) we instead monitor the average garbage collection time. As we know the throughput value depends on the average garbage collection time then this is acceptable as feedback.

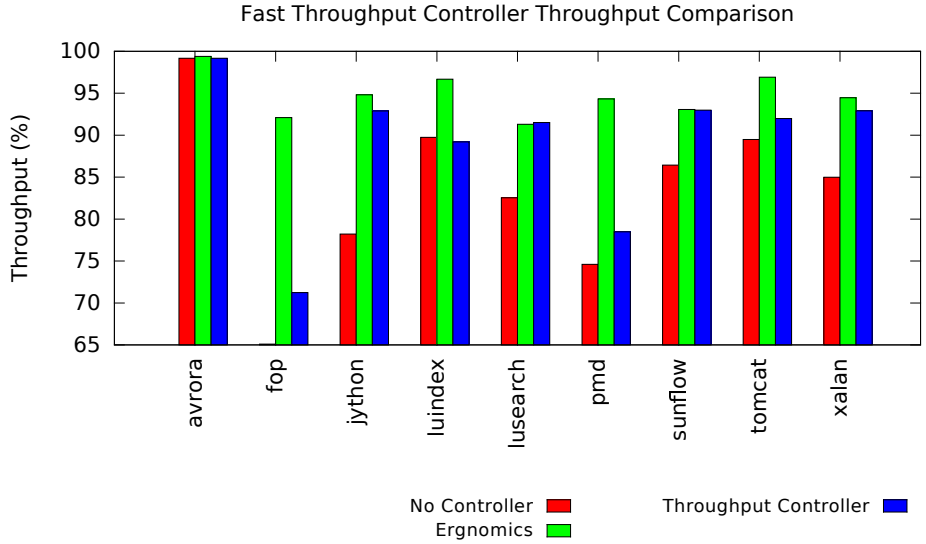
To track the average garbage collection time taken we use the *AdaptiveWeightedAverage* class provided by HotSpot (for use within Ergonomics). This class provides an exponential averaging function as discussed in section 3.3.1. This class is advantageous due to its constant time and memory requirements. This is in contrast to a windowing average function which requires W memory elements and linear computation time.

Like with the fixed spacing controller we can develop both fast and slow response versions of the throughput controller by simply varying its scaling policy.

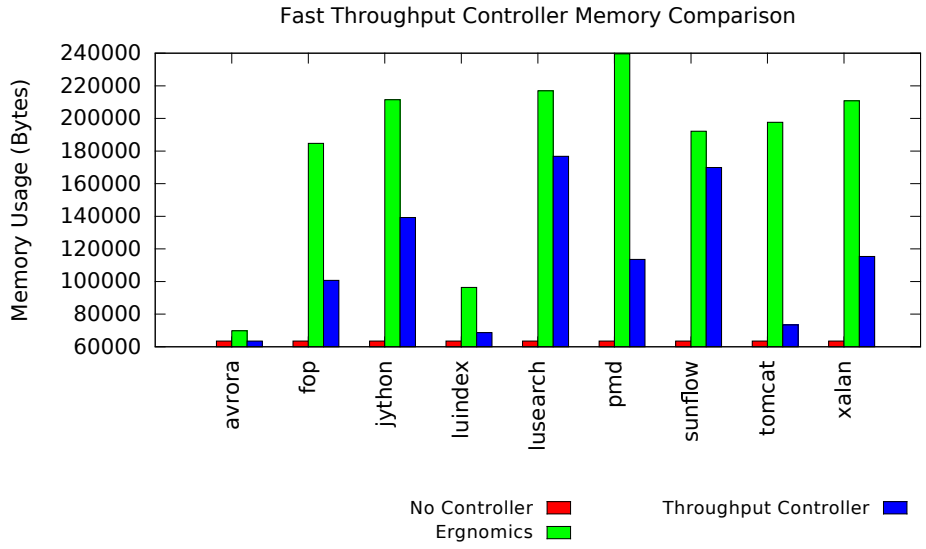
7.4.1 Evaluation

Fast Throughput Controller – Ergonomics Comparison

The results of the fast throughput controller are shown graphically in Figure 7.10. In this case the controller is given a time in garbage collection goal of 0.05 seconds, thus we are aiming for a 95% throughput goal. A table of results is also given in Appendix A.



(a)



(b)

Figure 7.10: Fast throughput controller comparison

The controller never meets its 95% throughput goal however, like the fixed spacing case, in 7 out of 10 cases breaks the 90% throughput mark. There are several possible reasons why the controller does not meet its goal. Firstly it is unknown whether allocation rate at a garbage collection accurately predicts the future allocation rates. One way to reduce this issue is to use weighted averaging for the allocation rates, in the hope that we can determine the average rate more closely. Secondly the controller depends upon tracking the time taken to

perform a garbage collection and feeding this back. For short running programs this weighted average may not be accurate which leads to errors in the output. To alleviate this problem it may be necessary to overestimate garbage collection times until a point in which the system settles to a well defined average value (should this point ever occur). With these improvements in place we may well meet the required 95% throughput goal at the cost of overusing memory that may not be strictly necessary.

Luindex is a particularly interesting case in that its throughput is similar to that of the no resize case; differing greatly from the fast spacing controller which gives 93% throughput. We gain an understanding of this by looking at the GC time taken statistics as shown in Figure 7.11.

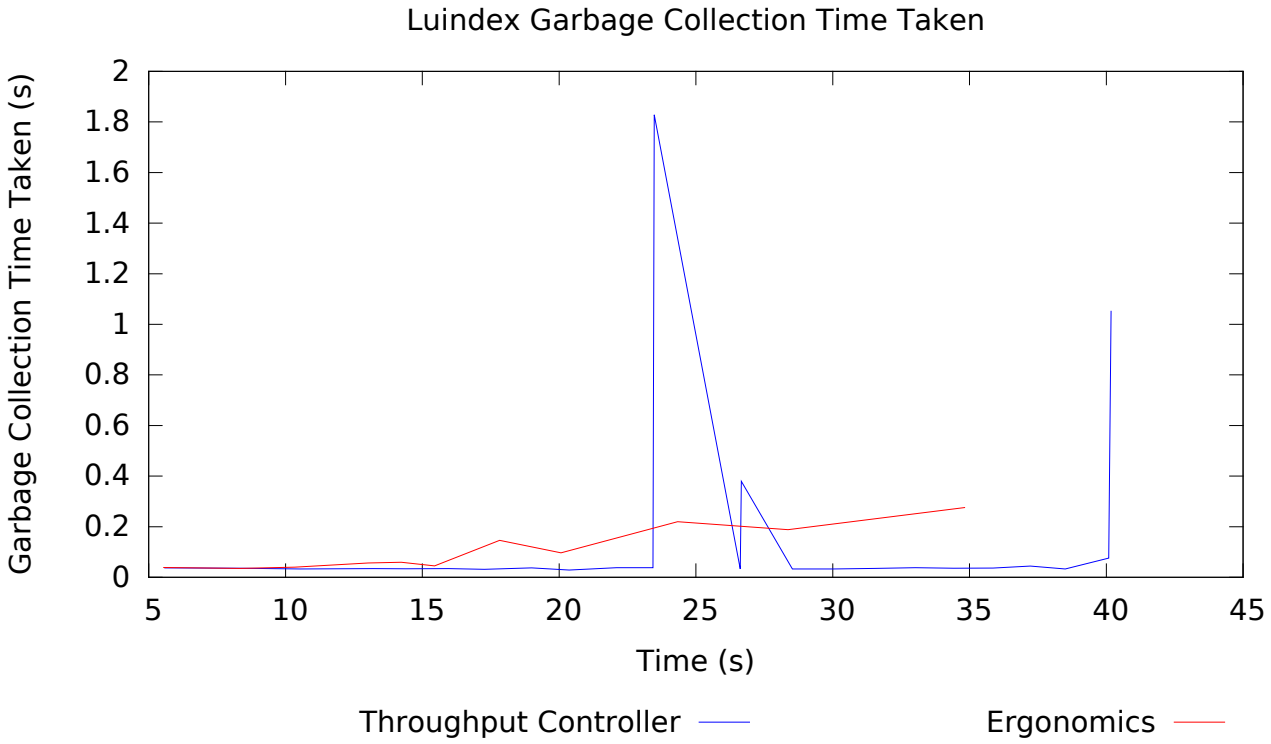


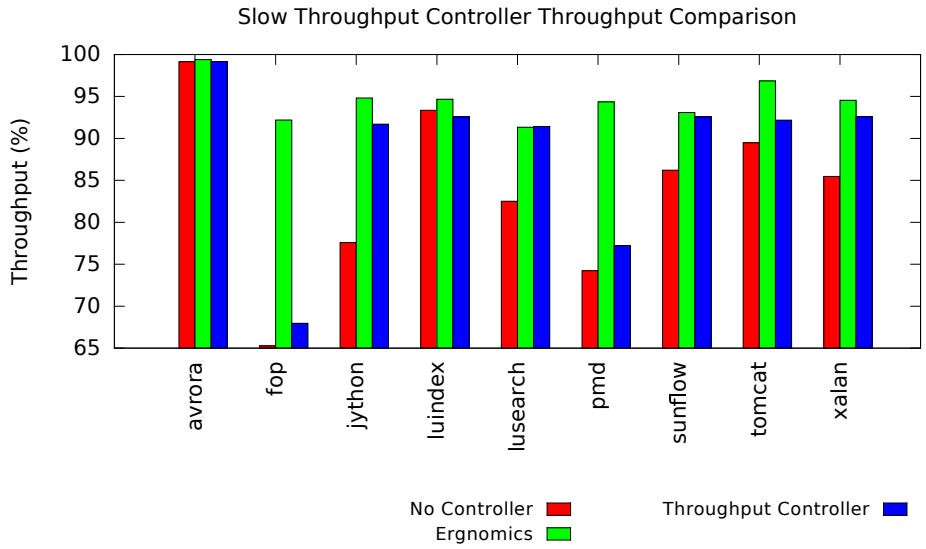
Figure 7.11: Time taken par GC for Luindex

From this graph we can gather that a full garbage collection occurs at around $time = 10$ seconds and again at around $time = 40$ seconds. This shows a common problem of the heap size being too low. With a low heap size objects may be promoted to the old generation early (they are not *really* long lived objects) only to die before the next full collection. We can see that Ergonomics avoids this issue due to its large heap stopping early promotion. The solution to this issue is tracking the amount of live data which is being promoted and using this as an additional controller parameter. This issue is discussed further in Chapter 8.

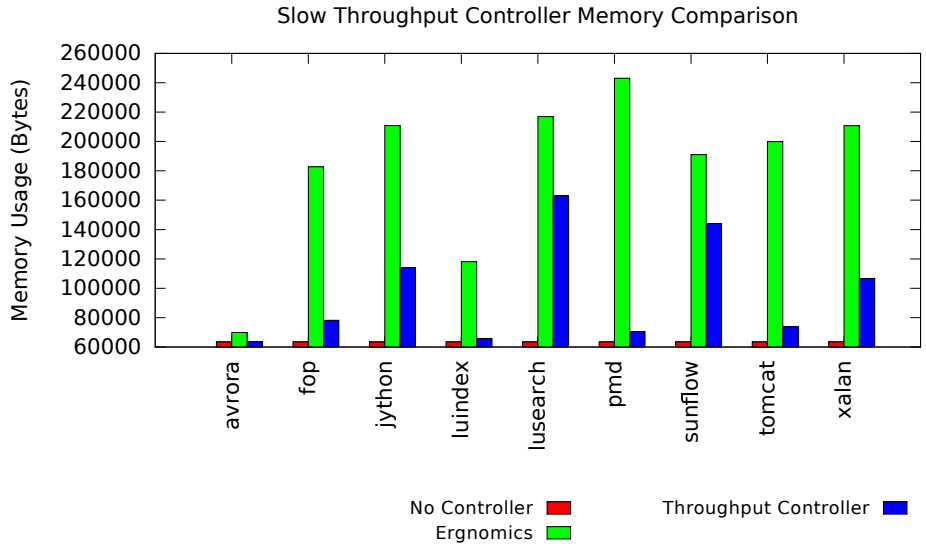
Slow Throughput Controller – Ergonomics Comparison

The controller comparison results for the slow throughput controller are shown in Figure 7.12.

Likewise with the other controllers we achieve good overall performance with fop and pmd being the benchmarks we struggle most to gain high performance from. The Luindex full garbage collection issue is still present within the slow controller due to the lower memory requested.



(a)



(b)

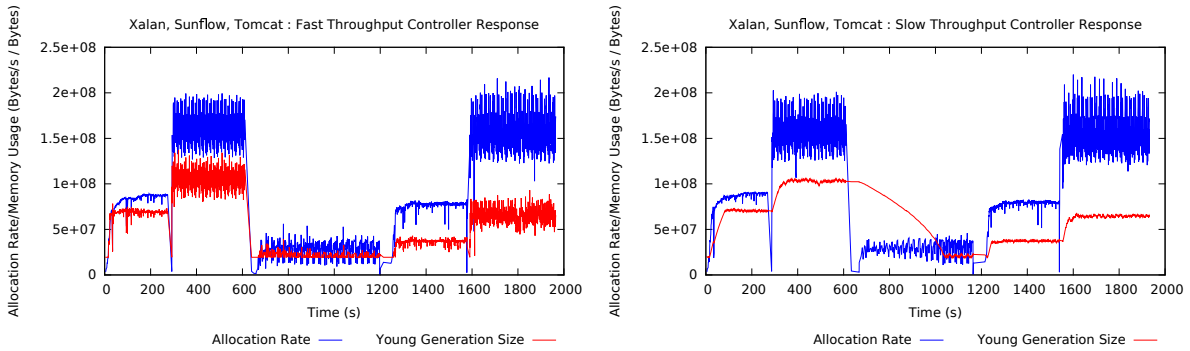
Figure 7.12: Slow throughput controller comparison

Throughput Controller Response

Like with the spacing controller we are also interested in the response characteristics of a throughput controller to changing plants. There is a potential area of difficulty in using a weighted average as there is a period of time in which we are estimating the average garbage collection time of a particular batch job output on the garbage collection times of a previous job. From experimentation we see that is not a large issue mainly due to the average being 80% weighted in favour of new samples.

Figure 7.13 shows the response of the throughput controller to a dynamically changing system.

We observe that unlike the fixed spacing controller where the allocation rate is tracked as closely as possible the throughput controller can reduce the overall memory requirements (if appropriate) due to the knowledge of average garbage collection times. The slow response controller follows in a similar fashion to that of the slow



(a) Fast Controller

(b) Slow Controller

Figure 7.13: Response Characteristics Of The Throughput Controller

response case of the fixed spacing controller.

Throughput Controller – Conclusions

The throughput controller is slightly less performant in terms of throughput than the fixed spacing controller introduced in Section 7.3. However, in many benchmark runs we still break the 90% throughput mark. We have not met the requested 95% throughput goal which implies there is either an error in our model or that the average garbage collection time has significant error. To further improve the controller we could measure the *actual* throughput and use this as an additional feedback term.

The addition of feedback to track the average time for a garbage collection allows the throughput controller to achieve high throughput with reduced memory usage which is highly beneficial for constrained or overloaded systems.

7.5 Controller Design Comparison

Now that we have looked at the performance characteristics of our four controller designs it is interesting to perform a comparison between them. In particular this allows us to determine if the feedback used in the throughput case has any noticeable performance improvements.

Table 7.1 presents this comparison between the throughput and spacing controllers.

To better visualise this data we have come up with a suitable performance metric: *Space Rental : Throughput ratio*.

Space rental corresponds to the area under the curve of young generation memory usage over time. This is measured in bytes per second and is an indicator of the runtime memory efficiency during a run. As we are concerned with the area under a curve, we may calculate the space rental numerically using the *Trapezoidal rule* for numerical integration and the heap size data gathered from the garbage collection logs.

This metric allows us to compare controllers with higher memory requirements, yet better performance, with those of lower memory requirements and reduced performance. In general the *lower* the value of space rental : throughput implies a higher efficiency in terms of *either* memory usage or throughput.

Metric	Benchmark	Spacing Fast	Spacing Slow	Throughput Fast	Throughput Slow
Average Runtime (ms)	avrora	10698.20	10860.95	10934.35	10740.10
	fop	3089.15	3357.35	3210.65	3358.05
	jython	24164.35	25442.75	25033.25	25056.75
	luindex	1896.75	1870.65	1870.30	1890.50
	lusearch	32468.30	32624.30	32599.25	32942.00
	pmd	44290.25	52404.64	48680.90	49625.70
	sunflow	15578.70	15564.90	15653.20	15798.10
	tomcat	19923.05	19603.00	20478.90	20439.55
	xalan	11921.30	12041.05	12135.80	12396.90
Average Throughput (%)	avrora	99.16	99.15	99.18	99.16
	fop	74.74	67.45	71.24	67.95
	jython	93.38	92.38	92.89	91.70
	luindex	93.42	92.28	89.21	92.56
	lusearch	91.78	91.68	91.51	91.42
	pmd	86.48	72.64	78.48	77.21
	sunflow	93.15	92.82	92.99	92.56
	tomcat	93.15	93.17	91.99	92.18
	xalan	93.43	92.99	92.91	92.58
Average Memory Usage (Bytes)	avrora	63488.00	63744.00	63488.00	63744.00
	fop	124753.70	78373.33	100693.33	78202.67
	jython	155911.76	120719.78	139269.85	114161.07
	luindex	63707.43	63890.29	68672.00	65772.31
	lusearch	214865.81	186259.25	176758.25	162972.98
	pmd	130115.37	65807.96	113602.23	70575.83
	sunflow	197531.02	153581.13	169882.76	144018.61
	tomcat	81045.39	81387.46	73400.23	73720.00
	xalan	132959.42	117131.90	115353.97	106674.23

Table 7.1: Controller Comparison

This metric is a novel contribution that is equally as applicable to the controller comparison with Ergonomics but has been left out for simplicity. We hope this metric will be adopted in the future as a method for comparing programs with differing efficiency characteristics, in terms of both memory usage and throughput, on equal footing.

It is interesting to notice that, in general, the slow response controllers give a better memory, performance trade-off than their fast counterparts. Using both the space rental per percentage throughput and the information presented in the table we make the following conclusions:

- The fixed spacing controller achieved high performance without requiring the use of performance metrics. This is recommended when memory usage is not an issue.
- Throughput controllers, due to the tracking of the garbage collection time, use less memory than their fixed spacing counterparts in many cases. This form of control is recommended when working within a memory constrained system.

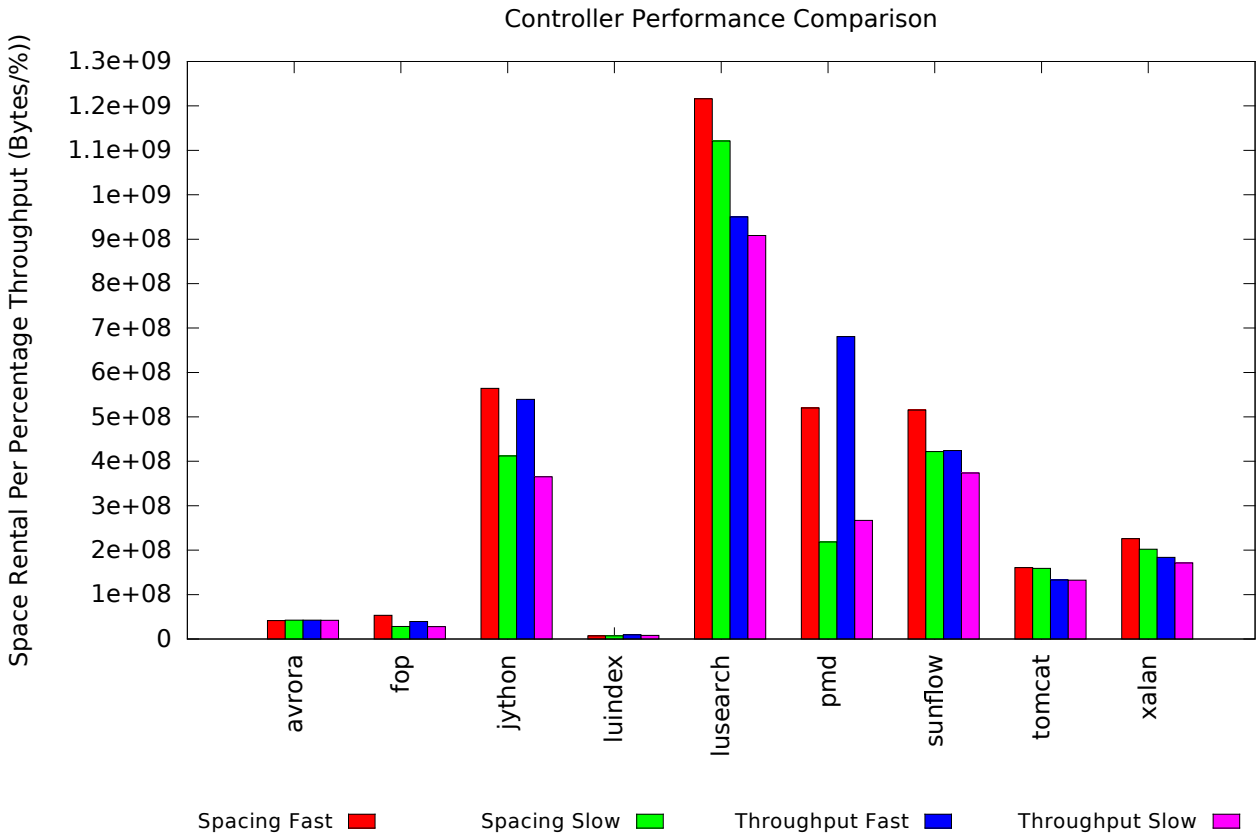


Figure 7.14: Comparison of the various controller designs

7.5.1 Conclusions

All of the controller designs presented have different performance characteristics. In most cases however, they can achieve similar throughput measures to that of Ergonomics commonly breaking the 90% throughput mark. The controllers are conservative with memory and in many cases achieve significant memory reductions when compared to Ergonomics.

When attempting to control a system under the influence of changing allocation profiles, the controllers quickly readjust to maintain high performance.

Now that we have an idea of the performance properties of the controller designs we briefly turn our attention to the non-performance characteristics of the controllers.

7.5.2 Non-Performance Characteristics

The controllers we have designed strive for simplicity. A simple, yet principled, controller design has the following three key benefits:

- A simple controller design allows easier proof of correctness. It is important we know that we have implemented the mathematical model correctly.
- Simple designs can be easily ported into other garbage collection systems and even dynamically *hot swapped* at run time, if the environment allows for this. Soman et al.[11] show a similar method for

dynamically changing the garbage collector implementation at run time. As future work it would be interesting to investigate the cross over of the two systems and perhaps implement both systems in parallel such that the virtual machine chooses both a garbage collector *and* a sizing policy dynamically.

- A small implementation size reduces the resource requirements of the virtual machine itself. This allows the design to be implemented on constrained devices such as embedded systems.

A very simple code complexity comparison between our controllers and that of HotSpot's Ergonomics system is detailed in Table 7.2. We claim simplicity based on analysis of source code metrics including lines of code, class dependency counts and a simple structural analysis. For simplicity we include assertion statements code that would not be compiled within an optimised build; classes which require inheritance are combined as if they were a single class and we focus only on the main controller classes and count any helper classes as a dependency.

The *Flow Control/Conditional Statements* metric consists of a count of all *if, for and while* statements present within the controller designs. This metric, although very basic, gives us an insight to the structural complexity of our controllers. More advanced structural analysis such as cyclomatic complexity[7] has been considered as future work.

Controller	Lines of Code	Class Dependencies	Flow Control/Conditional Statements
Ergonomics	1276	6	74
Fixed Spacing Controller	52	1	4
Throughput Controller	72	2	5

Table 7.2: Controller code complexity comparison

7.6 Interpretation Of Results

Before wrapping up this chapter we briefly summarise our result gathering process and potential sources of error within our results.

Data gathering for a particular controller was done within a single benchmark run. 20 iterations were chosen as enough time for the just in time compiler to perform optimisations and then to have multiple benchmark iterations where the system is relatively stable. Once the system becomes stable the system tends towards an average runtime and it is these averages that are used as the result.

This approach was chosen due to the difficulties in performing multiple controller runs. We have very limited control over the just in time compiler; which may optimise differently between runs. Our statistics are also gathered during the controller run/garbage collection phase. Due to the non-periodic nature of the garbage collection phases it is difficult to compare data across runs without using some form on interpolation.

Another side effect of non periodic statistics gathering is in the throughput calculations. Although we have defined throughput as *time not spent performing garbage collection per second*, since we do not have one second interval statistics we have instead chosen to use average throughput over the entire run. This process was described in Equation 7.13.

Chapter 8

System Limitations and Future Work

The evaluation of our controllers given in Chapter 7 has highlighted various limitations in our current controller design. In this chapter we present solutions to our current controller limitations and lay the foundations for future work in this area.

8.1 Discussion of Limitations

The single biggest limitation of our controller is the dependence on the running program's current allocation rate. This begs the question, how good is the current allocation rate at approximating future allocation rates? In many cases our benchmarks show that programs tend toward an average allocation rate. However, by only considering the latest allocation rate values our controller can be very oscillatory around this point. This oscillatory behaviour causes many pages to be requested and freed requiring significant operating system calls. To reduce the effects of the oscillatory behaviour we propose maintaining a running average of the previous n allocation rates. As an alternative to a windowed running average it would be possible to use a weighted average for constant time and memory performance while still allowing for a smoothing effect. The reason this was not done in our controller is due to the high performance we can currently achieve with just a single look back, this was however on a lightly loaded system.

The Pmd benchmark also highlights a limitation of our controller designs. Recall that allocation rate pattern for Pmd was very oscillatory, quickly moving between high allocation rate spikes and areas of low allocation between. The controllers has no way to infer that this situation is happening and will happily follow the allocation pattern resizing the memory in a likewise fashion. The solution we propose again comes from control theory. Derivative control, generally used as part of a PID controller as described in Chapter 5, attempts to predict future changes in system error by monitoring the slope of the error signal. We can employ a similar technique here and instead of only monitoring the rate at which the allocations happen – $\frac{dA}{dt}$, we can also monitor the allocations *acceleration* $\frac{d^2A}{dt^2}$. This allows us to detect when allocation spikes happen and deal with them appropriately.

One such algorithm that may be applied is to scale up as usual when the first spike happens, if there is then a rapid change in the opposite direction it is likely that we have detected a spike. In this case the controller should move to a slow scale approach. If there are no other spikes detected in some time period t then we can return to fast control mode. If more spikes are detected before time t then it is likely that we have a program with similar allocation rate characteristics as pmd and this will need handled accordingly. There is a significant flaw in this approach. Many applications incur a similar allocation rate spike once the JIT compiler makes its optimisations. It therefore becomes difficult to distinguish between the virtual machine spike and a spike caused by the program itself – this is why we must always scale up quickly when the first spike is detected.

Although we have attempted to model the garbage collection system as a black box in practice we depend upon the heap layout of the collector. Our current system model depends upon having a generational garbage collector with a single region where all allocations are performed. This presents an issue when applying our system model to different collector types. For example the *Garbage First* (G1) algorithm (also present in HotSpot) uses a *region* based approach rather than a generational approach. The lack of an abstract, garbage collector independent model is a severe system limitation.

The aggressive resizing of only eden space and not the young generation survivor spaces also presents a subtle problem. The weak generation hypothesis implies that most objects will die young; but what if all the objects stay alive? In this situation we have a problem as we cannot fit all the objects into to-space. The course of action taken by the garbage collector is to promote all the objects into the old generation, as this is the only generation with enough space (assuming eden maintains its empty after garbage collection invariant). This issue with promoting early is that it is likely many of those objects are not actually long living yet they are taking up space in the old generation, this causes full garbage collections to occur more often which has a negative effect on overall system performance. To alleviate this issue the controller could maintain statistics on the average amount of live data being copied from eden at each garbage collection. This allows the controller to respond to high live object counts by scaling the to-space. Likewise with the allocation rate data it is unknown how good a predictor of future live data being copied that the current live data copied statistics are.

The controllers presented do not manage the old generation. This could prove to be a limitation for long running processes in which major collections are common. We believe that it would be possible to derive a similar system model which models all generations. This would allow a controller to optimise multiple generations in parallel. For example if we know that many objects are not being promoted it may be a good decision to scale the young generation upwards and reduce the memory allocation for the old generation, freeing up pages for the operating system.

Within our model we have assumed that the heap can resize up until a maximum bound. Expanding the heap requires additional pages from the operating system. On a heavily loaded system it is possible that paging will occur. Should paging occur our program throughput will diminish. A more optimised controller should detect this situation and use this as an additional parameter, by doing so our controller is much more applicable within the cloud environment.

8.2 Future Work

There are several areas of interest for future work within the area of dynamic heap resizing controller.

In Chapter 6 we showed two forms of allocation rate calculation. Our controllers track the allocation rate at each garbage collection based on the current eden size and the time since the last garbage collection. The other method is to track each allocation as it occurs on the allocation path. It would be interesting to contrast these two approaches and to experiment further with tracking on the allocation path. Two potential research questions are as follows:

- Can we still maintain high allocation performance even with the required allocation path mutex lock in place? What are the effects of losing some allocations should we decide to collect statistics in a non thread safe manner?
- Is there a benefit to the controller forcing an early garbage collection in order to scale sooner? For example if the controller predicts a spike is about to happen is it worthwhile making additional space before it happens?

Although we have presented four controller designs they are based on two key ideas with two scaling methodologies. An interesting area of investigation is the effects of dynamically changing the scaling method or even the full controller at runtime. For example, if we detect that the system is low on memory or beginning to thrash the virtual memory we may wish to move to a slow response controller to avoid requesting a high number of pages only to return them some short time later. If we do not mind a performance penalty but would like to reduce memory requirements it may be wise to switch to the fast throughput controller.

The key benefit of our approach is the strong grounding in a mathematical model, albeit one that makes numerous assumptions. By developing more rigorous mathematical approaches to system modelling it would be possible to develop more advanced controller designs with an array of different properties.

We believe control theory ideas are well suited to be applied within computing science. Controllers can be designed for high reliability and responsiveness. One such area of applicability is multi-processor designs, in which changing workloads need quickly rebalanced across processors.

Chapter 9

Conclusion

Building on the work of White et al.[17] we have further explored the application of control theory to dynamic heap resizing. We have shown how memory management and control theory can be described analogously. A system model was formulated which identifies the throughput of a garbage collected system and using this model four similar controller designs have been created. Analysis using a standard Java benchmarking suite shows that our controllers outperform a non-adaptive heap set up. Comparison with the current heap sizing techniques applied within the HotSpot JVM shows similar throughput performance. In many cases the average memory usage of our controller outperforms that of HotSpot's current controller.

We wanted to keep system performance tuning as simple and easy to understand as possible. Our controllers are significantly easier to understand than the current HotSpot alternative and, in general, give good performance without any user intervention. The overall simplicity of our controllers, small implementation size and mathematical grounding make it a viable alternative to current heap sizing mechanisms.

9.1 Contributions

- In Chapter 3 we provided a survey of current state of the art heap resizing algorithms.
- In Chapter 4 we show how a memory management system can be viewed as a control problem and how it fits into the standard control model.
- Chapter 7 shows how applying a mathematical model to the creation of a goal-based system can simplify heap resizing algorithms and move away from the current trends of heuristic algorithms.
- Finally, we have shown how to implement controller designs (Chapter 6) as a proof of concept system for the HotSpot Java Virtual Machine. From experimentation this proof of concept system shows how a simple, control-theory-based controller design can achieve similar performance to the current state of the art heap resizing algorithm present in the OpenJDK (version 7u) while generally showing a reduction in the overall memory usage (Chapter 7).

9.2 Personal Reflection

This project has allowed me to gain insight into the inner workings of many runtime systems specifically focusing on HotSpot. Comprehending such a system presented its own interesting challenges above and beyond the main project outcomes but I feel I have handled these well and learned a lot in the process.

As an electronic and software engineer this project presented an opportunity to combine both my areas of study. Determining a suitable (self defined) project and performing current state of the art analysis required the use of my research skills; system design and implementation required those of a software engineer and control theory ideas required that of an electrical engineer.

I had my doubts at the beginning of this project that my idea would give comparable performance to current state of the art implementations. After all I had limited experience in this area. In the end my doubts were unfounded and I can indeed solve real world problems. The biggest lesson I have learned is that any idea, no matter how small, is worth pursuing. If you put in the time and effort you never know where it will lead or what results you might get.

9.3 Acknowledgements

I would like to thank Jeremy Singer and Callum Cameron for their assistance and feedback throughout this project.

Bibliography

[1] GHC Source Code Commentary: GC.c.

[2] 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 *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, October 2006. ACM Press.

[3] N Gandhi, DM Tilbury, Y Diao, J Hellerstein, and S Parekh. Mimo control of an apache web server: Modeling and controller design. In *American Control Conference, 2002. Proceedings of the 2002*, volume 6, pages 4922–4927. IEEE, 2002.

[4] GHC Team. Glasgow haskell compiler homepage. `http://www.haskell.org/ghc/`. Accessed March 5, 2014.

[5] Singer J and Cameron C. Personal correspondence, 2014.

[6] Simon Marlow, Tim Harris, Roshan P James, and Simon Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap. In *Proceedings of the 7th international symposium on Memory management*, pages 11–20. ACM, 2008.

[7] Thomas J McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, (4):308–320, 1976.

[8] OpenJDK hotspot group. OpenJDK hotspot group homepage. `http://openjdk.java.net/groups/hotspot/`. Accessed March 5, 2014.

[9] Oracle. Java se 6 hotspot[tm] virtual machine garbage collection tuning. `http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html`. Accessed October 9, 2013.

[10] T. Patikirikorala, A. Colman, J. Han, and Liuping Wang. A systematic survey on the design of self-adaptive software systems using control engineering approaches. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2012 ICSE Workshop on*, pages 33–42, June 2012.

[11] Sunil Soman, Chandra Krintz, and David F. Bacon. Dynamic selection of application-specific garbage collectors. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 49–60, New York, NY, USA, 2004. ACM.

[12] Adam J Storm, Christian Garcia-Arellano, Sam S Lightstone, Yixin Diao, and Maheswaran Surendra. Adaptive self-tuning memory in db2. In *Proceedings of the 32nd international conference on Very large data bases*, pages 1081–1092. VLDB Endowment, 2006.

[13] Azul Systems. Jvm parameter tuning examples. `http://www.azulsystems.com/technology/jvm-tuning-parameters-comparison`. Accessed March 26, 2013.

[14] Jikes Team. Jikes research virtual machine homepage. `http://jikesrvm.org/`. Accessed March 5, 2014.

[15] Jikes Team. Jikes research virtual machine project progress. `http://docs.codehaus.org/display/RVM/Project+Status`. Accessed March 20, 2014.

[16] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM Sigplan Notices*, 19(5):157–167, 1984.

[17] David R White, Jeremy Singer, Jonathan M Aitken, and Richard E Jones. Control theory for principled heap sizing. In *Proceedings of the 2013 international symposium on International symposium on memory management*, pages 27–38. ACM, 2013.

[18] Paul R Wilson. Uniprocessor garbage collection techniques. In *Memory Management*, pages 1–42. Springer, 1992.

[19] Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, Arif Merchant, Pradeep Padala, and Kang Shin. What does control theory bring to systems research? *SIGOPS Oper. Syst. Rev.*, 43(1):62–69, January 2009.

Appendices

Appendix A

Tables of Results

A.1 Fast Spacing Controller

Metric	Benchmark	No Controller	Ergonomics	Fixed Spacing Controller
Average Runtime (ms)	avrora	10845.25	10633.60	10698.20
	fop	3502.65	2476.90	3089.15
	jython	29583.45	24118.30	24164.35
	luindex	1906.10	1847.00	1896.75
	lusearch	36319.70	31111.25	32468.30
	pmd	53157.20	40641.60	44290.25
	sunflow	17100.20	15360.95	15578.70
	tomcat	20872.30	19624.40	19923.05
	xalan	13687.10	11698.95	11921.30
Average Throughput (%)	avrora	99.18	99.45	99.16
	fop	65.03	92.07	74.74
	jython	77.77	94.68	93.38
	luindex	90.82	93.08	93.42
	lusearch	82.37	91.30	91.78
	pmd	72.17	94.01	86.48
	sunflow	86.08	92.95	93.15
	tomcat	89.58	96.88	93.15
	xalan	85.32	94.55	93.43
Average Memory Usage (Bytes)	avrora	63488.00	94440.73	63488.00
	fop	63488.00	184176.64	124753.70
	jython	63488.00	210788.85	155911.76
	luindex	63488.00	118784.00	63707.43
	lusearch	63488.00	216911.82	214865.81
	pmd	63488.00	249030.34	130115.37
	sunflow	63488.00	193593.27	197531.02
	tomcat	63488.00	198322.89	81045.39
	xalan	63488.00	210855.72	132959.42

Table A.1: Fast One Second Controller Comparison

A.2 Slow Spacing Controller

metric	benchmark	no controller	ergonomics	fixed spacing controller
average runtime (ms)	avrora	10634.75	10571.15	10860.95
	fop	3473.55	2474.15	3357.35
	jython	30783.15	24152.15	25442.75
	luindex	1923.25	1810.05	1870.65
	lusearch	36268.45	30988.20	32624.30
	pmd	52341.20	40407.75	52404.64
	sunflow	17131.80	15577.10	15564.90
	tomcat	20901.60	19678.85	19603.00
	xalan	13446.55	12375.70	12041.05
Average Throughput (%)	avrora	99.17	99.50	99.15
	fop	65.14	92.14	67.45
	jython	78.08	94.70	92.38
	luindex	90.03	94.74	92.28
	lusearch	82.47	91.28	91.68
	pmd	72.65	94.47	72.64
	sunflow	86.08	93.06	92.82
	tomcat	89.35	96.89	93.17
	xalan	84.99	94.83	92.99
Average Memory Usage (Bytes)	avrora	63488.00	92649.74	63744.00
	fop	63488.00	183685.12	78373.33
	jython	63488.00	210796.61	120719.78
	luindex	63488.00	118579.20	63890.29
	lusearch	63488.00	216909.34	186259.25
	pmd	63488.00	261654.65	65807.96
	sunflow	63488.00	192077.32	153581.13
	tomcat	63488.00	197786.22	81387.46
	xalan	63488.00	210855.72	117131.90

Table A.2: Slow One Second Controller Comparison

A.3 Fast Throughput Controller

Metric	Benchmark	No Controller	Ergonomics	Throughput Controller
Average Runtime (ms)	avrora	10829.65	11141.80	10934.35
	fop	3492.80	2478.40	3210.65
	jython	30130.80	24778.80	25033.25
	luindex	1915.10	1808.50	1870.30
	lusearch	36534.05	30871.55	32599.25
	pmd	51151.70	40728.45	48680.90
	sunflow	17312.10	15350.05	15653.20
	tomcat	21007.05	19440.55	20478.90
	xalan	13462.55	11447.00	12135.80
Average Throughput (%)	avrora	99.18	99.40	99.18
	fop	65.10	92.10	71.24
	jython	78.22	94.81	92.89
	luindex	89.74	96.68	89.21
	lusearch	82.54	91.30	91.51
	pmd	74.60	94.32	78.48
	sunflow	86.44	93.07	92.99
	tomcat	89.49	96.91	91.99
	xalan	84.99	94.46	92.91
Average Memory Usage (Bytes)	avrora	63488.00	69825.42	63488.00
	fop	63488.00	184709.12	100693.33
	jython	63488.00	211497.58	139269.85
	luindex	63488.00	96395.64	68672.00
	lusearch	63488.00	216912.65	176758.25
	pmd	63488.00	239629.13	113602.23
	sunflow	63488.00	192171.81	169882.76
	tomcat	63488.00	197644.34	73400.23
	xalan	63488.00	210785.10	115353.97

Table A.3: Fast Throughput Controller Comparison

A.4 Slow Throughput Controller

Metric	Benchmark	No Controller	Ergonomics	Throughput Controller
Average Runtime (ms)	avrora	10935.30	10727.80	10740.10
	fop	3490.95	2472.90	3358.05
	jython	29669.15	24271.00	25056.75
	luindex	1888.90	1818.45	1890.50
	lusearch	36607.20	30931.25	32942.00
	pmd	51195.85	40615.55	49625.70
	sunflow	17113.70	15519.85	15798.10
	tomcat	20938.10	19314.35	20439.55
	xalan	13651.25	11745.30	12396.90
Average Throughput (%)	avrora	99.14	99.39	99.16
	fop	65.25	92.20	67.95
	jython	77.58	94.81	91.70
	luindex	93.35	94.67	92.56
	lusearch	82.51	91.34	91.42
	pmd	74.23	94.36	77.21
	sunflow	86.21	93.09	92.56
	tomcat	89.48	96.85	92.18
	xalan	85.46	94.55	92.58
Average Memory Usage (Bytes)	avrora	63488.00	69893.69	63744.00
	fop	63488.00	182784.00	78202.67
	jython	63488.00	210804.36	114161.07
	luindex	63488.00	118118.40	65772.31
	lusearch	63488.00	216914.30	162972.98
	pmd	63488.00	243069.75	70575.83
	sunflow	63488.00	190993.07	144018.61
	tomcat	63488.00	199967.22	73720.00
	xalan	63488.00	210754.21	106674.23

Table A.4: Slow Throughput Controller Comparison