We are all Economists Now:
Economic Utility for Multiple Heap Sizing

Callum Cameron   Jeremy Singer

University of Glasgow

firstname.lastname@glasgow.ac.uk

Abstract

Multiple virtual machine (VM) workloads are increasingly common, given the growth of managed enterprise application systems and consolidated virtual servers. Until now, there has been no principled approach to partitioning memory resource between multiple co-located VMs. In this paper, we develop a general framework for multi-VM heap sizing, based on the principle of utility maximization from microeconomic theory. We apply utility maximization to static heap sizing, and obtain performance improvements slightly better than current best-practice static heap sizing, and comparable with HotSpot ergonomics (current best-practice dynamic heap sizing). The major advantage of our approach is its simplicity and predictable resource utilization.

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

Keywords virtual machines, microeconomics, resource allocation

1. Introduction

Multiprogram workloads consisting of isolated virtual machine (VM) instances are now commonplace. Each VM hosts a single managed application. Typical scenarios include enterprise application platforms and consolidated virtual servers. An individual VM's heap size can vary with time, based on application heap mutation and garbage collection (GC) behavior. At any particular point in time, a VM has a minimum heap size required in order to make progress. Heap sizes above this minimum value enable more rapid progress, since less GC activity is required. For this reason, VM heap sizing has a significant impact on managed application performance.

At present, explicit heap size control is generally left to experts. For instance, Shirazi provides informal guidelines [20] for heap size tuning. He advocates a trial-and-error approach to find the best setting for an application, based on application-specific performance metrics. Due to the complexity and effort involved in tuning, most Java VM (JVM) users simply retain the default system

settings. On OpenJDK versions 1.6 and 1.7 the default maximum heap size is 25% of the physical RAM in the system [14]. For a managed application running on such a VM, the heap size is not permitted to exceed this hard limit.

However neither tuning nor the default heap sizing approach effectively addresses the situation when multiple VMs are running concurrently. One possibility is to keep the default VM maximum heap size, then abdicate responsibility to the underlying OS mechanisms to manage paging and resident set sizes for each VM process. The key problem is that most commercially available VMs do not communicate with the OS to ensure sensible heap sizing to avoid excessive paging problems, i.e. the OS does not know how much or which region of a VM heap is actually free memory.

Recent research into paging awareness for VMs generally focuses on avoidance of page faults via anticipative forced GC to reduce the working set size for a VM, e.g. [13] or by preventing the garbage collector from chasing pointers into memory that is swapped out, e.g. [12]. However these approaches do not explicitly aim to *maximize throughput* across all applications, nor do they address *fairness* in memory distribution between VMs.

In this paper, we present a general theoretical framework for multi-VM heap sizing, based on the concept of *utility maximization* in microeconomic theory. Our technique enables the effective partitioning of system memory between multiple VMs that are executing concurrently. Given (a) an overall budget for available system memory, (b) a standard measure of utility for managed application execution, and (c) some profiling information about each managed application, then we can divide the memory resources *fairly* between concurrently executing VM instances so as to *maximize the overall utility of the system*.

Our results show that this utility-based approach to heap sizing works well in practice. Our new approach performs at least as well as two existing heap sizing practices: (1) divide memory equally between VMs, and (2) allow the VM's default dynamic sizing policy to manage each VM independently while respecting a total memory budget. In terms of overall system throughput, our approach regularly outperforms (1) and sometimes outperforms (2) for a range of benchmark-based VM workloads.

1.1 Contributions

This paper makes the following three contributions:

1. We introduce a novel and general theoretical basis for multi-VM heap sizing, based on utility maximization in a microeconomic framework.

2. We demonstrate how to solve the multi-VM heap sizing optimization problem analytically (for two VMs) and numerically (for an arbitrary number of VMs).

3. We provide an empirical study to show that our approach improves throughput slightly for standard VM workloads (i.e. Da-

Capo running on OpenJDK) in relation to existing best practice. Our approach is simple and has a formal analytical model.

2. Microeconomic Theory

Consumer theory [18] describes how spending should be split between commodities to maximize overall 'happiness'. The same theory can be applied to heap sizing in a system of concurrent VMs. How should memory be partitioned between the VMs to maximize overall throughput?

2.1 Utility and Throughput

A consumer's 'happiness' (formally, *utility*) depends on how much of each commodity is consumed. For a single commodity, this is modeled by a *utility function*, $U(x)$. Utility functions have two general features. The more of the commodity is consumed, the higher the utility[1]. However, the more the consumer already has, the smaller the gain from consuming an additional unit. This is the property of *diminishing returns*. More formally:

P1: $U(x)$ is strictly increasing.

P2: $\frac{dU}{dx}$ is strictly decreasing, but always positive.

Intuitively, we expect managed applications to behave in a similar way. The larger the heap, the less time spent in GC, and the higher the 'throughput'. 'Throughput' is fully defined in Section 3.2; for the moment it can be taken as equivalent to utility. As Figure 1 shows, at least some programs behave in this way.
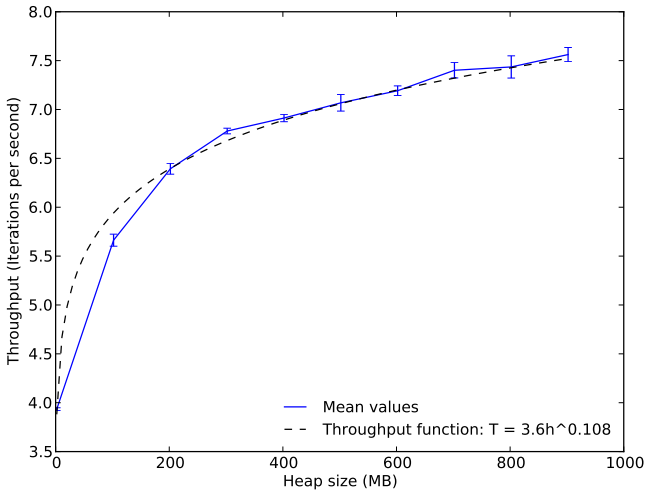

Figure 1: Throughput of the lusearch benchmark with small input and four threads. Each point denotes a looped execution of lusearch for 20 minutes with a fixed heap size. This program behaves as the utility model suggests.

2.2 Combining Utilities

Consumers buy many commodities. The individual utility functions are combined to give a total utility function in terms of the quantities of each commodity. For two commodities, this is $U(x, y)$. Sets of (x, y) points which give the same total utility are *isoutility curves* (also called *indifference curves*), as shown by the contour lines in Figure 2. An isoutility curve represents all the combinations of commodities which make a consumer equally happy.

---

[1] For some commodities (such as hamburgers), utility decreases after a certain point. We do not consider this possibility at this stage; see Section 6.
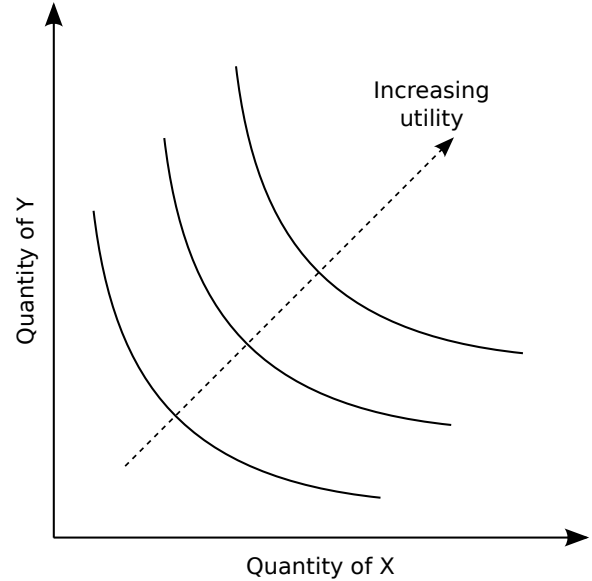

Figure 2: Combined utility of two commodities. The contour lines are isoutility curves.

The isoutility curves in Figure 2 are convex to the origin because of the conditions P1 and P2 imposed on individual utility functions, and the way the utility functions are combined. To ensure that isoutility curves have this shape, the overall utility function must also respect P1 and P2. For instance, multiplying the individual utilities is acceptable, whereas negating one of them before multiplication is not.

Although Figure 2 shows two commodities, the same properties hold for many commodities. The isoutility surfaces (for three commodities) or hypersurfaces (for more than three commodities) are also convex to the origin.

2.3 The Budget Constraint

A consumer's goal is to maximize total utility. However, there are limits to consumption. Commodities cost money, and the consumer only has a certain amount to spend. The consumer is constrained by a *budget*. Figure 3 shows a budget line superimposed on the isoutility curves from Figure 2. The axes represent the total expenditure on a commodity, i.e. the quantity consumed multiplied by the price of a unit. The budget line divides the utility space into two regions: *feasible* (under budget or exactly on budget), and *infeasible* (over budget). The budget line is straight because all combinations which are exactly on budget satisfy the equation $X \cdot P_X + Y \cdot P_Y = B$, by definition, where B is the budget, X and Y are the quantities of two commodities, and P_X and P_Y are the unit prices of the two commodities. This generalizes to many commodities, where the budget plane or hyperplane is described by the equation $\sum_{i=1}^{N} X_i \cdot P_{X_i} = B$.

The consumer's goal is to maximize utility while being within budget[2]. This amounts to finding the quantities of each commodity that maximize the total utility function within the budget constraint. However, because the isoutility lines are convex to the origin, the point of maximum utility occurs on the budget line, where the budget line touches the highest isoutility curve which it touches. At this point, the budget line is tangential to the isoutility curve,

---

[2] The consumer is indifferent to the amount of money spent (as long as it is within budget). Having money 'left over for future use' is not a consideration. Only utility matters.
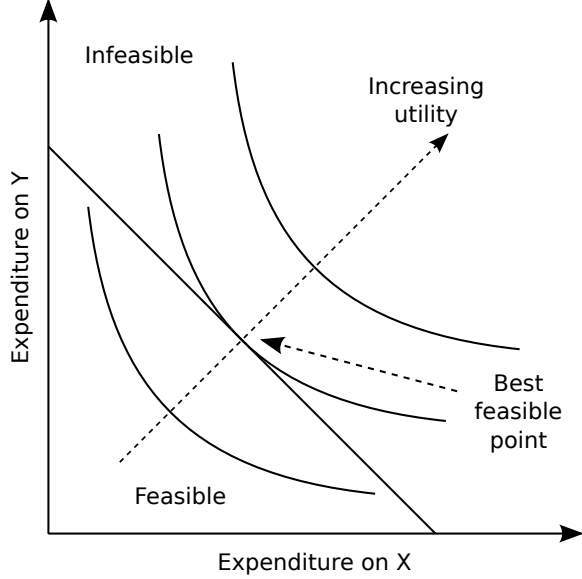
Figure 3: Combined utility of two commodities with a budget line. The best feasible point is on the isoutility curve where the budget line is tangential to the curve.

so there is exactly one best point. Thus we only need to consider points on the budget line, which has one less dimension than the total utility space. This also generalizes to the many-commodity case.

We must impose one more constraint on the total utility function for this to be valid. A consumer cannot *specialize*; at least a little bit of everything must be bought. This amounts to saying that the isoutility curves must never touch the axes.

2.4 Application to Heap Sizing

Heap size h and throughput T are analogous to quantity consumed and utility, respectively. Various models can be used which meet the criteria for a throughput function. One is $T(h) = ah^b$ where $0 < b < 1$ (which we call the 'root' model because it represents an 'n$^{\text{th}}$ root' function). Another is $T(h) = a \cdot ln(bh)$ (which we call the 'log' model). The constants in these models are program specific. Section 3.1 describes how to determine them for any particular program.

The 'no specialization' constraint follows from the problem definition. We want all VMs to progress, so all must be allocated at least some memory. We therefore combine throughputs by multiplication (rather than, say, addition) so that setting any of the allocations to zero gives a total throughput of zero. This gives the total throughput function $T(h_1, \ldots, h_N) = \prod_{i=1}^{N} T_i(h_i)$.

The budget constraint becomes the *memory budget*, M. This is the total amount of memory we want to partition between the VMs. In the simplest case, all megabytes are considered equal, so we have $\sum_{i=1}^{N} h_i = M$. By varying the 'value' of a megabyte for each VM, we can introduce *prioritization* (see Section 6).

Heap sizing introduces an additional constraint: the *minimum heap size* requirement. Each program has a minimum heap size below which it cannot run. At the point of maximum throughput, each VM must be allocated at least its minimum heap size.

2.5 Finding the Maximum Throughput

Once the constants in the individual throughput functions have been determined, and we have chosen a memory budget, how do we find the point of maximum throughput?

Some models permit an analytical solution, so we can derive a formula for the best point, for a particular number of VMs. For instance, the formula for the 'root' model with two VMs is derived as follows.

The total throughput function is $T(h_1, h_2) = T_1(h_1)T_2(h_2) = ah_1{}^b ch_2{}^d$, where the constants are taken from the individual throughput functions. The best point occurs on the budget line, so $h_1 + h_2 = M$. Each program must be allocated at least its minimum heap size, so $h_1 \geq p$ and $h_2 \geq q$, where p and q are the respective minimum heap sizes. On the budget line, $h_2 = M - h_1$. Substituting this into T gives a function of one variable, $T_{budget}(h_1) = ah_1{}^b c(M - h_1)^d$. To find the global maximum of T_{budget}, we set its derivative to zero, which gives the following result:

$$\max h_1 = \frac{bM}{b + d} \qquad (1)$$

Applying the minimum heap conditions, we get the following result for the best value of h_1:

$$\text{best } h_1 = \begin{cases} \max h_1 & \text{if } p \leq \max h_1 \leq M - q \\ p & \text{if } \max h_1 < p \\ M - q & \text{if } \max h_1 > M - q \end{cases} \qquad (2)$$

In all cases, best $h_2 = M - $ best h_1.

For this model, a similar formula can be derived for any given number of VMs. Once derived, the formula can be evaluated in constant time.

Other models, such as the 'log' model, do not permit an analytical solution. In this case, the best point must be found using numerical optimization, which is considerably slower.

Because we only need to consider points on the budget line, we reduce the number of dimensions in the problem by one, which makes finding an analytical solution possible in those models that permit it. (Without this constraint, the analytical solution would be underconstrained.) For those models that require numerical optimization, it reduces the number of dimensions of the objective function. For numerical optimization, we use the L-BFGS-B algorithm [7] provided by the SciPy library [19]. In this case, reducing the number of dimensions reduces the complexity of the optimization.

3. Experimental Method

Economic theory predicts the best partition of memory between concurrently executing VMs. But do these predictions match up to reality? To test the theory, we must measure the total throughput of concurrently executing VMs, at many heap size allocations, and compare the results to the predictions made by the theory.

3.1 Individual Throughput Functions

The throughput models introduced in Section 2.4 contain unknown constants. These constants characterize a specific program running on a specific machine. We must find the constants for each program individually, before we can run many programs concurrently.

If we measure a program's throughput at many heap sizes, we can use simple linear regression to fit the model to the data, and find the constants. However, we must now define throughput, and how we can measure it.

3.2 Measuring Individual Throughput

Managed applications make faster progress the less time is spent in GC. Throughput is the rate of making progress. However, this is

difficult to measure directly without instrumenting the VM. A convenient proxy for throughput is the number of completed iterations of a program per second. For deterministic programs, a single iteration represents a fixed amount of progress. Changing the heap size changes the number of completed iterations per second. Completed iterations per second can be measured by running a program in a loop for a fixed time, and dividing the total completed iterations by the total length of the run.

The advantage of this approach is that it can be used with an unmodified VM, and unmodified programs, simply by measuring execution time. The obvious disadvantage is that it only works for deterministic programs.

To get a single *(heap size, throughput)* pair from a run, the heap size must be fixed for the duration of the run. This introduces another limitation – the results are, in some sense, averages over the whole run. They do not account for changes in program behavior, and the average obtained may not actually be the 'best' value at any given time. All these limitations are discussed in Section 6.

The DaCapo benchmark suite [6] provides several deterministic Java programs ideally suited to this approach. These are open-source programs wrapped in a benchmarking harness. The harness runs the programs in a loop, and emits diagnostic messages after each iteration. By running the harness for an extremely large number of iterations, we can terminate the harness at the end of the fixed measurement time, and calculate how many iterations were completed. Additionally, the HotSpot JVM provides an option to fix the heap size at invocation, which overrides the usual ergonomics behavior. This configuration is used throughout.

3.3 Measuring Combined Throughput

Once we have the individual throughput functions, we can use the theory in Section 2.5 to predict the best partition of memory for any combination of programs. To test the prediction, we run the programs concurrently and measure the actual throughput they achieve.

The programs are run concurrently at many heap size allocations, including the best allocation predicted by theory, several allocations on the budget line, and a selection of allocations elsewhere in the feasible region of the throughput space. For each allocation, the throughput of each program is measured over a fixed time, and then all are combined as defined by the throughput model. In this way, a map of the combination's behavior throughout the throughput space is built up.

There are several interesting points in a throughput space.

- The *best measured throughput*, at the allocation where the highest throughput actually occurs.

- The *best predicted throughput*, at the allocation where the theory predicts the best throughput *should* occur.

- The measured throughput at the allocation where the theory predicts the best throughput should occur. We call this the *observed throughput*, because it is the throughput that will be observed by a user who has set the heap sizes based on the predictions. It is therefore the most important point for comparison with the other points.

The absolute values of the throughputs at these points are not interesting, because they are program specific. Far more interesting are the relative values of the points. We can define several useful comparisons.

- The *prediction accuracy*: the ratio of the best measured throughput to the best predicted throughput. This shows how well the theory predicts the *value* of the best throughput. Values closer to one are better. Values less than one mean the theory overes-

timates the best throughput, and values greater than one mean the theory underestimates.

- The *Euclidean distance* between the heap size allocation at the best measured throughput, and the heap size allocation at the observed and best predicted throughputs. This shows how well the theory predicts *where* the best throughput will occur, and smaller values are better. Note that this metric considers only the heap sizes, not the values of the throughput (which are considered by the previous metric), and is therefore a Euclidean distance in one less dimension than the overall space.

These two metrics show how closely the measured throughput surface matches the shape of the predicted throughput surface. However, there is a third, more important, metric:

- The *practical accuracy*: the ratio of the observed throughput to the best measured throughput. This is especially important because a user will not sample the throughput space as we have done, and so will not know where the best throughput would occur. The user only cares that the theory recommends heap sizes that give a throughput as close to the maximum as possible, which is what this metric measures. This metric will always be less than or equal to one, with higher values being better.

3.4 Choosing a Memory Budget

Because the theory assumes that throughput always increases with heap size, it ignores the effect of paging when the heap is much larger than physical memory. If paging becomes significant, throughput might decrease beyond a certain point, but the theory does not take account of this (see Section 6). Therefore to ensure that the theory's assumptions remain valid, a total memory budget should be chosen which is no more than the available physical memory.

4. Empirical Evaluation

All experiments used programs from the DaCapo benchmark suite version 9.12 'Bach' [6], running in the HotSpot JVM from Open-JDK v1.7.0.45[3]. The experiments were carried out on machines with quad-core Intel Core i5 processors running at 3.2 GHz, with 4 GB of memory, running Linux 2.6.32 x86.64. Measurement scripts and the full results set can be found in our repository at `http://www.anyscale.org/icooolps14.tar.xz`.

4.1 Individual Throughput Functions

Individual throughput functions were measured as described in Section 3.2, for many DaCapo benchmarks, with different input sizes and thread counts. Figures 1, 4, and 5 show examples of programs that fit the 'root' model with varying degrees of accuracy. Each point is the arithmetic mean of the completed iterations per second of five 20-minute runs. The error bars show ± one standard deviation.

4.2 Combined Throughput

The individual throughput functions were used to predict the best allocation for various benchmark combinations. These combinations were then measured at many heap size allocations. Figure 6 shows the measured throughput space for several two-program combinations. Figure 7 shows the predicted throughput for each of the points in Figure 6a, for comparison.

---

[3] During the course of our experiments, the OpenJDK package on our centrally managed Linux systems was upgraded from v1.7.0.45 to v1.7.0.51. We repeated many of the measurements and observed that there was no noticeable performance difference between these two minor release versions.
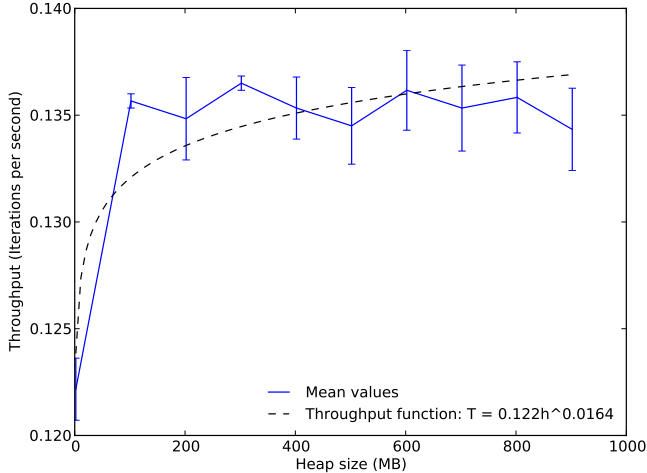
Figure 4: Throughput of the sunflow benchmark with default input and one thread. This program almost behaves as the model suggests.
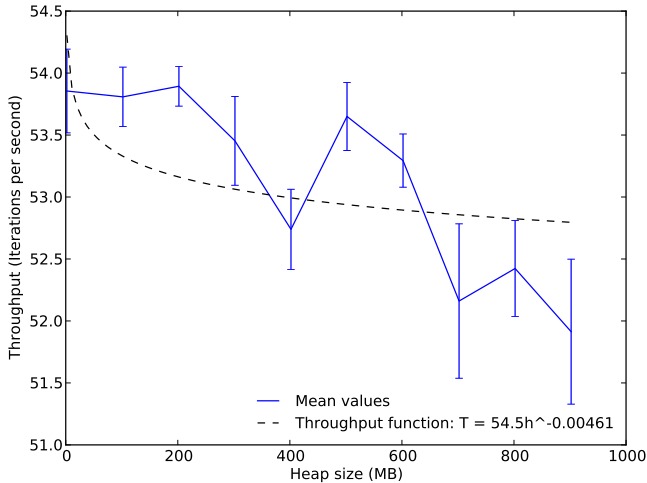

Figure 5: Throughput of the pmd benchmark with extra-small input and one thread. This program does not fit the model, because b is negative.

Figure 6a shows a typical case. The best measured throughput lies close to the budget line, as expected. The Euclidean distance to the best predicted throughput is quite small. The prediction accuracy is 0.91, and the more important practical accuracy is 0.96, i.e. the throughput observed using the theory to choose heap sizes is within 96% of the best possible throughput. The measured isothroughput lines have approximately the shape that the theory predicts, suggesting that the theory predicts the shape of the throughput space quite well in this case.

Figure 6b shows an exceptional case where the theory predicts the location of the best measured throughput exactly. The Euclidean distance is 0, and the practical accuracy is 1. However, the isothroughput lines are uneven, suggesting that in this case the theory does not predict the throughput of other points in the space very well.

Figure 6c shows a surprising result. The theory performs worst for combinations where two instances of the same benchmark are run together. On average, these combinations have a larger Eu-

clidean distance and a lower practical accuracy than combinations of different benchmarks. The highest measured throughput tends to be towards the edge of the space, with one instance near its minimum heap size. We would expect the throughput to be highest when memory is partitioned equally between the benchmarks, but this is not the case. A preliminary investigation using `strace` shows that when memory is partitioned equally, the total time spent executing system calls is significantly higher in some cases than when the heap sizes differ. We surmise that when two identical benchmarks with identical heap sizes are executing together, they contend for shared system resources more than out-of-phase benchmark instances with different heap sizes.
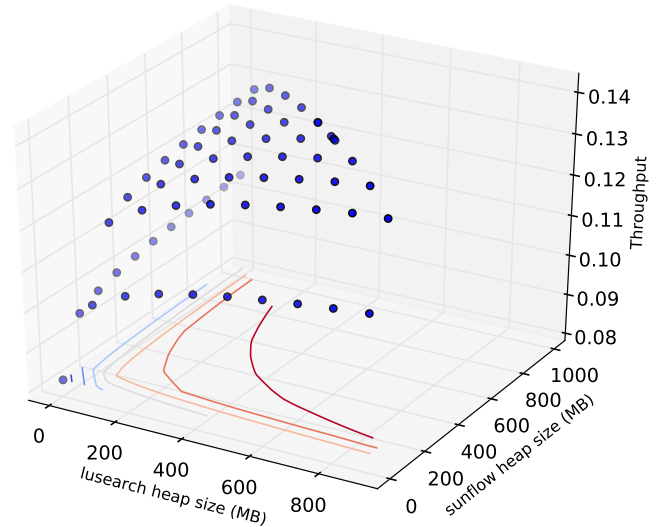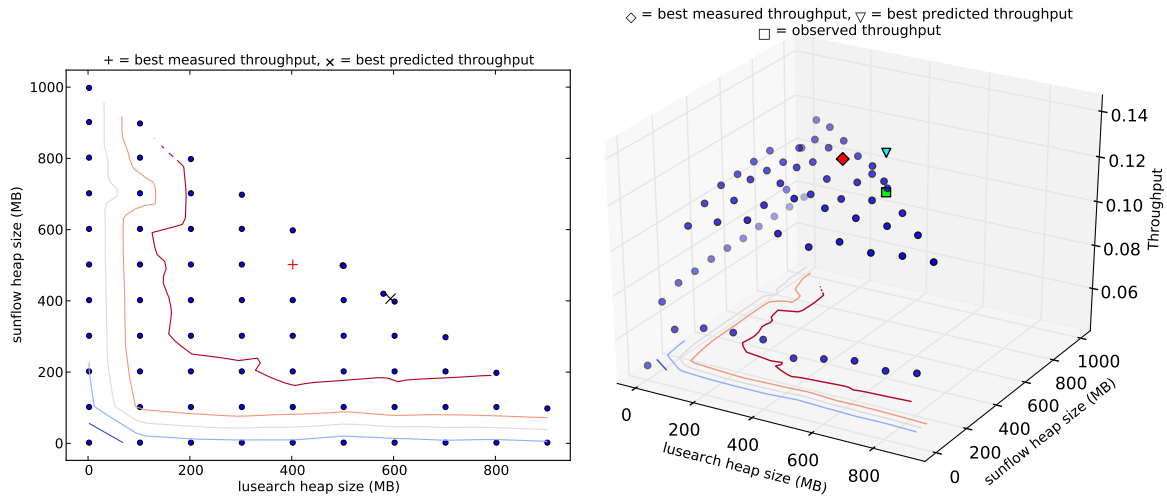

Figure 7: Predicted combined throughput of lusearch with default input and 1 thread, and sunflow with default input and 3 threads, using the 'root' model. The memory budget is 1 GB.

Figure 8 shows the combined throughput for a three-program combination. Since we are now attempting to visualize a four-dimensional dataset, the three heap sizes are shown on the axes, and the throughput of each allocation is represented by its color, from black for lower throughputs, to white for higher. The best measured and best predicted points both lie on the budget plane, as expected, and they have a similar Euclidean distance to the two-program cases. The practical accuracy is also in a similar range.

The results of many combinations[4] are shown in Table 1, in terms of the metrics defined in Section 3.3. In each combination, the total number of benchmark threads has been kept less than the number of cores (four, in this case), so that these threads will get as much CPU time as possible. The errors in the prediction accuracy and practical accuracy are standard deviations, propagated through calculation from the standard deviations of the measured throughputs. The errors in the average Euclidean distance are the standard deviations of the absolute Euclidean distance values. Combinations where several instances of the same benchmark are run together have, on average, a higher Euclidean distance and lower practical accuracy than combinations of different benchmarks. When running different benchmarks together, the theory predicts heap sizes which give a throughput that is, on average, 97% of the best possible throughput.

---

[4] There was not enough time to exhaustively measure all possible combinations. These combinations were chosen based on how well the individual throughput functions fit the model, and so that the total number of threads was never more than the number of cores on the measuring machines.

(a) lusearch with default input and 1 thread, and sunflow with default input and 3 threads.
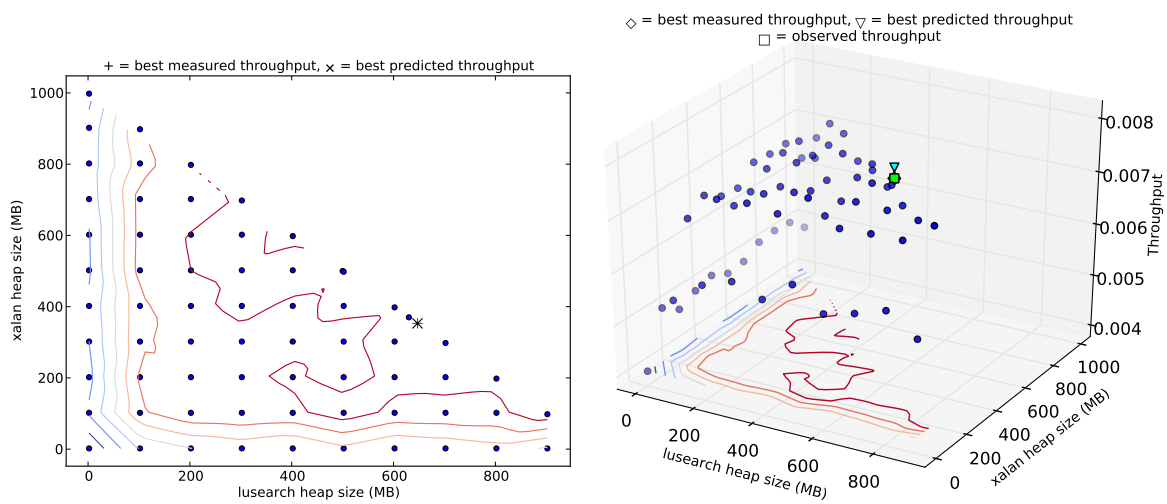
(b) lusearch with large input and 1 thread, and xalan with large input and 1 thread.
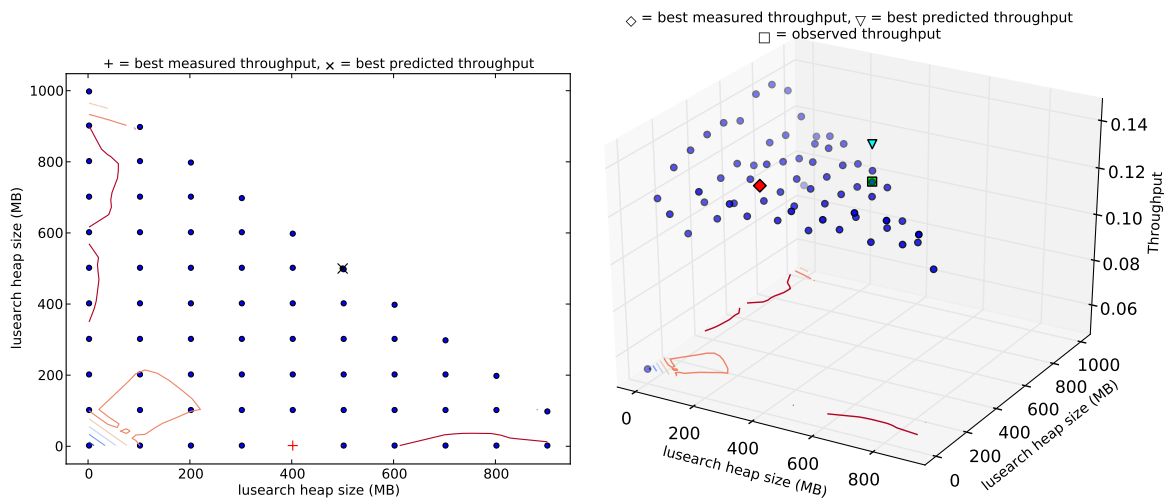
(c) Two instances of lusearch with default input and 1 thread.

Figure 6: Combined throughput of two-benchmark combinations, using the 'root' model. The memory budget is 1 GB. Each left graph is the view from above of the right graph. Contour lines show isothroughput.

Benchmarks (name, input size, threads)	Euclidean distance	Prediction accuracy	Practical accuracy
avrora large 1, avrora large 1	411.54	0.56 ± 0.01	0.99 ± 0.01
avrora large 1, lusearch large 1	229.10	0.95 ± 0.01	0.99 ± 0.02
avrora large 1, sunflow large 1	351.00	0.95 ± 0.02	0.95 ± 0.02
avrora large 1, sunflow large 3	35.36	0.60 ± 0.01	0.96 ± 0.02
lusearch default 1, lusearch default 1	507.55	1.01 ± 0.01	0.88 ± 0.02
lusearch default 1, sunflow default 1, xalan default 1	233.70	0.94 ± 0.03	0.94 ± 0.03
lusearch default 1, sunflow default 3	214.66	0.91 ± 0.01	0.96 ± 0.02
lusearch large 1, lusearch large 1	640.01	1.01 ± 0.01	0.92 ± 0.02
lusearch large 1, sunflow large 3	345.77	0.88 ± 0.01	0.97 ± 0.03
lusearch large 1, xalan large 1	0.00	0.98 ± 0.02	1.00 ± 0.03
pmd large 1, avrora large 1	332.23	0.72 ± 0.01	0.98 ± 0.02
pmd large 1, lusearch large 1	18.38	0.82 ± 0.02	1.00 ± 0.02
pmd large 1, pmd large 1	97.58	0.66 ± 0.00	0.99 ± 0.02
pmd large 1, xalan large 1	251.51	0.80 ± 0.00	0.98 ± 0.02
sunflow default 2, sunflow default 2	508.34	0.99 ± 0.01	0.97 ± 0.02
sunflow large 1, pmd large 1	196.58	0.81 ± 0.02	0.98 ± 0.03
sunflow large 2, sunflow large 2	640.01	1.02 ± 0.03	0.95 ± 0.02
xalan large 1, xalan large 1	313.70	0.97 ± 0.01	0.97 ± 0.04
xalan large 2, sunflow large 2	188.09	0.98 ± 0.02	0.98 ± 0.02
Average, all (19 combinations)	290.27 ± 185.93	0.87 ± 0.00	0.97 ± 0.01
Average, same-benchmark combinations (7 combinations)	445.53 ± 178.50	0.89 ± 0.01	0.95 ± 0.01
Average, different-benchmark combinations (12 combinations)	199.70 ± 117.84	0.86 ± 0.00	0.97 ± 0.01

Table 1: Comparison between predicted and measured throughputs for various benchmark combinations, using the 'root' throughput model. Data for other models is similar and can be found in our repository. A prediction accuracy greater than 1 indicates that the best measured throughput is higher than the best predicted throughput; a prediction accuracy less than 1 indicates that the best measured throughput is lower than the best predicted throughput.
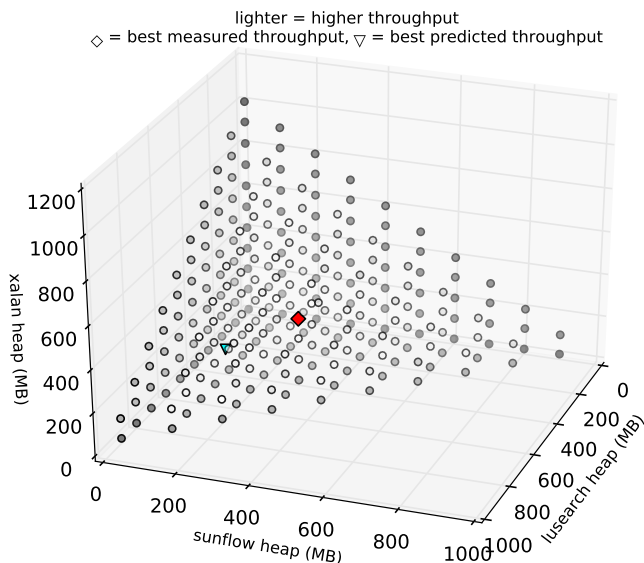

Figure 8: Combined throughput of lusearch, sunflow, and xalan, each with default input and 1 thread, using the 'root' model. The memory budget is 1 GB.

4.3 Comparison with Existing Mechanisms

We have examined the *accuracy* of the predictions made by the economic approach. However, it is also useful to compare the *performance* of the economic approach with other methods of partitioning memory among VMs.

In the *equal partition* approach, N VMs are each allocated $1/N$ of the memory budget. This does not take account of the characteristics of the programs, and is therefore the baseline that any more complex approach must beat. The equal partition is included in the sets of allocations measured in Figures 6 and 8.

Alternatively, the VM's default sizing policy (in this case, HotSpot ergonomics [24]) can be given full control over the heap size. This is the *unconstrained* approach. Rather than fixing the heap size at the beginning of a run, each VM will adjust its heap size throughout the run with no knowledge of the other VMs. To keep the comparison fair, we do not allow the VMs to be *fully* unconstrained. Instead, we use Linux control groups (cgroups) to impose a memory budget on the VMs, so that paging will be incurred if the heaps grow beyond this limit. To be worthwhile, our utility-based approach must give better performance than the existing default technique or we must demonstrate another improved non-functional characteristic, e.g. predictability or fairness.

The throughput of the unconstrained approach was measured for each of the benchmark combinations in Table 1. These measurements were taken in the same way as for the points in the economic throughput space.

Figure 9 shows the throughput of the equal partition, the best predicted throughput, and the observed throughput, relative to the throughput of the default unconstrained sizing policy. These values are averaged over all measured benchmark combinations, those where the benchmarks are the same, and those where the benchmarks are different. Several observations can be made.

- The default sizing policy is usually better than the equal partition. This is as expected, since the equal partition takes no account of program characteristics.

- The economic approach usually performs better than the equal partition.

- The observed throughput with the economic approach is usually at least as good as the default sizing policy, and sometimes slightly better. That is, the economic approach is as good as the default sizing policy *even though it fixes the heap sizes at startup*, and even with the limitations discussed in Section 6.
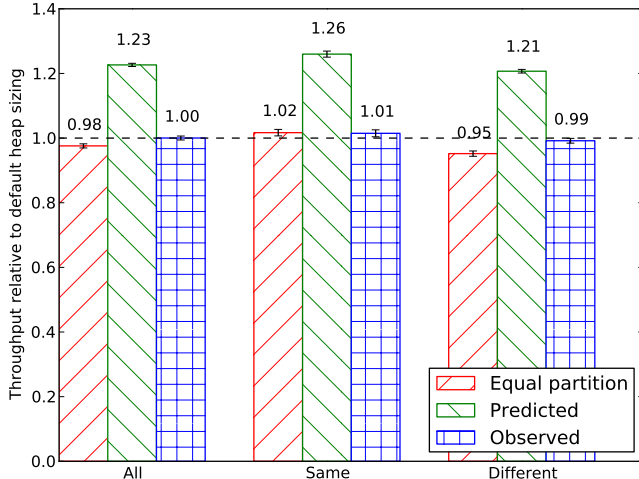
Figure 9: Throughput measurements for three heap sizing policies, relative to the default ergonomics approach. Higher is better. The policies are (1) static equal partitioning, (2) theoretical best predicted throughput *(unattainable)* and (3) observed throughput at best predicted point. We report arithmetic means over 'all' measured benchmark combinations from Table 1, then consider concurrent instances of the 'same' benchmark and of 'different' benchmarks.

We conjecture that if these limitations were to be overcome, the economic approach could perform better still.

5. Related Work

5.1 Application of Mathematical Frameworks

This section reviews how memory management researchers interpret garbage collection in the context of existing principled mathematical frameworks.

There is a long tradition of analyzing GC by means of analogy with well-understood physical systems. Baker [5] relates memory management to statistical thermodynamics. In this scheme, memory storage locations are mapped to the conserved energy in the system. Baker argues that mutators increase entropy, whereas collectors reduce entropy. In some sense, a garbage collector acts like a refrigerator. Baker [4] also considers a hypothetical case in which object lifetimes are distributed in the same way as radioactive half-life decay times. He proposes this model as a convenient mathematical abstraction, to show that the mark-cons ratio is the same for all generations with this kind of object lifetime distribution, even when the weak generational hypothesis is empirically valid, i.e. most objects die young.

Clinger and Hansen [8] explore this radioactive half-life analogy further. They present a meta-analysis to suggest that long-lived objects have lifetimes that are independent of age (i.e. half-life decay), whereas the majority of young objects die very quickly. They use this insight to construct a new collector that has two generations, with the old generation being collected using a non-predictive policy (i.e. it does not make assumptions about object lifetimes when collecting the old generation). Clinger and Rojas [9] show that many benchmarks' object lifetime distributions may be characterized by a linear combination of radioactive decay models with different half-lives and proportions. They give empirical evidence to show that such models are appropriate for many real-world observed programs / benchmarks.

As an over-simple generalization, it seems that GC models derived from physical systems deal well with *how* and *when* to collect, but these models do not provide particular assistance for *heap sizing*.

Since economics deals with the distribution of scarce resources among competing entities, it has long been considered a natural fit for resource allocation problems in distributed systems, e.g. [16, 27].

Singer et al. [22] appear to be the first to apply microeconomic principles to GC in managed runtime environments. Using an analogy with demand theory, they treat heap memory as a price and GC overhead as a consumable good. They show how this fits with simple supply and demand concepts such as the law of diminishing returns. The main limitations of this work are that it only applies to a single VM in isolation, and that there is no notion of throughput optimization in their formalism. Simão and Veiga [21] present a semi-formal description of an adaptive resource sharing scheme that has the explicit objective of maximizing *yield* across multiple VMs. However this preliminary work does not present a concrete implementation and is not explicit about the optimization technique used to maximize the yield metric.

5.2 Empirical Analysis

This section reviews quantitative characterizations of GC, generally involving regression analysis to fit an analytic equation to the observed GC behavior. For each paper, we summarize what is measured, the form of the equation(s) and the regression technique where relevant.

Stefanović et al. [23] consider mathematical models for object lifetimes and compare these models with observed behavior for 58 Smalltalk and Java programs. They find that the mathematical models do not agree entirely with observed behavior, but they generally follow a gamma distribution. This is consistent with the analogies of radioactive half-life in Section 5.1.

Hertz and Berger [11] report an empirical study to measure the runtime overhead of GC relative to explicit memory management. They observe that the *frequency* of GC is inversely proportional to the heap size. They further observe that the *relative time* spent in GC for an Appel-style generational collector is inversely proportional to the square of the heap size, h. They proceed to fit parameters for a simple quadratic equation of the form $\text{time}(h) = a/(b - h^2) + c$. Above three times the minimum heap size for an application, there is little difference between the overheads of explicit memory management and GC. In contrast, we study combined throughput across multiple applications as well as individual application throughput functions. Also, we combine characteristic equations for single VM workloads within a principled economic framework.

Tay et al. [25, 26] derive an equation to predict the number of page faults a managed application will incur during its execution. This depends on the application, the VM and the underlying system configuration. They rearrange the page fault equation to formulate a dynamic heap sizing rule which has four parameters. These parameters are not interpretable, but they must be empirically determined for an application running with a specific input on a specific platform. The parameter fitting requires offline calibration via linear regression or online calibration via sliding window averages. Our work is similar to theirs. However we impose a user-defined memory budget rather than a soft 'available memory' limitation. Also, we attempt to maximize combined throughput of multiple managed applications in a single equational framework, whereas their approach is based on local optimization for each individual VM.

White et al. [29] treat a single VM as a black-box process and use control-theoretic tuning to fit a proportional-integral-derivative

(PID) controller equation to the heap sizing mechanism of the memory manager. The tuning is a manual process requiring domain expertise. They control heap sizing via a user-specified throughput value. Our work improves on their approach since we handle multi-VM workloads and we automatically maximize throughput rather than requiring user-specified values.

Lengauer and Mössenböck [17] describe a black-box tuning approach to the HotSpot GC system. They simultaneously tune multiple parameters for a single VM (although not including the maximum heap size) using search-based techniques to maximize an individual application's throughput.

5.3 Resource Sharing Virtual Machines

This section reviews related work that deals with multiple virtualized workloads in a single system and shares resources between the concurrently executing VMs.

Hertz et al. [13] describe a scheme to optimize throughput for multiple distinct VMs executing heterogeneous workloads. They present an information-sharing approach, where all VMs provide dynamic metrics about page faults and resident set size to a central controller. The controller uses heuristics to determine when to force full-heap GCs for particular VMs to reduce overall system memory pressure, with the aim of avoiding page faults. This scheme operates orthogonally to heap sizing in that it only adds an extra reactive trigger for full-heap GC when paging is imminent.

Alonso and Appel describe an advisor service [1] which is a user-level daemon that dispenses heap sizing advice to concurrently executing SML/NJ runtimes. Each runtime registers with the daemon and provide dynamic metrics about its execution time and heap space requirements at each GC. In return, the daemon specifies how the runtime should resize its heap. The service aims to improve overall throughput by ensuring the working sets of all VMs fit into physical memory. This is similar to our work, except we perform static rather than dynamic sizing. We do not require VM instrumentation, only ahead-of-time profiling. Also, we use a principled economic framework, whereas they derive specialized equations for their model based on expert knowledge of the underlying GC algorithm, which may not be transferable.

The multitasking virtual machine (MVM) [15] hosts isolated container-based Java applications in a single managed runtime environment. Each application has its own local nursery space; however all applications share a system-wide global space for mature objects. The MVM uses a best-effort system to support fair resource allocation. The Resource Management Interface (RM) [10] is a small, customizable API that enables the enforcement of resource management policies in MVM. This framework enables resource limits to be placed on memory usage of individual isolates but RM does not specify how to divide resources between isolates. It is a mechanism for resource sharing, but not a high-level decision-making framework.

KaffeOS [2, 3] is another runtime system that supports Java application isolation and enforcement of resource consumption limits. This system supports per-application accounting for memory resources and GC overhead. Each application has a separate heap with its own decoupled GC. Each heap has a size limit, which is specified when the heap is created and not updated afterwards. There appears to be little dynamicity in heap sizing; a heap can grow to its limit, but must not exceed it.

Many resource sharing mechanisms are also applicable in system-level virtualization schemes such as Xen (e.g. [28]).

6. Limitations and Future Work

Our approach to heap sizing has several limitations.

Static Sizing: The best allocation is predicted statically. It is calculated before the programs run and fixed throughout the run.

This requires programs to be profiled ahead of time, to determine their individual throughput functions, which takes a long time. Finding each of the throughput functions in Figures 1, 4, and 5 took around sixteen CPU-hours. The constants in the throughput functions are machine-specific because the same program running on a faster machine will complete more iterations per second, so the profiling must be done on machines identical to the one used for running. Profiling is therefore a considerable practical burden.

This approach is likely to be applicable for cloud-based server farms, where individual jobs might recur frequently but the dynamic mix of concurrent jobs is unpredictable.

Statically calculated throughput functions are in a sense 'averages' over the whole run. They do not account for changes in program behavior or inputs. Better throughput might be achieved by monitoring the behavior at runtime, and periodically adjusting the throughput functions to match the current behavior.

Throughput Metric: Using *completed iterations* as a proxy for throughput is another limitation. The benefit of this metric is that it can be measured from outside the VM, but it only works for deterministic programs. For input-dependent programs (i.e. almost all interesting programs), the same input must be used for profiling and running, so the throughput functions are accurate. In practice this rules out all programs that interact with a user. Even for batch programs like the DaCapo benchmarks, some nondeterminism is introduced by the OS. Scheduling, contention for I/O devices, and so on, are all unpredictable influences on the length of an iteration. This may account for some of the difference between the predicted and measured throughputs. Short of being the only processes on the system, which is impractical in realistic environments, there is little that can be done about this.

To overcome these limitations, we propose to investigate a 'dynamic' approach. In this system, VMs will measure throughput internally, using more reliable metrics such as allocation rate or GC time. Each VM will periodically report its current heap size and throughput to an external process. Based on recent measurements, this process will periodically generate throughput functions, use them to predict the best allocation of memory at the moment, and send heap size recommendations to the VMs. It will remove the requirement for deterministic programs, and for profiling beforehand, making it much more useful in practice.

Paging: Regardless of whether a static or dynamic approach is used, there are limitations in the underlying theory which must be addressed. Section 2 defines individual throughput functions as strictly increasing. This is valid for programs that fit entirely in physical memory, but for large programs that require paging, allocating very large heaps may actually decrease throughput. This is because common garbage collection techniques violate locality of reference, and performing a collection requires much of the virtual address space to be paged in. The best allocation may no longer lie on the budget line, but could be anywhere in the feasible region of the throughput space.

Equal Priorities: Throughout this paper we have assumed that a megabyte of allocation has the same value to one process as to another. In economics, units of different commodities may have different prices, as described in Section 2.3. In the heap sizing case, we have set $P_i = 1$ for all values of i. By allowing different programs to have different values of P, we can change the relative priorities of the programs, and change the calculated memory allocation. This could be a useful enhancement both to the static and dynamic cases.

Workload Interference: The theory also combines throughputs on the assumption they are independent. This ignores the effects of contention and paging. To counter this, it might be useful to investigate the effect of the number of running processes on the

accuracy of predictions, and whether particular combinations of programs affect the results in any predictable way.

Additionally, we have only considered heap sizing, but the economic approach could be applied to the allocation of other resources, such as hardware threads, in a similar way. We have only applied the economic approach to the HotSpot VM, but in theory it could be applied to any VM, or even to a heterogeneous collection of VMs. All of these are directions for future investigation.

7. Conclusions

In this paper, we have applied principles of microeconomic utility to the problem of heap sizing for multi-VM workloads. We have shown empirically that our utility-based static resource allocation scheme performs better than current best-practice static schemes and is comparable with current best-practice dynamic schemes in most cases.

Our utility-based heap sizing approach is simple and offers predictable resource utilization. Given the growing significance of multiprogramming for virtualized systems, e.g. in the context of cloud computing frameworks, we expect that utility-based resource allocation schemes will become increasingly relevant.

In the future, we aim to devise a dynamic version of our utility-based heap sizing scheme. We will also consider utility-based allocation for other computational resources apart from heap memory.

Acknowledgments

References

[1] R. Alonso and A. W. Appel. An advisor for flexible working sets. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 153–162, 1990.

[2] G. Back and W. C. Hsieh. The KaffeOS Java runtime system. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27 (4):583–630, 2005.

[3] G. Back, W. C. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proceedings of the 4th Symposium on Operating System Design and Implementation (OSDI)*, 2000.

[4] H. G. Baker. Infant mortality and generational garbage collection. *ACM SIGPLAN Notices*, 28(4):55–57, 1993.

[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 *Proceedings of the ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM, 2006.

[7] R. H. Byrd, P. Lu, J. Nocedal, and C. Zhu. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific Computing*, 16(5):1190–1208, 1995.

[8] W. D. Clinger and L. T. Hansen. Generational garbage collection and the radioactive decay model. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 97–108, 1997.

[9] 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.

[10] G. Czajkowski, S. Hahn, G. Skinner, P. Soper, and C. Bryce. A resource management interface for the Java platform. *Software: Practice and Experience*, 35(2):123–157, 2005.

[11] M. Hertz and E. D. Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *Proceedings of the ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 313–326, 2005.

[12] M. Hertz, Y. Feng, and E. D. Berger. Garbage collection without paging. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 143–153, 2005.

[13] M. Hertz, S. Kane, E. Keudel, T. Bai, C. Ding, X. Gu, and J. E. Bard. Waste not, want not: resource-based garbage collection in a shared environment. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management (ISMM)*, pages 65–76, 2011.

[14] P. Hohensee. Reduce cap on maximum heap size?, 2011. http://mail.openjdk.java.net/pipermail/hotspot-dev/2011-November/004724.html, confirmed by inspection of source code file globals.hpp.

[15] M. Jordan, L. Daynès, G. Czajkowski, M. Jarzab, and C. Bryce. Scaling J2EE application servers with the Multi-tasking Virtual Machine. Technical Report SMLI TR-2004-135, 2004.

[16] J. F. Kurose and R. Simha. A microeconomic approach to optimal resource allocation in distributed computer systems. *IEEE Transactions on Computers*, 38(5):705–717, 1989.

[17] P. Lengauer and H. Mössenböck. The taming of the shrew: Increasing performance by automatic parameter tuning for Java garbage collectors. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, pages 111–122, 2014.

[18] G. S. Maddala and E. Miller. *Microeconomics: theory and applications*. McGraw-Hill, 1989. ISBN 0-07-039415-6.

[19] SciPy developers. Scipy, 2014. http://scipy.org/.

[20] J. Shirazi. *Java Performance Tuning*. O'Reilly, 2003.

[21] J. Simão and L. Veiga. VM economics for Java cloud computing: An adaptive and resource-aware Java runtime with quality-of-execution. In *Proceedings of the IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 723–728, 2012.

[22] J. Singer, R. E. Jones, G. Brown, and M. Luján. The economics of garbage collection. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management (ISMM)*, pages 103–112, 2010.

[23] D. Stefanović, K. S. McKinley, and J. E. B. Moss. On models for object lifetime distributions. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management (ISMM)*, pages 137–142, 2000.

[24] Sun. Garbage collector ergonomics, 2004. http://docs.oracle.com/javase/1.5.0/docs/guide/vm/gc-ergonomics.html.

[25] Y. C. Tay and X. Zong. A page fault equation for dynamic heap sizing. In *Proceedings of the 1st Joint WOSP/SIPEW International Conference on Performance Engineering*, pages 201–206, 2010.

[26] Y. C. Tay, X. Zong, and X. He. An equation-based heap sizing rule. *Performance Evaluation*, 70(11):948–964, 2013.

[27] C. A. Waldspurger, T. Hogg, B. A. Huberman, J. O. Kephart, and W. S. Stornetta. Spawn: A distributed computational economy. *IEEE Transactions on Software Engineering*, 18(2):103–117, 1992.

[28] J. Wan, L. T. Yang, Y. Li, X. Xu, and N. Xiong. An adaptive management mechanism for resource scheduling in multiple virtual machine system. In *Autonomic and Trusted Computing*, volume 6906 of *Lecture Notes in Computer Science*, pages 60–74, 2011.

[29] D. R. White, J. Singer, J. M. Aitken, and R. E. Jones. Control theory for principled heap sizing. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management (ISMM)*, pages 27–38, 2013.