Cameron, C., Singer, J., and Vengerov, D. (2015) The Judgment of Forseti: Economic Utility for Dynamic Heap Sizing of Multiple Runtimes. In: International Symposium on Memory Management, Portland, OR, USA, 14 June 2015, pp. 143-156. ISBN 9781450335898

Copyright © 2015 ACM

A copy can be downloaded for personal non-commercial research or study, without prior permission or charge

Content must not be changed in any way or reproduced in any format or medium without the formal permission of the copyright holder(s)

When referring to this work, full bibliographic details must be given

http://eprints.gla.ac.uk/105894/

Deposited on: 18 June 2015

Enlighten – Research publications by members of the University of Glasgow
http://eprints.gla.ac.uk

The Judgment of Forseti: Economic Utility for Dynamic Heap Sizing of Multiple Runtimes

Callum Cameron Jeremy Singer
University of Glasgow, UK
cjcameron7@gmail.com
jeremy.singer@glasgow.ac.uk

David Vengerov
Oracle Labs, USA
david.vengerov@oracle.com

Abstract

We introduce the Forseti system, which is a principled approach for *holistic memory management*. It permits a sysadmin to specify the total physical memory resource that may be shared between all concurrent virtual machines on a physical node. Forseti models the heap size versus application throughput for each virtual machine, and seeks to maximize the combined throughput of the set of VMs based on concepts from economic utility theory. We evaluate the Forseti system using a standard Java managed runtime, i.e. OpenJDK. Our results demonstrate that Forseti enables dramatic reductions (up to 5x) in heap footprint without compromising application execution times.

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

General Terms Measurement, Performance

Keywords Heap sizing, virtual machines

1. Introduction

In many data center and enterprise server scenarios, multiple virtual machines (VMs) are co-located on a single physical node. These VMs might be language-level, e.g. Java virtual machine (JVM), or system-level, e.g. Docker container. A key characteristic of such systems is the *elastic memory usage*. For an individual VM, its memory resource requirements will vary over time. For a physical node where VM spin-up and shutdown is interactive, the whole-system memory resource requirements will vary over time.

Copyright © ACM, 2015. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in ISMM'15, June 15, 2015, Portland, OR, USA, http://dx.doi.org/10.1145/2754169.2754180.

ISMM'15, June 14, 2015, Portland, OR, USA.
Copyright © 2015 ACM 978-1-4503-3589-8/15/06...$15.00.
http://dx.doi.org/10.1145/2754169.2754180

This paper introduces Forseti[1], a principled technique for *holistic memory management*, that enables a sysadmin to specify the total physical memory resource that may be shared between all concurrent VMs on a physical node. Forseti controls the maximum amount of memory that each individual VM uses, aiming to meet the total memory target specified by the sysadmin. Our Forseti system is similar to Alonso and Appel's advisor service [3]. Whereas they control memory usage to avoid paging, we have a fixed memory usage target and seek to optimize total system throughput within that constraint. In summary, our advisor shepherds multiple concurrent VMs, aiming to prevent the sum of their runtime heap sizes from exceeding a fixed amount.

1.1 Current Practice

Current recommended practice for enterprise servers running JVM applications is static heap size provisioning [18]. This static heap configuration may be based on guidelines supplied by developers (e.g. [2]) or service providers (e.g. [1]). Alternatively, optimal parameters may be determined by a search-based approach (e.g. [15]).

There are several problems with this approach. When VMs have varying memory profiles over time, or multiple VMs are admitted to the system queue at unpredictable time intervals, then it is inevitable that the static heap size provision will be suboptimal. In some cases, there will be *under-utilization*, i.e. 'spare' memory goes unused. In other cases, there will be *over-commitment* which can lead to the system running out of memory, either thrashing or causing fatal runtime errors in VMs. In both cases, the system behavior is undesirable. A holistic memory manager can address both problems. Further, a holistic memory manager is able to allocate resources to competing VMs in such a way as to maximize the overall throughput of the system.

1.2 Forseti System

Our solution, Forseti, is a system daemon that is assigned a total memory target. Individual VMs register with the Forseti daemon when they start executing. They regularly send heap

[1] named after the Norse deity of justice and reconciliation

size and throughput readings to the daemon. Each registered VM receives periodic advice from the daemon about the current system-optimum value for that VM's maximum heap size.

Forseti improves system performance, enabling concurrent VMs to run in small heap sizes. We show that, in many of these situations, best-practice static resource partitioning fails or is particularly slow. This paper addresses the challenging scenarios where changes in an application's heap size have significant effects on its throughput.

There is minimal instrumentation overhead with Forseti. In terms of the VM codebase, for instance, the Java OpenJDK runtime was extended with fewer than 100 source lines of code to provide hooks into the daemon. There is low runtime overhead too, due to socket-based communication. Effectively, the system requires one IPC call per VM garbage collection, plus n IPC calls for the n VMs every advice period (tens of seconds).

1.3 Contributions

This paper makes the following three contributions:

1. We describe the implementation of the Forseti system for dynamic heap resizing, based on principles from economic utility theory.
2. We report our experience in applying this system to the OpenJDK runtime system.
3. We evaluate the Forseti system, demonstrating that it enables relatively **low heap memory footprints** while ensuring good performance. In the best example from our results, see Section 5.4, Forseti achieves the same execution time as the default OpenJDK heap sizing mechanism for a set of four Java benchmarks, while ensuring the maximum combined heap footprint is only **20%** of the default maximum combined footprint.

2. Motivation

In a system that hosts multiple concurrent VMs, how should we partition the underlying memory resource between them to get the best overall throughput?

In general, each VM manages the size and layout of its own heap. Adjustments are made at runtime based on generic metrics such as *throughput* (proportion of execution time spent doing useful work rather than GC) which the VM measures as it runs. Existing systems are good at managing the heap to improve their own performance. However, they do not take account of other VMs running on the same system (except implicitly, through the impact other VMs have on their own performance measurements). With all the VMs aggressively trying to optimize their own performance, resources such as memory might be oversubscribed, causing system-level effects like paging or single-VM effects like memory starvation. The overall performance of the system can be degraded as a result.

It is common in enterprise applications for many VMs to be co-located on a single physical node. The node administrator may be more concerned that the overall system performs well than with the performance of any of the individual VMs. If the VMs' heap-sizing mechanisms were aware of other VMs, they might be better able to achieve this goal by acting *cooperatively* rather than *competitively*. This requires each VM to make performance information available for the rest of the system to use.

Different VMs have different needs for the layout and internal management of their heaps, depending on the garbage collection algorithms they use and the characteristics of the programs they run. Because these details are so specific, they are of little use to a general system trying to help many different VMs cooperate. Additionally, for a general system to try and dictate these details to VMs would require an enormous duplication of code and effort. Instead, such a system should be based on generic metrics that apply to all VMs, such as total heap size, throughput, and maximum permitted heap size. The system could use the heap size and throughput metrics from each VM to predict the maximum heap size each VM should be permitted, so as to maximize performance of the overall system.

Cameron and Singer [8] implement a simple multi-VM heap sizing system based on static decisions derived from analytic solutions. The heap size and throughput metrics are based on offline profiling, and the maximum heap sizes are fixed at startup. However, reported results show that it does not perform better, on average, than allowing the VMs to use their existing heap sizing mechanisms with no knowledge of each other. This static sizing approach is unsuitable for practical use. Most importantly, programs' behavior changes throughout their lifetimes, sometimes in unpredictable ways, and this static approach does not adapt to take this into account (in fact, it only works with programs that exhibit deterministic behavior).

Our work is designed to overcome these limitations. We modify several VMs to report their current heap size and throughput metrics after every GC. An additional process, the Forseti daemon, runs concurrently with the VMs and receives their reports. Periodically, it uses the most recent data to calculate the maximum heap sizes that should lead to the best overall system performance, and sends these size recommendations back to the VMs. The heap-resizing mechanisms in the VMs have been altered to respect these recommendations. In this way, the system can adapt to changing behavior. The Forseti system is described in detail in Section 4. The daemon process uses economic utility theory, see Section 3, to determine appropriate heap sizes for each VM. Other optimization strategies, perhaps based on machine learning or control systems engineering, could be plugged into the daemon instead.

3. Microeconomic Theory

Consumer theory is based on the concept of *utility*, which is a measure of a consumer's happiness. For a single commodity, a *utility function* describes how utility depends on the amount of the commodity the consumer has. Utility functions are usually increasing (more commodity gives higher utility), but with diminishing returns (the more the consumer has already, the less benefit they get from an additional unit).

This is analogous to the typical behavior we expect from a program with a managed heap [5] that is sufficiently small to avoid significant paging [24]. We assume this scenario throughout the paper. In general, the larger the heap, the less time the program spends in garbage collection, and consequently the throughput is higher (the proportion of time spent doing non-GC work). But the bigger the heap already is, the less throughput is improved by making the heap bigger still. This gives us a *throughput function* which is equivalent to a utility function, with heap size in place of quantity, and throughput in place of utility. We recognize that this throughput function may not be applicable to all GC algorithms, e.g. in the G1 collector [10] throughput is independent of heap size, assuming the marking threshold is set at a fixed fraction of the heap.

We can model the typical throughput relationship mathematically using a function which is strictly increasing, and has a derivative that is strictly decreasing but always positive; the need for these conditions is similar to other systems e.g. [4, 8, 19]. We have chosen to use a power law of the form $T(h) = ah^b$, where $a > 0$ and $0 < b < 1$. The real-valued constants a and b are application-specific. Figure 1 shows this function.

Figure 1. Example throughput function $T(h) = ah^b$, where $a > 0$ and $0 < b < 1$. The function is strictly increasing and exhibits diminishing returns.

When people buy more than one commodity, consumer theory tells us how best they should split their resources. We first combine the individual utility functions for the commodities to make an overall utility function in terms of the quantities of both commodities. We then maximize this function within the constraint that the consumer cannot spend more than their overall *budget*. This tells us how much of each commodity the consumer should buy.

In the case of memory management, we combine the throughput functions for the running programs to make an overall throughput function for the system. We chose to combine the functions using multiplication, which preserves the convex properties of the throughput function. Thus the overall function has the standard Cobb-Douglas form from economic theory [9, 16] i.e. $T_{\text{overall}}(h_1, h_2) = T_1(h_1)T_2(h_2) = a{h_1}^b c{h_2}^d$. The constraint we apply here is the *memory target*, the total amount of memory we intend to allocate to the entire system. There is a second constraint in memory management that is not present in economics: the *minimum heap* constraint. Programs require a certain minimum heap size to run at all, so we must make sure each program is given at least this amount. We now optimize the total overall throughput function within these constraints to predict the heap sizes which should give the best overall throughput of the system. Figure 2 shows this situation graphically.

Although the examples given here only use two programs, the theory generalizes to more. In the general case, the overall throughput function for N programs is $T_{\text{overall}}(h_1, \ldots, h_N) = \prod_{i=1}^{N} T_i(h_i)$.

4. Forseti System

The system[2] has two parts: the Forseti daemon which makes maximum heap size recommendations, and the VMs which have been modified to interact with it. The general architecture of the system is shown in Figure 3.

4.1 Forseti Daemon

The Forseti daemon runs concurrently with the VMs, on the same machine. It observes their behavior and makes heap sizing recommendations with the aim of improving the overall throughput of the system. The total memory target for the daemon is set at launch time. All communication between the daemon and the VMs uses Unix sockets.

4.1.1 Individual Throughput Functions

To apply the economic theory described in Section 3, an individual throughput function is required for each of the VMs in the system. The throughput functions have the form $T(h) = ah^b$, where the constants a and b are program-specific.

The VMs send readings to the Forseti daemon, consisting of the heap size and throughput of their most recent period of mutation (Section 4.2.1 describes how throughput is calculated). For each VM, the daemon keeps a cache of

[2] We will publish our code, experimental scripts and datasets to an open repository when double blind review process is complete.

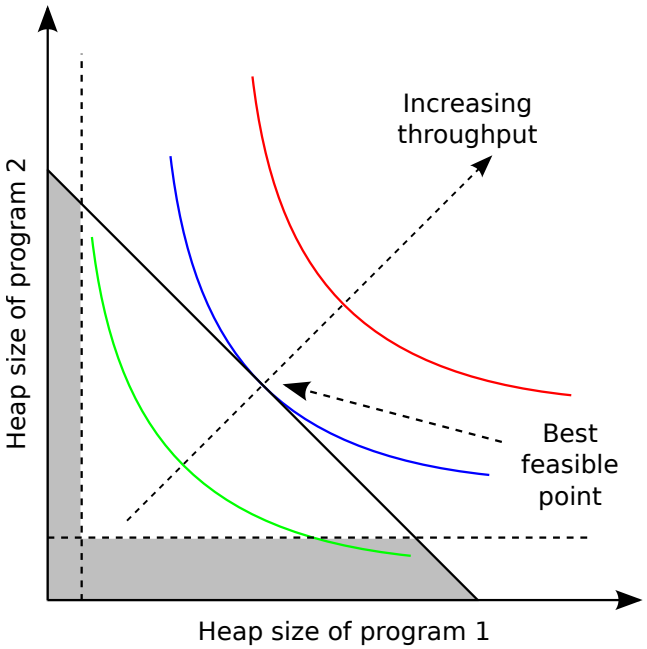


Figure 2. The combined throughput function for two programs; $T_{\text{overall}}(h_1, h_2) = T_1(h_1)T_2(h_2)$. The curved lines represent equi-throughput contours in the function. Throughput increases towards the upper right of the figure. The solid diagonal line represents the total memory target. The dashed horizontal and vertical lines represent the minimum heap sizes of the two programs, and the shaded areas are the areas excluded from consideration by the minimum heap constraint. Heap size combinations below the target line are feasible; those above it are infeasible (they use more memory than is available). The best feasible point found by the optimization occurs on the target line, where this line touches the highest equi-throughput contour line.

these readings. These caches have a fixed size, and readings are replaced using a least-recently-received policy. When the VM's behavior undergoes a phase change, old readings from the previous phase (which no longer reflect the VM's behavior) will gradually be replaced by new readings from the current phase. The readings in the cache should therefore always reflect the VM's current behavior. New readings for sizes that are already in the cache are combined with the existing reading using an exponentially-decaying average, so that the influence of the old readings decreases over time.

When the daemon requests an individual utility function, the constants a and b are calculated by curve-fitting the general function given above to the readings in the cache. Curve-fitting only makes sense when there are at least two points in the cache; if this is not the case, then no function can be generated. Therefore, the very first change in heap size for a VM is determined by the default underlying ergonomics system without consulting the Forseti daemon; this allows the system to get a second point for that VM.

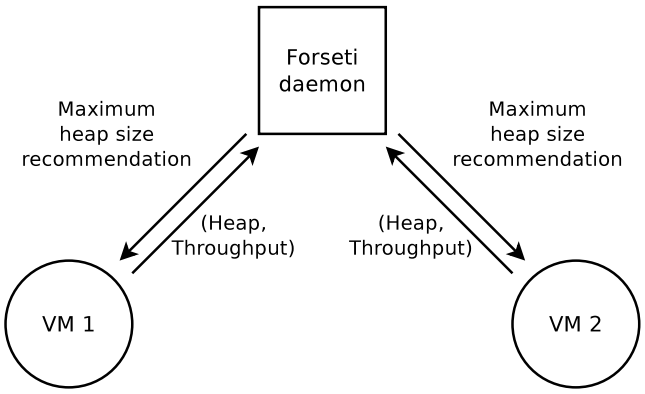


Figure 3. High-level architecture of the system. All N VMs (here $N = 2$) report their heap size and throughput to the Forseti daemon after every GC. The daemon periodically calculates maximum heap size recommendations based on recent readings, and sends these back to the VMs.

4.1.2 Making Recommendations

VMs send readings to the daemon whenever a GC completes. The daemon responds at regular intervals (by default, every ten seconds). At this time, an individual throughput function is generated for each VM, and the daemon optimizes the combined throughput function to find the combination of heap sizes which is predicted to give the best overall throughput.

If only one VM is present, it is assigned all the memory available, respecting the target. If two or more VMs are present, the overall throughput function is optimized using the L-BFGS-B algorithm [7] for numerical optimization, which suggests new values of heap sizes for individual VMs that are expected to increase the combined throughput based on the fitted throughput functions. The daemon then sends out the recommendations to their respective VMs

When a VM receives a heap size recommendation, it will grow or shrink its heap as appropriate at the next GC event. The step size of the change is determined by the HotSpot ergonomics component, which alters heap size in a gradual manner. The VM continues to act on the same heap size recommendation until it receives a new recommendation from the daemon. In general, the frequency of recommendations is lower than the frequency of GCs.

4.1.3 VM Lifecycle

The Forseti daemon also responds to changes in the number of VMs. A newly-created VM announces itself to the daemon by sending its first reading. At the next recommendation, the daemon will take the new VM into account. When a VM terminates, it should send a death notification to the daemon, which will delete the late VM's reading cache. Even if a VM crashes or fails to send a death notification, the daemon will detect its absence eventually (attempting to send a recommendation to it will fail), and clean up appropriately.

4.2 Modified VMs

VMs intended to work with the Forseti system must do two things: send readings to the daemon, and receive recommendations from it. The VM must be modified to support this, although we have encapsulated most of the work in a library so that the modifications are as small as possible. We have modified the OpenJDK VM to work with our system.

4.2.1 Calculating Throughput

After every GC, the VM should send a reading to the daemon consisting of two things:

- The heap size during the period of mutation immediately preceding this GC.
- The throughput over the whole period of this GC and the mutation immediately preceding it.

Throughput is defined by the formula:

$$throughput = \frac{mutationTime}{mutationTime + gcTime} \quad (1)$$

In a VM with only a single type of GC, the definition of throughput can be applied directly.

In a VM with generational GC, the situation is more complicated. Applying the throughput definition directly to all GCs gives different values of throughput for major and minor GCs. Minor GCs, which typically have short pause times, give high throughput readings, while major GCs, with much longer pause times, give much lower throughput readings. The outlying points representing major GCs disrupt the curve-fitting algorithm used in the daemon to such an extent that the generated functions are essentially useless.

Vengerov [22] proposes an analytic throughput model for generational heap layouts, which defines a smoothed throughput function based on parameters that are observed at each major and minor GC. Although the conditions in our system do not match all the assumptions used to derive the analytic model (e.g. it is derived for a VM in a steady state, whereas we are also concerned with startup behavior), we apply it for our experiments throughout this paper and get useful results.

4.2.2 Proxy Library

All communication and most of the throughput calculations are encapsulated inside a proxy library with a simple and well-defined interface. The VM must initialize the proxy at startup, and delete it at shutdown. The library is implemented in C++, which it is expected most VMs will be able to link against.

From the VM programmer's point of view, all that is required to send readings to the daemon is to:

- Call a function when a GC starts, and another when a GC ends.
- Provide a way for the library to query the current heap size (by implementing a pure virtual function).

To receive recommendations from the daemon, the library uses a background thread. The VM can then call a non-blocking function to test whether a new recommendation has been received since the last call. It is then up to the VM programmer to modify the heap-sizing code to treat that recommendation as an upper limit, which the heap will not exceed.

The proxy library can report an estimate of the minimum heap size of the currently running program to the daemon. This value may be specified by the sysadmin or extrapolated conservatively from observed runtime behavior.

4.2.3 HotSpot JVM

We modified version 8 of the HotSpot JVM provided by OpenJDK to use the proxy library and communicate with the Forseti daemon. Specifically, we modified the 'parallel scavenge' garbage collector, which is a generational collector, and hence we used the Vengerov method to calculate throughput.

To take account of the recommendations received, we modified the 'adaptive size policy', which decides how the heap should be resized after each GC. The existing policy checks a number of metrics to make its decision. We added our recommendation check as the first (i.e. highest priority) metric: if the heap is bigger than the recommended size, shrink (using the existing resizing code); else, continue with the original logic. Because of the encapsulated proxy library, the changes needed to implement this were minimal. The sizes received from the daemon are *recommendations*, not exact instructions, so the heap is not immediately forced to that size. Instead, the existing heap-sizing mechanism grows and shrinks the heap gradually, within the recommended size limit.

5. Evaluation

We performed experiments to compare the Forseti system against the default heap-sizing algorithm in OpenJDK.

5.1 Experimental Platform

All experiments were carried out on a single machine with a quad-core Intel Core i5-3570 CPU clocked at 3.4 GHz, 16 GB of RAM, running Linux Mint 17 with kernel 3.13.0-24-generic 47-Ubuntu SMP. The VM used in the experiments was our modified version of HotSpot from OpenJDK 8. All unnecessary services on the machine were disabled.

5.2 Selecting Experimental Workloads

The DaCapo benchmark suite [6] provides several Java benchmarks based on real-world open-source applications. These benchmarks consist of fixed-workload iterations, where each iteration is a repetition of the benchmark program in the same JVM process. By adjusting the number of iterations, we can construct a fixed-workload benchmark of any desired length. We then treat the benchmark as a black box, and consider a single execution of a JVM running N iterations

to be a single fixed-workload program. By running several of these programs concurrently (in separate JVM processes), we can construct a multi-VM workload.

Previous work by Cameron and Singer [8] indicates that workloads consisting of two instances of the same benchmark have different performance characteristics from combinations of two different benchmarks. We therefore randomly selected several same-benchmark workloads and several different-benchmark workloads from the set of all possible two-VM combinations of the DaCapo benchmarks (ignoring any benchmarks known not to work under HotSpot 8).

5.3 Selecting Memory Target Sizes

For each combination of benchmarks, we selected memory targets to use in the experiments based on multiples of the total minimum heap size of the combination (i.e. the sum of the individual minimum heap sizes of the benchmarks in the combination). The minimum heap size for each benchmark (memory needed to complete without crashing) is determined through manual experimentation. However, we did this only for the purpose of determining experimental environments with different 'stress levels' on each VM (multiples 1.1, 1.3, 1.5, 1.7, and 1.9$\times$ the minimum heap size), which in turn are used to illustrate how Forseti works in these different regimes. In a real deployment, we are targeting contexts where the VMs are already running (i.e. the sysadmins have already figured out the amount of memory required to run each application), and the sysadmins are now interested in reducing the total memory target (still keeping it above the sum of the minimum heap sizes that have previously determined) while maintaining a good combined throughput for all VMs. In such contexts, the L-BFGS-B algorithm used by Forseti will automatically discover the VMs that are close to their minimum memory requirements (the ones whose throughput is greatly reduced when their heap size is reduced) and will suggest giving them more memory, thus avoiding violating the minimum heap requirement for any VM.

5.4 Illustrative Examples

This section provides a case study that gives an intuitive feel for the Forseti system's behavior, prior to running a more systematic evaluation in Section 5.5.

Figure 4 shows the heap size and throughput readings received by the daemon for execution of several Java benchmarks. For these initial experiments, each benchmark is executed in isolation on a single VM and the daemon provides pseudo-random heap size recommendations. A throughput function for each benchmark is generated by curve-fitting the general throughput function $T(h) = ah^b$ to the observed points. The function has the expected shape for the power law throughput function shown in Figure 1.

Figures 5 and 6(a) show the behavior of four distinct VMs managed by the Forseti daemon over their lifetimes. The four VMs, indicated by the different colored lines, start execution at 100s intervals and finish at different times. In Figure 5, each plot shows the actual and recommended heap size for a single VM. In Figure 6(a) the upper panel shows actual heap sizes (gray lines) and the memory target (horizontal dotted black line) for all VMs, as well as the total actual memory usage (dashed black line). The vertical dashed lines show when each VM was created and destroyed. The bottom panel shows throughput for each of the VMs, over the same timescale. The daemon makes recommendations at ten-second intervals throughout the run.

The heap sizes of the VMs expand to make use of available memory, and contract when they become bigger than the recommended size. When a new VM starts, it has to complete several GCs before the daemon can build a model for it. Until this happens, the total memory usage tends to increase above the target, as can be seen at around 120 seconds. Except for these brief periods, the system keeps the total memory usage close to or below the target.

Figure 6(b) shows the behavior of the same four VMs without the Forseti daemon. In this case, the existing per-VM OpenJDK heap sizing mechanism is in full control, and is agnostic of the memory target. Comparing the two figures, we see that the total runtime of the four benchmarks is similar: 675 s with the daemon, and 667 s without it. However, the run with the daemon uses much less memory overall: 269 MB at the highest point, compared with 1411 MB at the highest point without the daemon. We see that the Forseti daemon allows a small time penalty (8 s) to be traded for a dramatic decrease in overall memory usage (the high watermark is 5.2$\times$ lower with Forseti).

5.5 Experimental Methodology

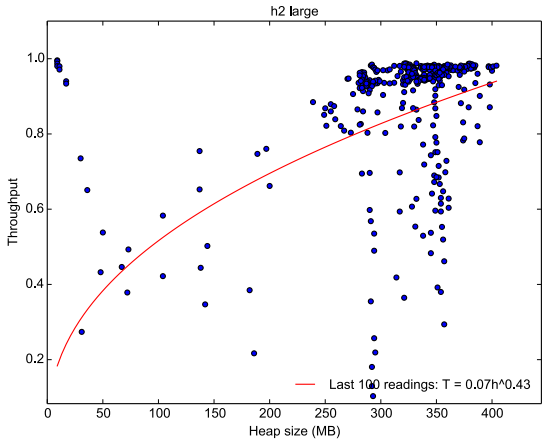
We compare three different heap-sizing mechanisms for concurrently-executing VMs, using various memory target values, M.

Unconstrained The baseline case; the VMs are run without the Forseti daemon. No size recommendations are made, and the VMs' existing heap-sizing mechanisms have full control. The maximum heap size for each VM is statically set to the target, M, using the `-Xmx` command-line parameter.

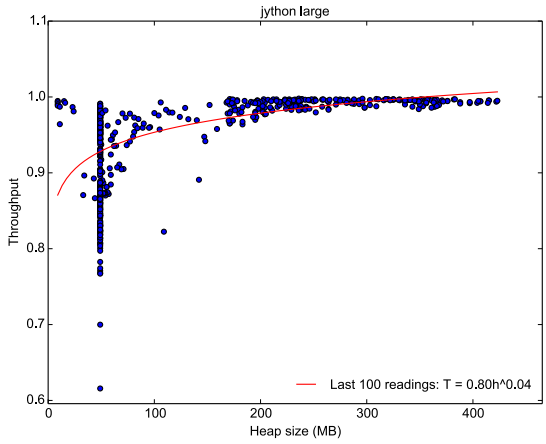
Static equal Memory is distributed equally among the VMs. The VMs are run without the Forseti daemon. No size recommendations are made, and the VMs' existing heap-sizing mechanisms have full control. The maximum heap size for each of the N VMs is statically set to M/N using the `-Xmx` command-line parameter.

Daemon The VMs are run concurrently with the Forseti daemon, and communicate with it. The daemon makes recommendations by applying economic theory to the readings provided by the VMs.

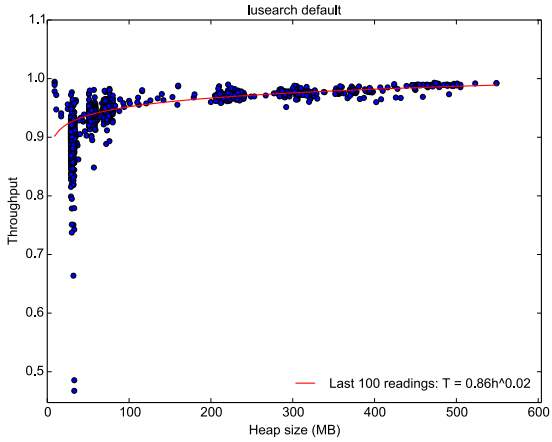
For every combination of workload, memory target size, and heap-sizing mechanism, the benchmarks in the workload are started concurrently on distinct JVMs, and the wallclock



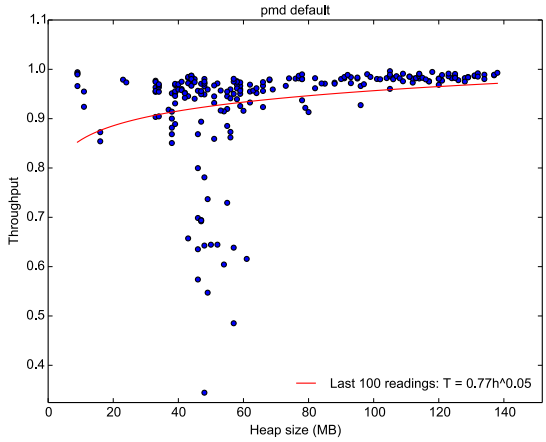
(a) h2/large



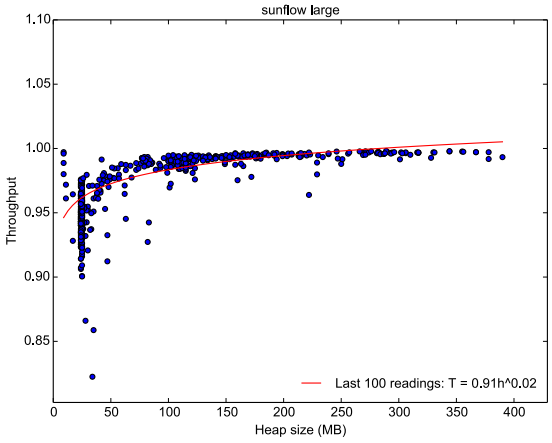
(b) jython/large



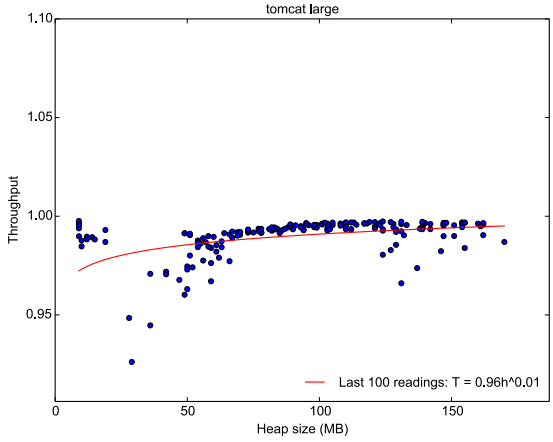
(c) lusearch/default



(d) pmd/default



(e) sunflow/large



(f) tomcat/large

Figure 4. Heap size and throughput readings received by the Forseti daemon issuing pseudo-random recommendations. The throughput function is generated by curve-fitting the general power law function to the readings.

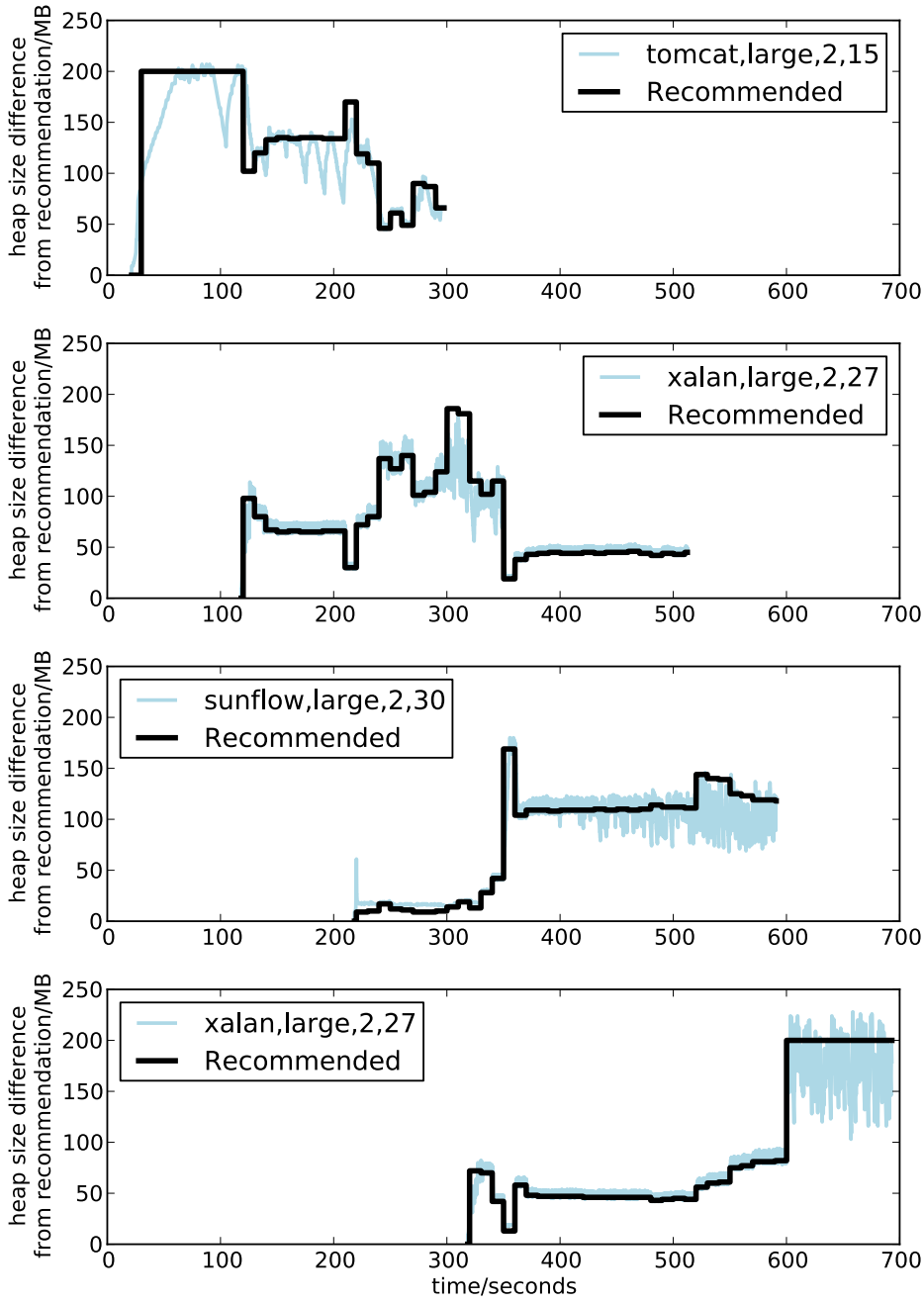
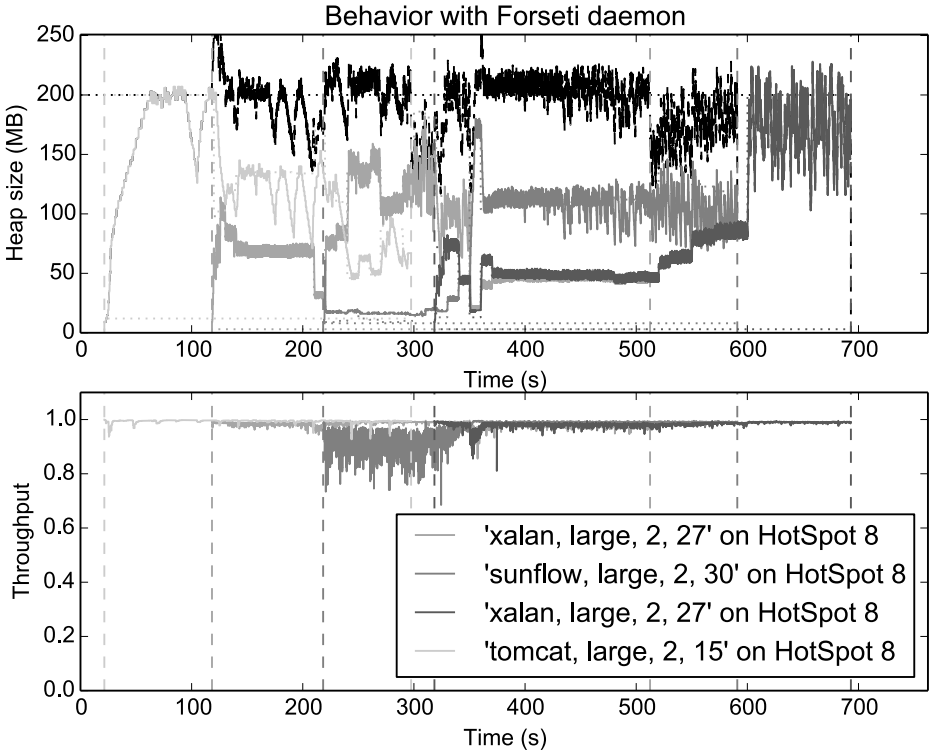
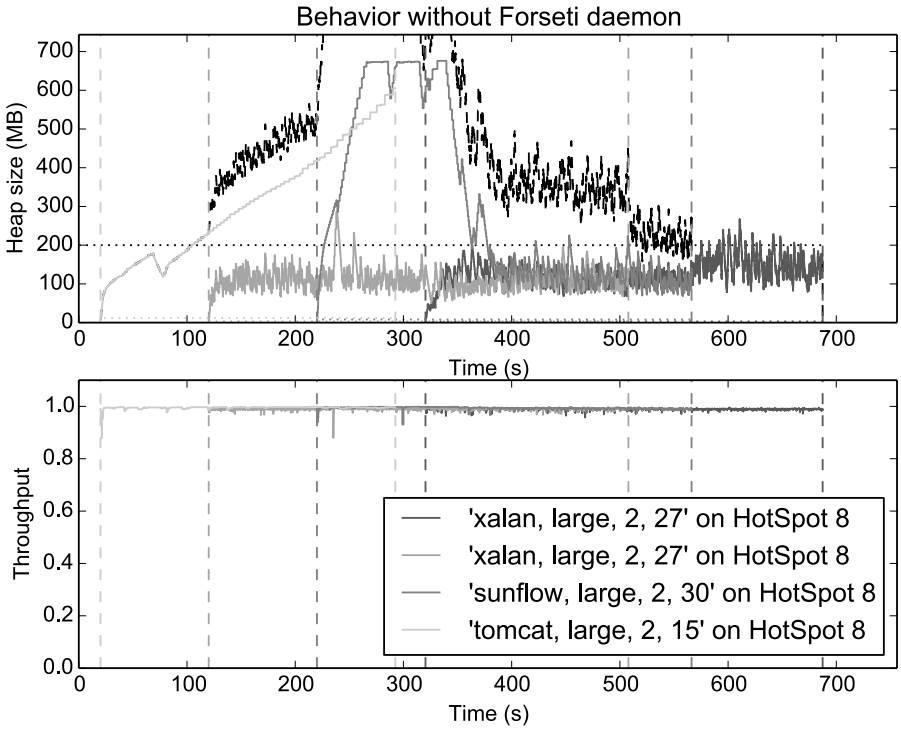


Figure 5. Behavior of four VMs with staggered start times at 0, 100, 200 and 300 seconds. Each plot shows the heap size for a single VM against time. The thicker black line shows the heap size recommendation made by the Forseti daemon, for that individual VM. At all times in this experiment, the sum of the size recommendations is 200 MB. The same four VMs are plotted on a single graph in Figure 6(a).



(a) with Forseti daemon



(b) default heap sizing

Figure 6. Behavior of four VMs with staggered start times at 0, 100, 200 and 300 seconds. The highest black line (only visible when more than one VM is executing, i.e. between around 100 and 600 seconds) indicates total combined heap size. In each case, the upper chart shows heap size, and the lower chart shows corresponding throughput. Note that the combined heap footprint is much smaller with the Forseti daemon, but the overall runtime is almost identical.

time for all of them to complete is recorded. This is repeated ten times. The heap-sizing mechanisms are compared to see which one gives the lowest total runtime on average.

5.6 Results

The results of the experiments are presented in Figure 7 and Table 1.

Benchmark combinations are defined in terms of name, input size, threads, and iterations, as described in Section 5.2. For each combination, memory targets between 1.1 and 1.9 times the combined minimum heap size are used, as described in Section 5.3; these are reported both as multiples and as absolute sizes in Table 1. For each benchmark and heap size, the execution times of the combination using the three heap-sizing mechanisms are compared (results are the mean of 10 repetitions, and uncertainties are $\pm$ one standard deviation from the mean). Figure 7 shows these execution times, relative to the baseline unconstrained configuration.

In most cases, the unconstrained baseline is the fastest, the static equal partition is the slowest, and the runtime with the Forseti daemon is in the middle (the times in Figure 6(b) in the previous section correspond to the unconstrained case). There are some anomalous results in the *static equal* case, where the execution time is an order of magnitude higher than the rest. These are cases where the combination has enough memory to complete, but only just; one of the VMs spends a long time in garbage collection before finally completing. Some cases do not complete successfully at all, because one of the benchmarks runs out of heap space and crashes. None of the Forseti daemon or unconstrained experiments crash.

Table 1 compares the combined heap size actually used for each configuration. For the static equal and unconstrained cases, these are predictable upper limits, because they are set statically. In the static equal case, the memory target M is divided equally between the VMs (the 'Target (MB)' column in the table). In the unconstrained case, the maximum combined heap size is $M \times N$, because each of the N VMs is given M. In the Forseti daemon case, we report the *maximum* of the sum of heap sizes over all JVMs, across the ten repetitions of each benchmark/target combination; this is considered to be an upper limit. In almost all cases, the maximum total memory usage with the daemon lies between the other two cases.

These results show that the Forseti daemon allows us to achieve two things:

- In comparison to the baseline unconstrained case, using the daemon allows us to trade off a small amount of execution time (arithmetic mean 5.7% slowdown) for a large reduction in maximum heap size (arithmetic mean 32% smaller). This is consistent with the illustrative example given in Section 5.4. It also gives us a simpler way to control the total heap size (by setting just one number, the target) than attempting to set the maximum heap sizes of each of the benchmarks statically to give the best performance.
- In comparison to the naive static equal case, using the daemon allows us to decrease the running time, or in some cases allow benchmarks to run to completion that otherwise would not complete at all, in exchange for a moderate amount of extra heap space (arithmetic mean 41% larger).

5.7 Overhead

In all reported experiments, the time overhead for running the Forseti daemon is small. We analyzed the 6104 experimental runs completed for this paper:

- mean experiment wall clock time is 412 seconds (max is 2300 seconds).
- mean daemon CPU time is 1.00 seconds (max is 5.94 seconds).
- mean daemon memory footprint is 23MB (max is 29MB).

The majority of the daemon memory footprint is occupied by the Python runtime, which is embedded in the daemon since it relies on the Python scipy L-BFGS-B implementation. We intend to develop a more time- and space-efficient daemon that uses C bindings to the original Fortran implementation of L-BFGS-B [28]. Note that the memory footprint measurements include storage for daemon history tables. The socket-based communication time between the daemon and the JVM is negligible.

6. Related Work

The closest related work is the advisor service proposed by Alonso and Appel [3]. They advocate system-wide centralized management of heap sizes for garbage-collected runtimes that have flexible working sets. The advisor aims to avoid page faults. There is no fixed memory target; instead it dynamically varies to minimize system paging. Multiple SML/NJ runtimes execute concurrently. They communicate with the advisor at every major GC, and receive advice about how to resize their heaps. The heap sizing model is based on an analytic model of copying GC. The advisor aims to equalize the CPU time distribution across all SML/NJ runtimes, including their GC overhead. At the same time, it seeks to minimize the overall GC overhead. Our Forseti system communicates with the daemon more frequently—at both minor and major GCs due to the use of Vengerov's throughput model [22]. Our daemon seeks to maximize application throughput across all VMs.

Hertz et al. [12] describe a cooperative GC scheme for system-wide paging avoidance. It is a best-effort approach, using a shared memory buffer for all VMs to record current resource usage and indicate if they are incurring page faults during GC. There are several different heuristic strategies to determine which other VM will voluntarily shrink its heap,

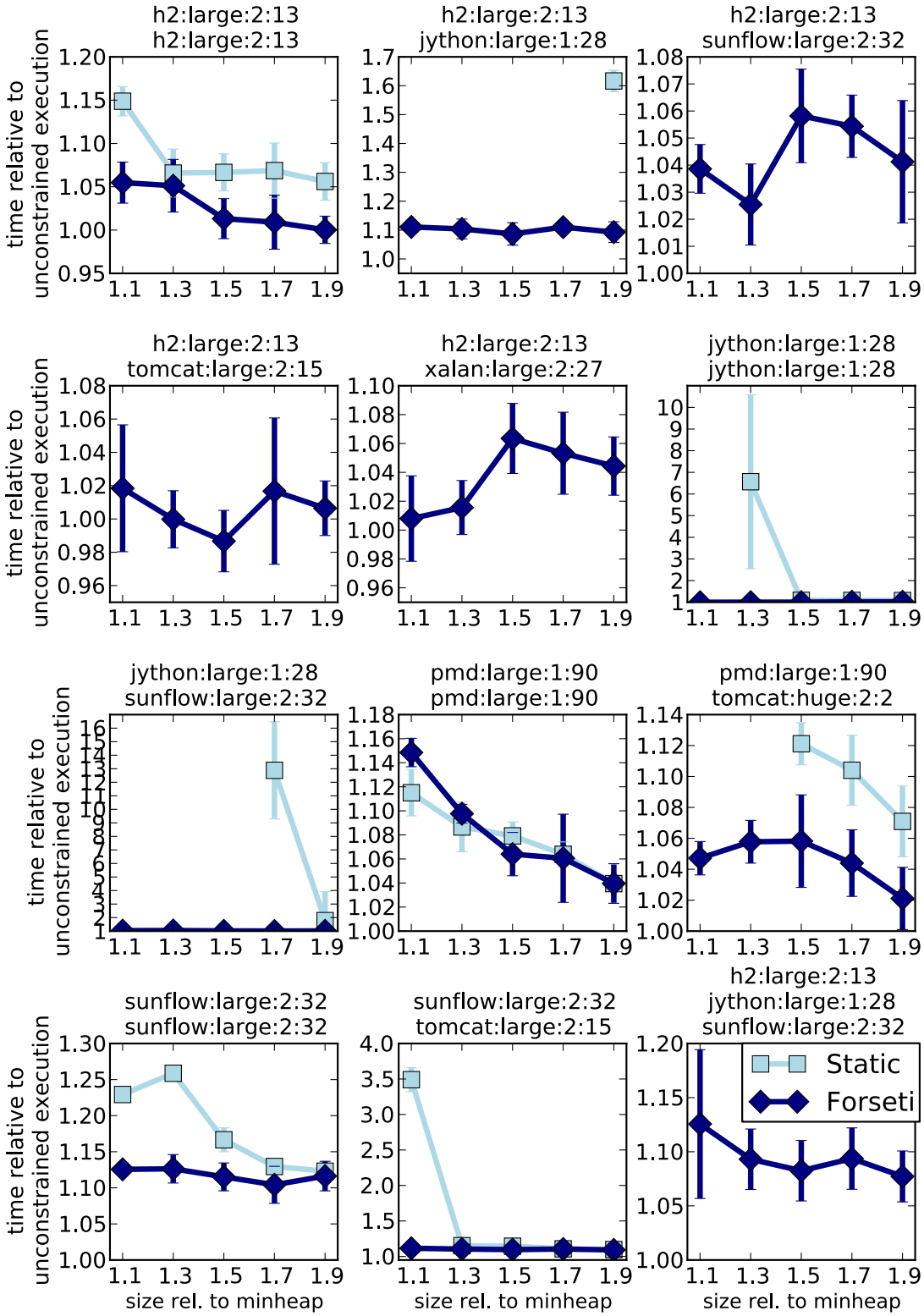


Figure 7. Experimental results, showing execution times for benchmark combinations with 'Static equal' and 'Forseti daemon' heap size configurations (lower is better). Times are reported relative to the baseline 'Unconstrained' configuration, see Section 5.5 for details. In some cases, there is no time reported for a 'Static equal' experiment since one of the benchmarks exhausted its heap space and crashed.

Benchmark combination (name, size, threads, iterations)		Target factor	Target (MB)	Daemon max heap (MB)
h2, large, 2, 13	h2, large, 2, 13	1.1	607	726
		1.3	718	824
		1.5	828	888
		1.7	938	946
		1.9	1049	999
h2, large, 2, 13	jython, large, 1, 28	1.1	328	429
		1.3	387	491
		1.5	447	547
		1.7	507	565
		1.9	566	593
h2, large, 2, 13	sunflow, large, 2, 32	1.1	312	394
		1.3	369	462
		1.5	426	498
		1.7	483	510
		1.9	540	567
h2, large, 2, 13	tomcat, large, 2, 15	1.1	317	398
		1.3	374	432
		1.5	432	500
		1.7	490	511
		1.9	547	552
h2, large, 2, 13	xalan, large, 2, 27	1.1	307	387
		1.3	363	451
		1.5	418	480
		1.7	474	480
		1.9	530	515
jython, large, 1, 28	jython, large, 1, 28	1.1	48	90
		1.3	57	94
		1.5	66	98
		1.7	75	99
		1.9	84	117
jython, large, 1, 28	sunflow, large, 2, 32	1.1	33	61
		1.3	39	71
		1.5	45	76
		1.7	51	81
		1.9	57	84
pmd, large, 1, 90	pmd, large, 1, 90	1.1	66	97
		1.3	78	118
		1.5	90	130
		1.7	102	148
		1.9	114	157
pmd, large, 1, 90	tomcat, huge, 2, 2	1.1	46	73
		1.3	55	82
		1.5	63	92
		1.7	71	97
		1.9	80	99
sunflow, large, 2, 32	sunflow, large, 2, 32	1.1	18	34
		1.3	21	42
		1.5	24	46
		1.7	27	53
		1.9	30	56
sunflow, large, 2, 32	tomcat, large, 2, 15	1.1	22	42
		1.3	26	49
		1.5	30	50
		1.7	34	53
		1.9	38	54
h2, large, 2, 13 jython, large, 1, 28 sunflow, large, 2, 32		1.1	337	543
		1.3	398	587
		1.5	459	687
		1.7	520	746
		1.9	581	761

Table 1. Experimental benchmark combinations, showing heap sizes used for each experiment. For 'Unconstrained' execution, each individual VM has a maximum heap size equal to the target. For 'Static equal' execution with N benchmarks, each individual VM has its heap size set to $1/N$ of the target. For 'Forseti daemon' execution, we report the maximum observed value of the sum of actual heap sizes for all VMs.

to allow the paging VM to make progress. There are many other systems to avoid or mitigate paging by heap resizing. However these solutions are more intrusive, either modifying the kernel virtual memory system [25], relying on program profiling [20, 26], or introducing new kernel modules [11].

Forseti differs from these systems in two respects. First, we have a fixed memory usage target, rather than a dynamically varying boundary imposed by the paging limit of the system. We control the heap sizes of multiple VMs to co-exist at or below the specified target if possible. Second, we aim to optimize overall throughput across the hosted VMs, based on microeconomic utility. Other systems feature best-effort or heuristic approaches. We note that microeconomic utility has been previously applied to JVM heap sizing [8], however this was only in the context of static heap sizing. In contrast, our Forseti system dynamically adapts to workload phase changes and unpredictable VM starts and finishes.

In this work, we have focused entirely on language-level VMs, such as the JVM architecture. We expect that the same techniques should be applicable for partitioning memory resource across system-level VMs, like Xen VMs or Docker containers. Salomie et al. [19] describe a modification to the OpenJDK runtime to support memory ballooning. Their technique is generally applicable to any applications that manage their own memory. They acknowledge that they are only providing a mechanism for memory resource allocation, without any specific allocation policy. Ginkgo [13] presents another JVM ballooning policy based on JNI, with a set of resource allocation constraints induced by service level agreements and solved with linear programming. Kim et al. [14] describe an approach to handle memory resource allocation for an isolated group of VMs, which may be subject to a common service level agreement. This is similar to our concept of a group target for memory resource in Forseti. Zhao et al. [27] present a working set size estimator to predict the memory requirements of each VM, then they use a memory resizer to determine the memory allocation for each VM. In the case that all working sets can fit into the memory, then memory is distributed proportionately to each VM. In the case of memory shortage, a hill-climbing algorithm is used to determine how to allocate memory between VMs so as to minimize page misses.

Finally, we realize that it is possible to consider the fair allocation of other physical resources across multiple runtimes, such as CPU or network bandwidth. A likely future trend is the consolidation of resource management for VMs at the rack-scale [17]. Our Forseti system is a small step in this direction.

7. Future Work

The Forseti daemon has several configurable parameters. Most important is the memory target size, which must be specified when the daemon is launched. The size of the caches, the frequency of recommendations, and the weighting given

to old and new readings in exponential averages can all be adjusted. All results reported in this paper have used the default values for these parameters (100 entries per cache, recommendations every 10 seconds, and a 50% weighting in the exponential averages). The performance may be improved relative to the results in Table 1 if smarter feedback learning mechanisms are used to tune the parameters. Such tuning, however, is outside the scope of this paper. We aim to demonstrate the *feasibility* of fitting functions to experimentally observed throughput points and then using a global optimization algorithm to change heap allocations for each VM. Our approach is shown to work well even without tuning.

It would be useful to investigate the performance of the system with heterogeneous VM combinations and potentially other runtime systems with managed memory, such as those for Haskell, Go, and JavaScript.

Many of the values observed by the system are noisy, as a result of non-determinism introduced by process scheduling, the non-deterministic nature of GC and JIT compilation in HotSpot, and the IPC used for communication between VMs and the daemon. In some cases, the throughput functions generated by the daemon, and hence the heap size recommendations made by it, are sensitive to noise. Introducing additional smoothing of data into various parts of the system might help to counter this.

At the moment, the system uses the microeconomic optimization model outlined in Section 3, but this could be replaced by another model, such as the market-based approach proposed by Vengerov [21], a control system [23] or a search-based approach. The daemon could be extended with a plugin system to allow these optimization models to be replaced and modified by a system administrator.

8. Conclusions

We have introduced Forseti, a *holistic memory manager* that seeks to maintain a set of managed runtime heaps within a fixed overall memory target size. Forseti uses principles from microeconomic utility theory to partition memory resource dynamically in such a way as to maximize system throughput. Our results indicate that Forseti enables dramatic (up to 5x, see Figure 6) reductions in OpenJDK heap footprints without significant execution time increase. Thus we feel that Forseti would be useful for cloud servers which host unpredictable, constantly changing, multi-VM workloads.

Acknowledgments

This research was supported by the EPSRC (under grant EP/L000725/1). We thank Jennifer Sartor for her patience and care in shepherding this paper. We also thank Mario Wolczko for his encouragement to pursue this work.

References

[1] Amazon elastic mapreduce. Checked 10 Feb 2015, `http://docs.aws.amazon.com/ElasticMapReduce/latest/DeveloperGuide/MemoryTuning.html`.

[2] The neo4j manual. Checked 25 Nov 2014, `http://neo4j.com/docs/stable/configuration-jvm.html`.

[3] 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.

[4] S. L. Bird and B. J. Smith. Pacora: Performance aware convex optimization for resource allocation. In *Proceedings of the 3rd USENIX Workshop on Hot Topics in Parallelism*, 2011.

[5] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: The performance impact of garbage collection. *ACM SIGMETRICS Performance Evaluation Review*, 32(1):25–36, 2004.

[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 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 169–190, 2006.

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

[8] C. Cameron and J. Singer. We are all economists now: Economic utility for multiple heap sizing. In *Proceedings of the 9th Implementation, Compilation, Optimization of OO Languages, Programs and Systems workshop*, 2014.

[9] C. W. Cobb and P. H. Douglas. A theory of production. *American Economic Review*, 18(1):139–165, 1928.

[10] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management*, pages 37–48, 2004.

[11] C. Grzegorczyk, S. Soman, C. Krintz, and R. Wolski. Isla Vista heap sizing: Using feedback to avoid paging. In *Code Generation and Optimization, 2007. CGO '07. International Symposium on*, pages 325–340, 2007.

[12] 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*, pages 65–76, 2011.

[13] M. R. Hines, A. Gordon, M. Silva, D. Da Silva, K. D. Ryu, and M. Ben-Yehuda. Applications know best: Performance-driven memory overcommit with Ginkgo. In *IEEE Third International Conference on Cloud Computing Technology and Science*, pages 130–137, 2011.

[14] S. Kim, H. Kim, J. Lee, and J. Jeong. Group-based memory oversubscription for virtualized clouds. *Journal of Parallel and Distributed Computing*, 74(4):2241–2256, 2014.

[15] 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.

[16] P. J. Lloyd. The origins of the von Thunen-Mill-Pareto-Wicksell-Cobb-Douglas function. *History of Political Economy*, 33(1):1–19, 2001.

[17] M. Maas, K. Asanovic, T. Harris, and J. Kubiatowicz. The case for the holistic language runtime system. In *Proceedings of the First International Workshop on Rack-scale Computing*, 2014.

[18] M. Raghavachari, D. Reimer, and R. D. Johnson. The deployer's problem: Configuring application servers for performance and reliability. In *Proceedings of the 25th International Conference on Software Engineering*, pages 484–489, 2003.

[19] T.-I. Salomie, G. Alonso, T. Roscoe, and K. Elphinstone. Application level ballooning for efficient server consolidation. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 337–350, 2013.

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

[21] D. Vengerov. A reinforcement learning approach to dynamic resource allocation. *Engineering Applications of Artificial Intelligence*, 20(3):383–390, 2007.

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

[23] 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*, pages 27–38, 2013.

[24] T. Yang, M. Hertz, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. Automatic heap sizing: Taking real memory into account. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management*, pages 61–72, 2004.

[25] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. Cramm: Virtual memory support for garbage-collected applications. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 103–116, 2006.

[26] C. Zhang, K. Kelsey, X. Shen, C. Ding, M. Hertz, and M. Ogihara. Program-level adaptive memory management. In *Proceedings of the 5th International Symposium on Memory Management*, pages 174–183, 2006.

[27] W. Zhao, Z. Wang, and Y. Luo. Dynamic memory balancing for virtual machines. *SIGOPS Oper. Syst. Rev.*, 43(3):37–47, 2009.

[28] C. Zhu, R. Byrd, J. Nocedal, and J. L. Morales. Software for large-scale bound-constrained optimization, 2011. `http://users.iems.northwestern.edu/~nocedal/lbfgsb.html`.