The Economics of Garbage Collection

Jeremy Singer
University of Manchester
United Kingdom
jsinger@cs.man.ac.uk

Richard Jones
University of Kent
United Kingdom
R.E.Jones@kent.ac.uk

Gavin Brown Mikel Luján
University of Manchester
United Kingdom

Abstract

This paper argues that economic theory can improve our understanding of memory management. We introduce the *allocation curve*, as an analogue of the demand curve from microeconomics. An allocation curve for a program characterises how the amount of garbage collection activity required during its execution varies in relation to the heap size associated with that program. The standard treatment of microeconomic demand curves (shifts and elasticity) can be applied directly and intuitively to our new allocation curves. As an application of this new theory, we show how *allocation elasticity* can be used to control the heap growth rate for variable sized heaps in Jikes RVM.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Memory management (garbage collection)

General Terms Economics, Measurement

Keywords Microeconomics, Allocation curve, Elasticity, Memory management, Garbage collection, Java

1. Introduction

The memory behaviour of garbage collected programs is complex. As well as empirical studies (for instance, [14, 17, 22, 27, 29]), a wide range of analytical studies exist. Some researchers have used concepts from other fields as *metaphors* for various aspects of memory management. For instance, Baker shows how concepts from statistical thermodynamics can be applied to regions of the heap [5]. Clinger and Rojas combine linearly different radioactive decay models of object lifetime distributions to model generational garbage collection [12]. Stefanović et al. explore well-known families of distributions using statistical techniques [28]. In this work, we apply some theory from *microeconomics* to characterize program behaviour with different automatic memory management configurations.

Economic theory has many parallels with the analysis of garbage collection performance. Both disciplines provide mathematical apparatus that attempts to understand, summarize and modify complex systems with numerous hidden variables.

In this paper, we show how an economic *demand curve* for a commodity has a clear analogue in the garbage collection domain. To the best of our knowledge, this is the first time that economic theory has been used in the context of automatic memory management. There are two main aims to our work. First, we intend to use economic theory to improve our understanding of memory management, by identifying parallels between concepts in each domain. Second, we aim to apply economic theory to control and optimize memory management behaviour.

The main contributions of this paper are:

1. the *allocation curve* as a graphical characterization of how a program interacts with GC for a range of fixed heap sizes.
2. an empirical investigation to show that the allocation curve has a standard shape across many Java benchmarks, for different garbage collection algorithms.
3. *allocation elasticity* to characterise the sensitivity of an application to changes in heap size.
4. the application of allocation elasticity to manage the growth of variable size heaps for Java benchmarks executing in Jikes RVM.

2. Microeconomics Background

Microeconomics studies interactions in a single market, i.e. relating to the supply and demand of a single commodity. The market may be affected by product scarcity or regulation. The theory considers the quantity demanded by buyers and the quantity supplied by sellers at each possible price per unit. This is known as *supply and demand analysis*.

A *demand curve* is a graph that shows the relationship between the price of a certain commodity, and the quantity that consumers will purchase at that given price. The negative slope of the curve is due to the *law of demand*, which states:

> If all other factors remain equal, the higher the price of a good, then the less people will demand that good.

In other words, the amount demanded of a commodity and its price are *inversely related*, other factors remaining constant. Figure 1 shows an example demand curve. By convention, the price p is specified on the vertical axis and the quantity q is given on the horizontal axis. Note that microeconomists plot the independent variable (e.g. the price) on the vertical axis and the dependent variable (e.g. the quantity demanded) on the horizontal axis, in contrast to the usual mathematical convention.

The demand curve was adopted and popularized by the British economist, Alfred Marshall, in his 1890 textbook *Principles of Economics*. Since then, it has become one of the most well-known aspects of microeconomic theory.

The relation between price of a good and its demand may vary. A *movement* refers to a change along a demand curve. If the price of a commodity changes, then the quantity demanded by a consumer will also change. Note that the demand relationship remains constant. Rather an individual consumer has moved from

Figure 1: An example demand curve for a commodity

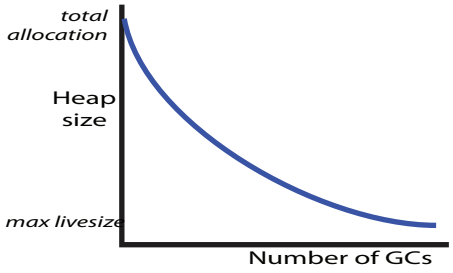


Figure 2: An example allocation curve for a program

one point to another, along the demand curve. In other words, a movement occurs when a change in the quantity demanded is caused only by a change in price, and vice versa.

A *shift* in a demand curve occurs when the quantity of a commodity demanded by consumers changes even though price remains the same, i.e. the demand is influenced by some other factor. An example may be the relative scarcity of alternative products. Shifts in the demand curve imply that the original demand relationship has changed. Graphically, the demand curve is shifted in relation to the axes.

Demand *elasticity* E measures the sensitivity of the quantity q demanded to changes in price p. It can be calculated as:

$$E = \frac{\%\text{change in quantity}}{\%\text{change in price}} = \frac{dq}{dp}\frac{p}{q} \quad (1)$$

Demand for a product is *inelastic* if consumers will pay almost any price for that product, *e.g.* life-saving drugs. Inelastic demand is denoted by a steep gradient in the demand curve (a large change in price hardly changes the quantity demanded). On the other hand, demand for a product is *elastic* if consumers will only pay a certain price, or a small range of prices, for the product. For instance, as the price of a particular foodstuff increases, consumers may turn to substitute foods instead. Elastic demand is denoted by a shallow gradient in the demand curve (a small change in price causes a dramatic change in the quantity demanded). Note that the elasticity of a good is not necessarily constant for all points on the demand curve.

3. Garbage Collection Analogy

We suggest that there are connections between microeconomics and memory management, and that these may enable us to gain a better understanding of memory management, and give us new techniques to analyse and optimize it. We introduce the *allocation curve* as a new graphical representation of a program's allocation activity and interaction with the garbage collector. The allocation curve incorporates several existing measures, and adds new information. The allocation curve is the analogue of the demand curve from microeconomics. Given a particular program, the allocation curve shows the relationship between the size of the heap used by that program, and the number of garbage collections required during the programs execution.

We relate the heap size to the *price* variable from the demand curve. In terms of program execution, the two standard costs are time and space. In terms of memory management, the primary cost is space. Thus it seems reasonable to treat heap size as the analogue of price, which is the independent variable in the demand curve, plotted on the vertical axis.

We relate the number of GCs to the *quantity* variable from the demand curve. A GC enables re-use of memory, which represents consumption of resource, in some sense. Thus it seems reasonable to treat number of GCs as the analogue of quantity, which is the dependent variable in the demand curve, plotted on the horizontal axis.

The allocation curve will have the same shape as the demand curve if programs' allocation behaviour satisfies something equivalent to the law of demand. Recall that this microeconomic law states that 'quantity and price are inversely related'. It is well-known that the number of GCs required by a program, given an appropriate scheduling of collections, is inversely correlated to heap size. We therefore summarise this relationship with the analogous *law of allocation*:

> If all other factors remain equal, the larger the heap size, then the fewer GCs required.

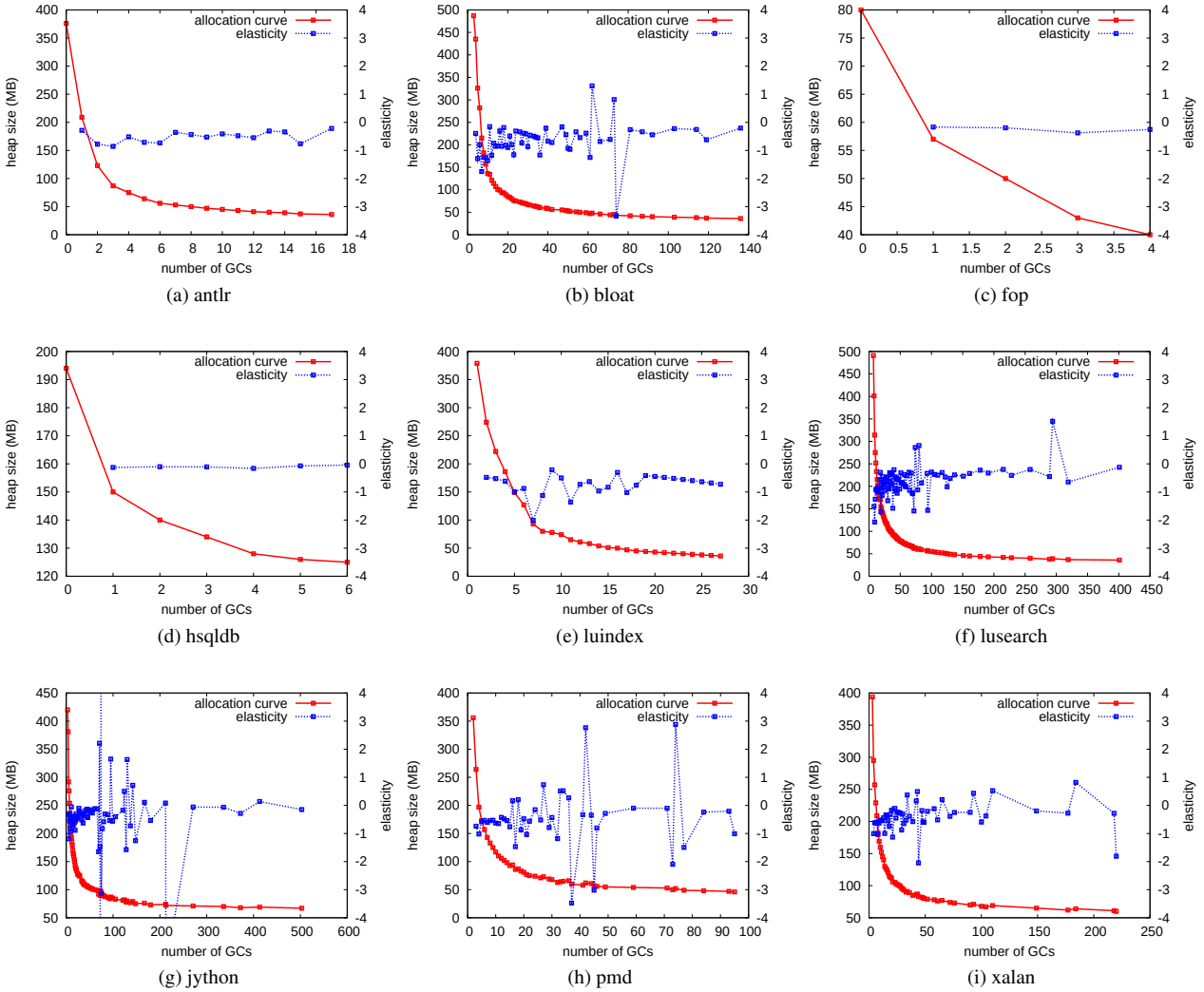
It makes intuitive sense that heap size should be inversely related to the number of GCs, since a larger heap will require fewer GCs for the same schedule of allocation.

Figure 2 shows the graph for an example allocation curve. The amount of memory required for successful program execution is given on the vertical axis. The number of garbage collections that occur during program execution is given on the horizontal axis. A point (x, y) on the allocation curve indicates that y is the *minimum* heap size for which the program may successfully complete execution with not more than x garbage collections, i.e. given an ideal scheduling of GCs. Obviously slightly larger memory sizes might enable the same program execution to complete with the same number of GCs. Garbage collection is *chaotic* in the formal sense that small changes in GC schedules can lead to large changes in the number of GCs required and the work that they must do, and hence to large changes in program performance [8]. Thus, in order to plot the perfect allocation graph, it would be necessary to execute a program with not only all possible different fixed heap sizes but also for all possible GC schedules, and to monitor the number of GCs required in each case. We describe our experimental method used to collect the empirical data presented and to compute approximate allocation curves in the following sections.

The two extreme points of the allocation curve are interesting. The leftmost point on the curve, when the number of GCs is 0, indicates the *total amount of memory allocated* by the program during its entire execution. This is equivalent to the heap size required for successful execution without a garbage collector, such as the `NoGC` policy in Jikes RVM / MMTk.

The furthest point on the right of the allocation curve, when the number of GCs tends to $+\infty$, indicates the *minimum heap size* in which this program will execute. This is equivalent to the high-watermark of the live data set throughout program execution, the *max livesize* metric [24].

The shape of the allocation curve is related to the uniformity of object death rate. For instance, if a program constantly allocates only short-lived objects for its entire execution, then the allocation



(a) antlr (b) bloat (c) fop

(d) hsqldb (e) luindex (f) lusearch

(g) jython (h) pmd (i) xalan

Figure 3: Allocation curves for selected DaCapo benchmarks using a full-heap mark-sweep collector

curve will have a constant negative gradient. Each GC is 'worth' as much as every other GC, since it removes the same amount of short-lived objects. On the other hand, if a program has a significant amount of long-lived objects, then the allocation curve will decay to an asymptote, as it reaches the point where extra GCs do no useful work since remaining objects are still live.

Taking g as the number of GCs, and h as the heap size, then it is important not to confuse the *gradient* dg/dh of the allocation curve with its *elasticity*, E.

$$E = \frac{\%\text{change in number of GCs}}{\%\text{change in heap size}} = \frac{dg}{dh}\frac{h}{g} \qquad (2)$$

Section 5.4 investigates the empirical properties of allocation elasticity. In summary, it is a measure of the sensitivity of the heap size to the number of GCs, and vice versa.

4. Related Work

The garbage collection literature contains many empirical and analytical studies of memory behaviour. The scope of the empirical studies ranges from low level analysis of the proportions of different kinds of objects in the heap and proportions of heap loads and stores [14], to studies of nursery survival rates and object mortality (e.g. [7, 22, 29]), to selection of garbage collection algorithms for different programs [15, 26, 27].

Stefanović et al. explored analytical object lifetime distribution models using statistical techniques to compare the models with observed behaviour for a large number of Java and Smalltalk programs [28]. They concluded that none of the well-known distribution families they investigated were entirely satisfactory. Baker offered a radioactive decay model of object lifetimes as an example of a distribution that cannot benefit from generational garbage collection. Although Baker expected a generational collector to perform neither better nor worse than a non-generation collector on this model, Clinger and Hansen calculated that younger-first generational collection would perform worse [11]. Clinger and Rojas argued, however, that a linear combination of these models (for young, intermediate and older objects) was adequate to model simple generation collectors [12].

Several studies have explored models and mechanisms to control program performance by varying parameters such as heap size. Vengerov provides an analytic model for the throughput of pro-

grams running on Sun's generational throughput collector [31]. The analysis is used to improve performance, not by varying overall heap size, but by adjusting the relative sizes of spaces in the HotSpot collector and the tenuring threshold (the number of minor collections that an object must survive before it is promoted) used.

Most other related work has investigated how to vary heap sizes to avoid paging (e.g. [1, 9, 13, 19, 20, 33, 34]. Yang et al [33, 34] employ reuse distance histograms and a simple linear model of the heap required; they require a modified virtual memory manager. The *Page-level Adaptive Memory Manager* attempts to discover the optimal heap size for a number of applications running on a machine, taking advantage of program's phase behaviour [35]. Grzegorczyk et al use allocation stalls as a warning of impending GC-induced paging [19]. Hertz et al introduce the *time-memory curve*, the shortest running time of a program in any heap size for a given amount of physical memory [20]. They use a 'whiteboard' to allow programs to exchange page fault and resident set size information in order to coordinate collections.

Few authors have used analogies to describe garbage collection. Baker presents an analogy between abstract statistical thermodynamics and garbage collection, equating energy with information in the heap: the mutator adds information/entropy by allocating and mutating objects [4]. The 'temperature' of the heap is the change in information resulting from the addition of one unit of 'energy', or equivalently the marginal cost to recover one unit of storage. From this, Baker shows that the weak generational hypothesis requires a younger generation to be 'cooler' than an older generation. Grossman discusses the similarities between garbage collection and transactional memory [18]. He argues that transactional memory provides solutions to the problem of safe concurrent programming with shared memory in the same ways that garbage collection does for programming with dynamic memory.

The only discussion of economics and dynamic memory management that we are aware is from Wilson et al [32]. They suggested that allocators that make poor use of memory might have incurred a real economic cost of over one billion US dollars worldwide in 1995 (on the assumption that there was $30 billion of RAM at stake: 100 million PCs, each with 10 MB of memory at $30 per megabyte). Interestingly, this monetary value has hardly changed in 15 years. Although there are more than a billion PCs today [16], say each with at least 512 MB of RAM, memory prices have plummeted to about $40 per GB.

5. Empirical Allocation Curves

This section presents empirical data from real-world Java applications, running on a state-of-the-art virtual machine. Our experimental results confirm that, over a wide range of Java programs, for two different GC algorithms, the allocation curves generated do have the properties we suggested in Section 3.

Empirically observed allocation curves are specific to a particular runtime system and garbage collection regime. For our experiments, we analyse Java benchmarks from the DaCapo suite [6], version 2006-10-MR2, using the `default` input sets. We execute Java code on the Jikes RVM system, v3.1.0 [2, 3] which is configured for an x86-64 Linux distribution. We use the *replay compilation* facility of Jikes RVM [21]. We record the latest compilation options for each method on a steady-state iteration of each benchmark. Then we re-execute each benchmark with this compilation profile data and take GC measurements on the *second iteration* of each benchmark execution. Since all optimizing compiler activity should occur in the first benchmark iteration with replay compilation; this is not included in our allocation data collection. As a caveat, we note that the overall heap size has to be sufficiently large for the optimizing compiler to operate correctly in the first iteration, which may somewhat inflate the overall memory requirements.

First we generate allocation curves for a simple full-heap collector, in Section 5.1. Then we switch to a generational collector, and generate more allocation curves in Section 5.2. We show that allocation curves can be shifted in the same way as demand curves, in Section 5.3. We discuss the concept of elasticity in Section 5.4 and show how some benchmarks are more elastic than others.

5.1 Full-Heap Collection

For this initial set of allocation curves, we use a full-heap collector for the sake of simplicity. The actual GC algorithm is `MarkSweep` from the MMTk framework, and the Jikes RVM configuration is `FastAdaptiveMarkSweep`.

In order to generate the allocation curve, we iterate over a number of fixed heap sizes, from 35MB to 500MB for all benchmarks. Note that we ignore explicit `System.GC` calls from the application code, since these cause spikes in the allocation curve. Strictly they are 'unnecessary' collections. At each fixed heap size, we execute each DaCapo benchmark and record the following data into a log file:

1. whether the benchmark completed execution successfully.
2. how many GCs occurred during benchmark execution.
3. the fixed size of the heap.

From the log files, it is possible to determine the minimum heap size for each number of GCs. This gives the information required to plot the allocation curve for each benchmark. The use of replay compilation avoids that part of the GC noise which is caused by the non-determinism of standard adaptive compilation.

Figure 3 shows the allocation curves for the DaCapo benchmark executions with a full-heap collector, using the experimental setup outlined above. Note that we omit analysis of the chart and eclipse benchmarks, since these do not execute properly on our test system.

In all cases, the empirical allocation curves generally obey the law of allocation. There are a few exceptional points that are not monotonically decreasing, due to unfortunate scheduling of GCs. Some benchmarks (fop and hsqldb) have a max livesize of the same order of magnitude as total allocation. There are few points on these curves. Other benchmarks (notably jython and xalan) have a pronounced 'knee' in the allocation curve, with a long tail as number of GCs tends to $+\infty$. Such benchmarks allocate very large amounts of short-lived data (in relation to their max livesize).

5.2 Generational Collection

We now switch to using a generational collector, which is the standard algorithm for most high-performance automatic memory management systems [23]. The insight underlying generational GC is the *weak generational hypothesis* [30] that most objects die young. By default, objects are allocated into a frequently collected nursery space, with the expectation that most objects will die after a small number of nursery collections. Long-lived objects are promoted to the less frequently collected mature space. This concentrates GC activity on an area of the heap that has the highest density of short-lived objects, thus improving the efficiency of the GC algorithm.

This second empirical investigation has two aims:

1. to determine whether allocation curves are general enough to work with more complex GC algorithms such as generational collection.
2. to determine whether an allocation curve for a program has a similar shape across different GC algorithms, i.e. is an allocation curve invariant to the GC regime?

Since our allocation curve framework only uses a single variable to measure 'number of GCs', we must relate minor (nursery) GCs and major (full-heap) GCs numerically. The admittedly simple

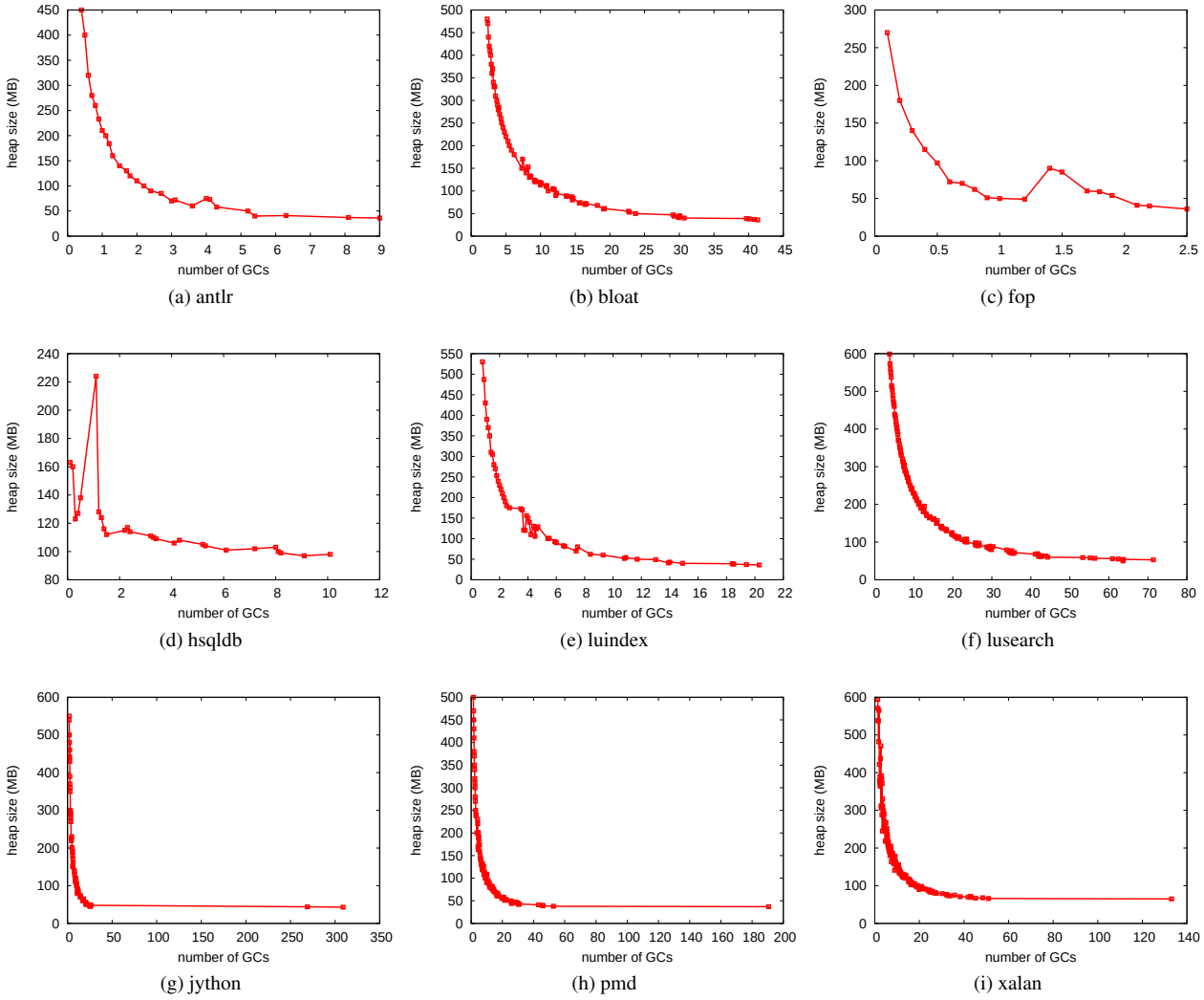


Figure 4: Allocation curves for selected DaCapo benchmarks using a generational collector

solution we adopt is to *weight* nursery GCs according to the size of the nursery in relation to the total size of the heap. For instance, say we have a fixed heap of size X, and we fix the nursery to occupy 10% of X. This allows us to calculate that a nursery GC is worth $0.1\times$ a full-heap GC.

In order to generate the allocation curve for generational GC, we use the `GenMS` collector from MMTk, with the `FastAdaptiveGenMS` configuration for Jikes RVM. We iterate over a number of fixed heap sizes, from 35MB to 500MB for all benchmarks, setting the fixed-size nursery to occupy 10% of the overall heap size. As before, we execute each DaCapo benchmark and record the appropriate data into a log file. We find minimum heap size for each number of GCs. Figure 4 shows the allocation curves for some DaCapo benchmark executions, using this setup.

The allocation curves are less smooth, since we are equating ten minor collections with one major collection which is not really fair. This accounts for the spikes, for instance in hsqldb.

However, generally the allocation curves have the same basic shape as for full heap collection. Overall, the allocation behaviour appears to follow the *law of allocation* outlined above. It is not the case that individual benchmarks have similar curves for both full heap and generational collection. In most cases they are markedly different.

5.3 Shifting the Allocation Curve

In microeconomics, the demand curve for a commodity may *shift*. This means the curve maintains its shape, but is translated along the p or q axis. A shift occurs when there is a change in a factor that influences the required quantity of that commodity, other than the underlying price. For example, an increase in the sales tax would raise the unit price of a commodity, which shifts the demand curve up the p-axis without changing its shape.

An equivalent scenario may happen in memory allocation. Suppose that objects become uniformly larger, perhaps as a result of an increased number of object header words. In effect, the object header is a 'tax' on the benchmark memory usage, imposed by the VM. Then the allocation curve is shifted upwards, as more memory is required for the same number of objects, although garbage collection is correspondingly able to recycle the additional memory at the same rate, since objects die at the same rate as previously.

Figure 5 shows the allocation curves for selected DaCapo benchmark executions, using the same `FastAdaptiveMarkSweep`

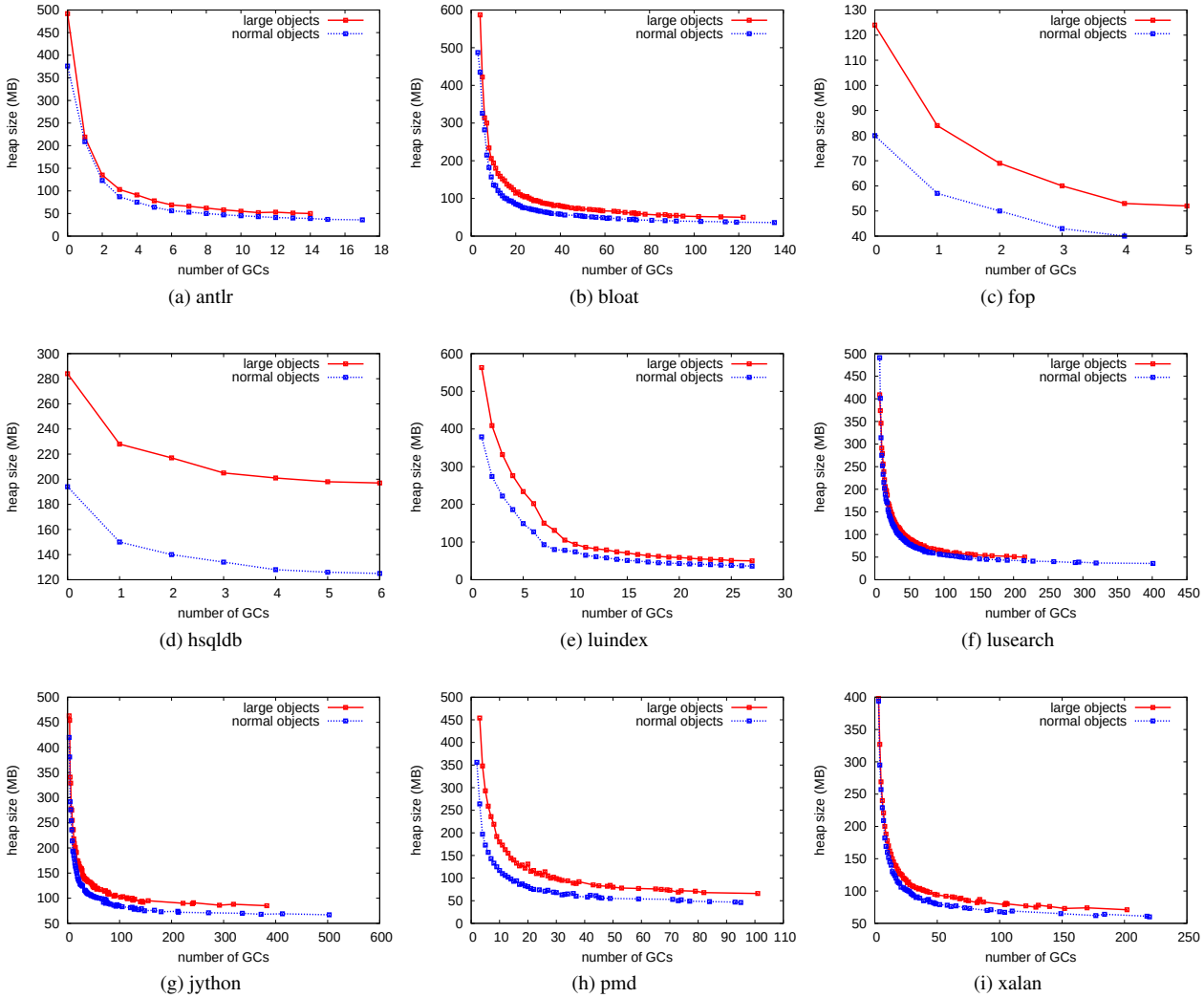


Figure 5: Allocation curves for selected DaCapo benchmarks, showing the curve for standard objects, and for objects with larger headers.

configuration as before. In each graph there are two allocation curves. The lower curve is for the standard Jikes RVM object model. The upper curve is for a non-standard object model, which includes an additional (empty) payload of four 32-bit words in every object header. Note that the upper curve, for larger objects, generally follows the same shape as the lower curve, only it is shifted up the vertical axis.

For some benchmarks, the large object allocation curve is only slightly higher than the normal object allocation curve. For instance, the gap is around 10MB for the lusearch benchmark. However, other benchmarks have large gaps between the curves. For instance, there is a distance of over 50MB for the hsqldb benchmark. This variation can be explained by the *average size* of objects in each benchmark. In a heap with many, comparatively small objects, the increased overhead for larger object headers is more pronounced than in a similar sized heap with fewer, comparatively larger objects.

5.4 Elasticity

Microeconomists use *elasticity* to describe how relative changes in one variable affect another variable. Demand elasticity measures how much the quantity demanded changes when the price changes. Demand elasticity can be computed at any point on a demand curve. Equation 1 gives the formula for computing demand elasticity.

We introduce *allocation elasticity* as a measure of how the number of GCs g changes when the heap size h changes. Allocation elasticity can be computed at any point on an allocation curve. Equation 2 gives the formula for computing allocation elasticity at a point. However, we cannot use this formula for our empirical allocation curves since the curve is not analytically differentiable. Instead, we use *arc elasticity*, which estimates the elasticity between two points (g_0, h_0) and (g_1, h_1) on the curve. Equation 3 gives the arc elasticity formula, which follows directly from the analogous arc elasticity for demand in microeconomics.

$$E = \frac{(h_1 - h_0)}{(h_1 + h_0)} \frac{(g_1 + g_0)}{(g_1 - g_0)} \tag{3}$$

Elasticity is always negative for curves that are monotonically decreasing. Because of this, economists generally ignore the sign. However we retain the sign since some of our empirical allocation

$$\text{currElasticity} = -1.\frac{\text{numGCs since last heap expansion}}{\text{heap size change at last expansion}}.\frac{\text{heap size before last expansion}}{\text{numGCs from start to last heap growth}} \quad (4)$$

Figure 6: Formula to compute the current elasticity during execution with a variable size heap

curves are slightly choppy, causing allocation elasticity to turn positive occasionally.

Elasticity varies along the allocation curve, as can be seen from Figure 3. These graphs plot the arc elasticity between adjacent points on the allocation curve. Since the allocation curve is not entirely smooth, and the points are not equidistant on either the g or the h axis, the elasticity is spikey. However it can be seen that elasticity is generally negative, except for points where the allocation curve does not decrease monotonically.

An allocation curve (or portion of the curve between two points) is said to be *inelastic* if $|E| < 1$. This is indicated by a steep slope in the allocation curve. The intuition behind this is that a small change in the number of GCs gives rise to a large change in the heap size, since the heap space is *overprovisioned*: heap size $\gg$ livesize so we are simply wasting space, although reducing the GC work required. For example, consider the allocation curve for the xalan benchmark in Figure 3(i), which clearly exhibits inelastic properties near the left hand side of the curve, as the number of GCs tends to 0.

Conversely, an allocation curve (or portion between two points) is said to be *elastic* when $|E| > 1$. This is indicated by a shallow slope in the allocation curve. A large change in the number of GCs causes relatively little change in the heap size. The intuition behind this is that heap size $\approx$ livesize so most GCs are not able to recycle much memory. For example, consider the allocation curve for the bloat benchmark in Figure 3(b), which tends to show increasingly elastic properties when $h > 55\text{MB}$.

6. Controlling Heap Growth using Elasticity

So far in this paper, we have applied microeconomic theory to characterize Java benchmark allocation behaviour. We have improved our understanding of garbage collection using the analogy with demand curves. However in this section, we go a step further. We use the microeconomic concept of *elasticity* to improve GC performance, by controlling the expansion rate for a variable size VM heap.

Often, a Java application is executed with a variable size heap. This may happen if the dynamic memory requirements of the application are not quantified ahead-of-time. In such cases, the VM user may simply specify an initial and a maximum heap size or may use the system's defaults. During the application execution, the VM may vary its heap size to any value up to the maximum limit.

In Jikes RVM, the `HeapGrowthManager` class is responsible for controlling the size of a variable heap. The default implementation uses a simple *heuristic* to determine whether the heap size should vary. The heuristic is based on the current *GC load* (i.e. the amount of time spent on the nth GC in relation to the total execution time since the end of the $(n-1)$th GC) and the current *live ratio* (i.e. the proportion of data on the heap that is not garbage, after the nth GC). These two values are used as indices in a two-dimensional lookup table of empirically determined heap growth ratios. Generally, if the live ratio and the GC load are relatively high, then the lookup value v is greater than 1, which causes the heap to grow to v times the current heap size. The heuristic is recomputed after every GC invocation. The lookup table values are determined solely by experimental tuning. The Jikes RVM developers report that the heuristic is 'somewhat stable and makes reasonable decisions.' However there is some sensitivity to the values, i.e. if they are perturbed significantly then variable heap size decisions are affected.

The main problem with the default heap growth heuristic is that it is opaque. It is not exposed to the VM user, and even if it were, cannot be explained or modified simply. Instead an expert VM programmer has to tweak the lookup table ratio values, and recompile Jikes RVM each time she wants to evaluate a change.

We investigate an alternative heuristic for heap growth management, based on the *allocation elasticity* of the executing application. When Jikes RVM is initialized, the user may specify a *target elasticity* value E as a command line option. The VM or benchmark documentation would have to provide some guidance on the range of suitable elasticity values, or the user would have to conduct experiments to determine an appropriate value. As the application executes, its current allocation elasticity is computed after each GC event, according to the formula in Figure 6. If the magnitude of the current elasticity *exceeds* that of the target E, then the heap size increases. Otherwise the heap size remains constant. The equation in Figure 6 computes the arc allocation elasticity between the start of application execution and the current time.

A large (negative) E value means that many GCs have to occur at the current heap size before the current allocation elasticity exceeds the threshold. This slows down the rate of heap growth. On the other hand, a small (negative) E value means that few GCs occur before the (negative) allocation elasticity falls below the threshold, which means the heap can expand rapidly.

The motivation for allowing the user to supply a target elasticity value ahead-of-time is to provide a *limit* on the rate of heap expansion. It enables the user to control how readily the VM can acquire new heap space.

To evaluate this elasticity-based heap expansion policy in Jikes RVM, we measure execution times for the same DaCapo benchmarks as earlier, specifying various E values between 0.1 to 10. All benchmark executions take place in a modified version of Jikes RVM 3.1.0, `FastAdaptiveMarkSweep` configuration. The evaluation platform is a lightly loaded Intel Core i7/920 machine clocked at 2.67GHz, with 6GB RAM, installed with x86_64 Linux 2.6.27.

For each benchmark/elasticity combination, we schedule 10 executions. We gather timing data from the second iteration of each benchmark execution, with replay compilation. As earlier, Jikes RVM is configured to ignore explicit `System.GC()` requests from the application. We report arithmetic means for times, with standard deviations, over the 10 runs.

The heap growth policy is to commence benchmark execution with a 25MB heap, and to expand the heap by multiplying its size by a fixed growth ratio after each GC when the current elasticity exceeds the target elasticity, E. This is an *exponential* heap growth model. Alternative growth models, such as linear, are possible within the same framework.

Figure 7 shows execution time results for selected DaCapo benchmarks, with variable sized heaps controlled using the elasticity heuristic outlined above. The E value is varied along the x-axis, note the log scale. The execution times are measured on the y-axis. Confidence intervals are one standard deviation either side of the arithmetic mean, for each result. We evaluate three different heap growth ratios: 1.1, 1.3, and 1.5. (The lookup table in the default `HeapGrowthManager` implementation has ratios in the same range

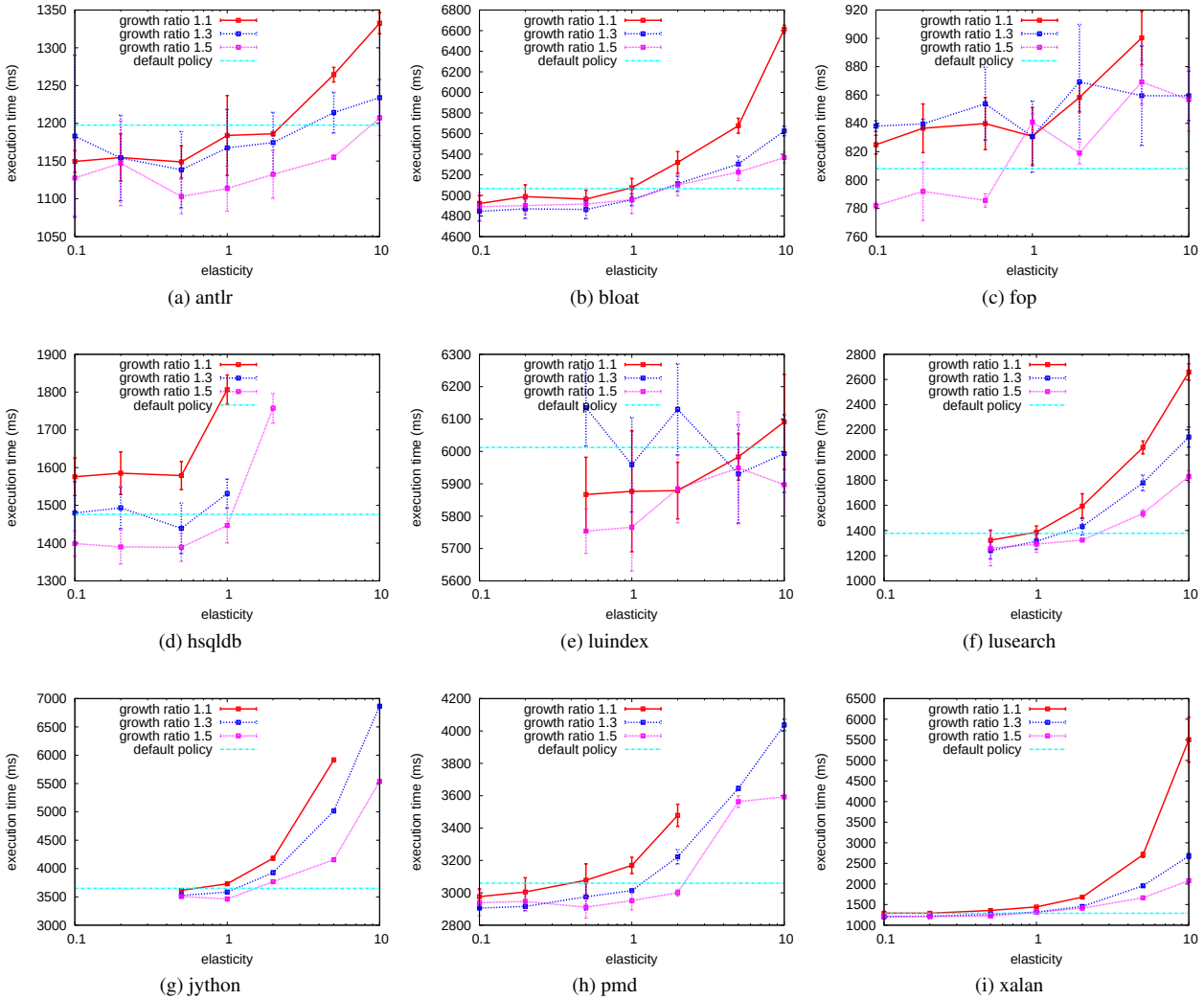


Figure 7: Execution times for selected DaCapo benchmarks using a full-heap collector, with variable sized heap based on elasticity heuristic, for several elasticity values and growth ratios

for heap expansion.) A larger heap growth ratio will cause the heap to expand more rapidly.

We compare the performance of our new elasticity heuristic for heap growth with the default Jikes RVM policy. In Figure 7, the horizontal line in each benchmark's graph shows the execution time with the default policy. For the default policy, we start with the same initial heap size of 25MB.

For most benchmarks, the graphs are fairly flat when $E < 1$. Above unit elasticity, the execution time increases sharply. This is because the heap growth is restricted, which causes more GCs to occur, which degrades the application execution time. For some benchmarks, such as jython and xalan, the default policy gives comparable performance with our new elasticity heuristic when $E < 1$. For other benchmarks, such as antlr and hsqldb, elasticity-based heap growth out-performs the default policy when $E < 1$, particularly for higher growth ratios.

Figure 8 shows the expanded heap sizes at the end of application execution. Some high-elasticity hsqldb configurations give `OutOfMemory` errors, since the heap grows too slowly to support the program's allocation schedule. It can be seen that low target elasticity and high growth ratio values lead to comparatively larger heap sizes, causing fewer GCs. Figure 8 also reports the final heap size with the default heap growth policy, shown as a horizontal line. In general, the default policy expands the heap to roughly the same size as when $1 \leq E \leq 2$, although this range is benchmark-specific, and also dependent on the growth ratio.

The data in Figure 8 explains why the default policy for antlr is *slower* than the elasticity-based policy when $E < 1$. In this range, the heap size with elasticity-based growth is significantly *larger* than with the default policy, for all growth ratios. Hence fewer GCs are incurred in the larger heaps, so overall application execution time is reduced. Basically, on an allocation curve, the elasticity-based policy picks a point to the *left* of the default policy—giving larger heap size, and better execution time. In contrast, consider the xalan benchmark, for which the elasticity-based policy does not outperform the default. The allocation curve for xalan in Figure 3(i) is *inelastic* for large heap sizes. So even though elasticity-based policy selects a larger heap size than the default policy, the number of GCs is generally not reduced.

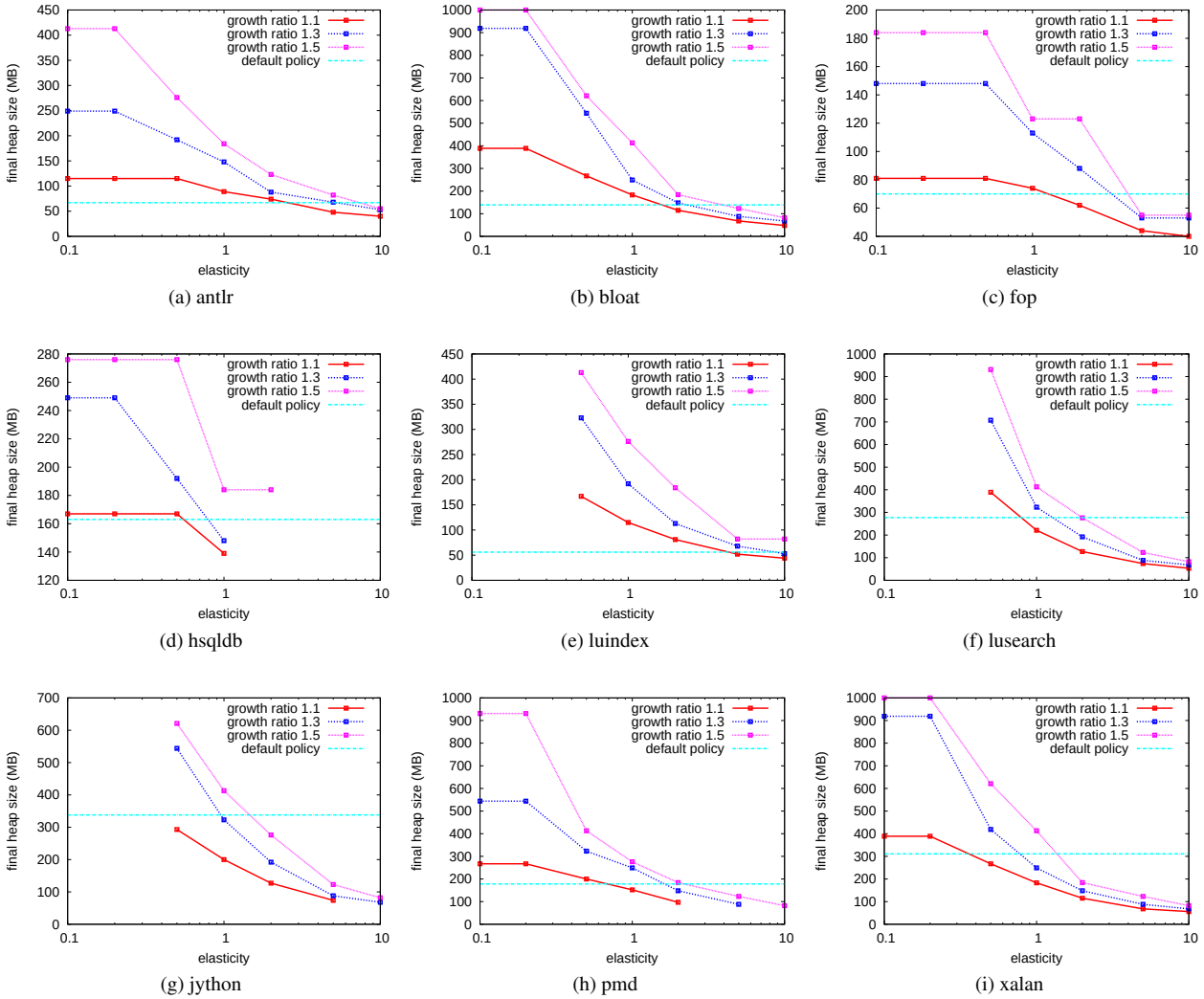


Figure 8: Final heap sizes for selected DaCapo benchmarks using a full-heap collector, with variable sized heap based on elasticity heuristic, for several elasticity values and growth ratios

In summary, the elasticity-based heap growth policy is more flexible than the default Jikes RVM heap growth policy, since our new heuristic can easily be tweaked to change the rate of heap growth. There is a range of elasticity values for which the heap growth is roughly equivalent to the default policy. However for very low E values, the heap expands more rapidly. For very high E values, the heap expands more slowly. This user control of the heap growth rate may be particularly useful in a server environment where multiple applications / VM instances are running concurrently and competing for physical memory.

As always, there is a *tradeoff* between heap size and execution time. A larger heap has fewer GCs, which improves the application execution time. However the allocation curves show that there are *diminishing returns* for increasing the heap size. Consider the highly inelastic part of an allocation curve as the number of GCs tends to 0. The heap size grows dramatically here, but the number of GCs (and hence the execution time) hardly varies.

Our new elasticity-based heuristic allows us to find a happy medium in the tradeoff between heap size and execution time. Ideally, the heap size should be somewhere near the *knee* of the allocation curve. To the left of the knee, application execution time does not improve significantly. To the right of the knee, application execution time is comparatively slow, due to the large number of GCs.

7. Conclusions

This paper has drawn a novel analogy between microeconomics and memory management. In particular, we have introduced the *allocation curve* as a graphical model for an application's interaction with the GC. This model is derived from the *demand curve* in microeconomics. There are some convincing parallels between demand curves and allocation curves, such as movements and shifts.

We have generated empirical allocation curves for a set of standard Java benchmarks, and discussed their properties. We define *allocation elasticity* and use it to give a new perspective on controlling heap growth in Jikes RVM. Elasticity-based heap growth is easier to configure and evaluate than the default lookup-table implementation. In the standard version of Jikes RVM, the only mechanisms for limiting the heap growth are: (i) to specify a max-

imum allowable heap size; or (ii) to alter the coefficients in the `HeapGrowthManager` matrix, and recompile the VM. Our elasticity alternative is potentially helpful as a middle-ground between these two extremes, since it allows the users to specify how quickly they want the VM to respond to changes in application allocation behaviour. So far, we have only considered how to use elasticity to *increase* the heap size. However we expect it would be a straightforward extension to handle heap shrinkage in the same framework.

With regard to future work in this area, we have only just scratched the surface of microeconomic theory. Other interesting concepts that may have analogues for memory management include supply curves, monopolies and competition.

References

[1] R. Alonso and A.W. Appel. Advisor for flexible working sets. In *ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems. Boulder, May 22–25*, 153–162. ACM Press, 1990.

[2] B. Alpern, C.R. Attanasio, J.J. Barton, M.G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S.J. Fink, D. Grove, M. Hind, S.F. Hummel, D. Lieber, V. Litvinov, M.F. Mergen, T. Ngo, J.R. Russell, V. Sarkar, M.J. Serrano, J.C. Shepherd, S.E. Smith, V.C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

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

[4] H.G. Baker. Thermodynamics of garbage collection. In *International Workshop on Memory Management*, Lecture Notes in Computer Science 637, 1992. Springer.

[5] H.G. Baker. Thermodynamics and garbage collection. *ACM SIGPLAN Notices*, 29(4):58–63, 1994.

[6] 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 *Intl. Conf on Object-Oriented Programming, Systems, Languages, and Applications*, 169–190, 2006.

[7] S.M. Blackburn, R. Garner, C. Hoffman, A.M. Khan, K.S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, S.Z. Guyer, M. Hirzel, A.L. Hosking, M. Jump, H. Lee, J.E.B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiederman. The DaCapo benchmarks: Java benchmarking development and analysis (extended version). Technical report, The DaCapo Group, 2006.

[8] S.M. Blackburn, P. Cheng, and K.S. McKinley. Myths and realities: The performance impact of garbage collection. In *Joint International Conference on Measurement and Modeling of Computer Systems*, 25–36, 2004.

[9] T. Brecht, E. Arjomandi, C. Li, and H. Pham. Controlling garbage collection and heap growth to reduce the execution time of Java applications. *ACM Transactions on Programming Languages and Systems*, 28(5), 2006.

[10] C. Chambers and A.L. Hosking, editors. *International Symposium on Memory Management*, 2000.

[11] W.D. Clinger and L.T. Hansen. Generational garbage collection and the radioactive decay model. In *Programming Language Design and Implementation*, 97–108, 1997.

[12] W.D. Clinger and F.V. Rojas. Linear combinations of radioactive decay models for generational garbage collection. *Science of Computer Programming*, 62(2):184–203, 2006.

[13] E. Cooper, S. Nettles, and I. Subramanian. Improving the performance of SML garbage collection using application-specific virtual memory management. In *Lisp and Functional Programming*, 43–52, 1992.

[14] S. Dieckmann and U. Hölzle. A study of the allocation behaviour of the SPECjvm98 Java benchmarks. In *European Conference on Object-Oriented Programming*, 92–115, 1999.

[15] R. Fitzgerald and D. Tarditi. The case for profile-directed selection of garbage collectors. In [10].

[16] Forecast: PC installed base, worldwide, 2004-2012. Gartner report, 2008.

[17] A. Georges, D. Buytaert, L. Eeckhout, and K. De Bosschere. Method-level phase behavior in Java workloads. In [25], 270–287.

[18] D. Grossman. The transactional memory / garbage collection analogy. In *Object-Oriented Programming, Systems, Languages, and Applications*, 695–706, 2007.

[19] C. Grzegorczyk, S. Soman, C. Krintz, and R. Wolski. Isla Vista heap sizing: Using feedback to avoid paging. In *International Symposium on Code Generation and Optimization*, 325–340. 2007.

[20] Ma. Hertz, J. Bard, S. Kane, E. Keudel, T. Bai, K. Kelsey, and C. Ding. Waste not, want not — resource-based garbage collection in a shared environment. Technical Report TR–951, The University of Rochester, 2009.

[21] X. Huang, S.M. Blackburn, K.S. McKinley, J.E.B. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: Improving program locality. In [25], 69–80.

[22] R.E. Jones and C. Ryder. A study of Java object demographics. In *International Symposium on Memory Management*, 121–130, 2008. ACM Press.

[23] R.E. Jones, and R. Lins. *Garbage Collection*. Wiley, 1996.

[24] T. Mann, M. Deters, R. LeGrand, and R.K. Cytron. Static determination of allocation rates to support real-time garbage collection. In *Languages, compilers, and tools for embedded systems*, 193–202, 2005.

[25] Object-Oriented Programming, Systems, Languages, and Applications, 2004.

[26] J. Singer, G. Brown, I. Watson, and J. Cavazos. Intelligent selection of application-specific garbage collectors. In *International Symposium on Memory Management*, 91–102, 2007. ACM Press.

[27] S. Soman, C. Krintz, and D. Bacon. Dynamic selection of application-specific garbage collectors. Technical Report 2004–09, UCSB, 2004.

[28] D. Stefanović, K.S. McKinley, and J.E.B. Moss. On models for object lifetime distributions. In [10], 137–142.

[29] D. Stefanović and J.E.B. Moss. Characterisation of object behaviour in Standard ML of New Jersey. In *Lisp and Functional Programming*, 43–54, 1994.

[30] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, 157–167, 1984.

[31] D. Vengerov. Modeling, analysis and throughput optimization of a generational garbage collector. In *International Symposium on Memory Management*, 1–9, 2009. ACM Press.

[32] P.R. Wilson, M.S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *International Workshop on Memory Management*, Lecture Notes in Computer Science 986, 1995. Springer.

[33] T. Yang, E.D. Berger, M. Hertz, S.F. Kaplan, and J.E.B. Moss. Autonomic heap sizing: Taking real memory into account. In *International Symposium on Memory Management*, 61–72, 2004. ACM Press.

[34] T. Yang, E.D. Berger, S.F. Kaplan, and J.E.B. Moss. CRAMM: Virtual memory support for garbage-collected applications. In *Operating System Design and Implementation*. 2006.

[35] C. Zhang, K. Kelsey, X. Shen, C. Ding, M. Hertz, and M. Ogihara. Program-level adaptive memory management. In *International Symposium on Memory Management*, 174–183, 2006. ACM Press.