An Equation-Based Heap Sizing Rule☆

Y.C. Tay[a], Xuanran Zong[b], Xi He[a]

[a]*National University of Singapore*
[b]*Duke University*

Abstract

For garbage-collected applications, dynamically-allocated objects are contained in a heap. Programmer productivity improves significantly if there is a garbage collector to automatically de-allocate objects that are no longer needed by the applications. However, there is a run-time performance overhead in garbage collection, and this cost is sensitive to heap size H: a smaller H will trigger more collection, but a large H can cause page faults, as when H exceeds the size M of main memory allocated to the application.

This paper presents a Heap Sizing Rule for how H should vary with M. The Rule can help an application trade less page faults for more garbage collection, thus reducing execution time. It is based on a heap-aware Page Fault Equation that models how the number of page faults depends on H and M. Experiments show that this rule outperforms the default policy used by `JikesRVM`'s heap size manager. Specifically, the number of faults and the execution time are reduced for both static and dynamically changing M.

Keywords: garbage collection, heap size, page fault, dynamic tuning

1. Introduction

Most nontrivial programs require some dynamic memory allocation for objects. If a program is long-running or its objects are large, such allocation can significantly increase the memory footprint and degrade its performance.

☆A shorter version of this paper entitled "A Page Fault Equation for Dynamic Heap Sizing" has appeared in First Joint WOSP/SIPEW International Conference on Performance Engineering (San Jose, California, USA, January 2010, pages 201–206)[1].

Email address: `mattyc@nus.edu.sg` (Y.C. Tay)

This can be avoided by deallocating memory occupied by objects that are no longer needed, so the space can be reused.

Manual memory deallocation is tedious and prone to error. Many languages therefore relieve programmers of this task by having a **garbage collector** do the deallocation on their behalf. Several such languages are now widely used, e.g. Java, C#, Python and Ruby.

Garbage collection is restricted to the **heap**, i.e. the part of user memory where the dynamically created objects are located. The application, also called the **mutator**, therefore shares access to the heap with the garbage collector.

1.1. The Problem

The heap size H can have a significant effect on mutator performance. Garbage collection is usually prompted by a shortage of heap space, so a smaller H triggers more frequent runs of the garbage collector. These runs interrupt mutator execution, and can seriously dilate execution time.

Furthermore, garbage collection pollutes hardware data and instruction caches, causing cache misses for the mutator when it resumes execution; it also disrupts the mutator's reference pattern, possibly undermining the effectiveness of the page replacement policy used by virtual memory management [2, 3].

While a larger heap size can reduce garbage collection and its negative impact, H cannot be arbitrarily large either. Memory is cheap, but systems are also often overloaded. A competing memory-intensive job, or a burst of job arrivals at a server, may severely reduce the memory allocated to a process.

If H exceeds the memory allocation M, part of the heap will have to reside on disk. This will likely result in page faults, if not caused by a mutator reference to the heap, then by the garbage collector. (In this paper, *page fault* always refers to a major fault that requires a read from disk.) In fact, it has been observed that garbage collection can cause more page faults than mutator execution when the heap extends beyond main memory [4].

Figure 1 presents measurements from running mutator `pmd` (from the `DaCapo` benchmark suite [5]) with `JikesRVM` [6], using `GenMS` in its `MMTk` toolkit as the garbage collector. It illustrates the impact of H on how page faults vary with M.

In the worst case, $H > M$ can cause page thrashing [7]. Even if the situation is not so dire, page faults are costly — reading from disk is several orders

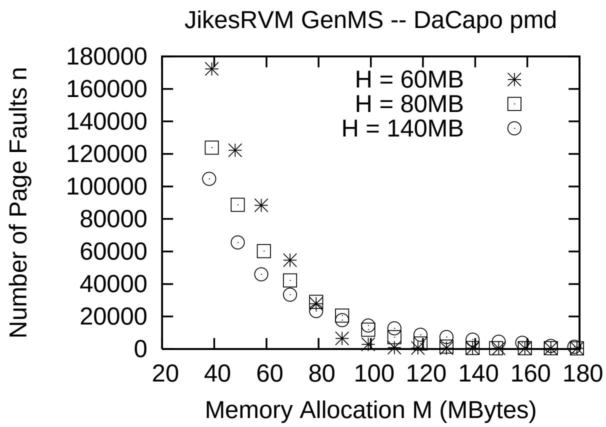


Figure 1: **How heap size H and memory allocation M affect the number of page faults n. The garbage collector is `GenMS` and the mutator is `pmd` from the `Dacapo` benchmark suite.**

of magnitude slower than from main memory — and should be avoided. It is thus clear that performance tuning for garbage-collected applications requires a careful choice of heap size.

Consider the case $H = 140$MBytes in Figure 1. If $M = 50$MBytes, then shrinking the heap to $H = 60$MBytes would trigger more garbage collection and double the number of page faults. If $M = 110$MBytes, however, setting $H = 60$MBytes would reduce the faults to just cold misses, and the increase in compute time would be more than compensated by the reduction in fault latency. This possibility of adjusting memory footprint to fit memory allocation is a feature for garbage-collected systems — garbage collection not only raises offline programmer productivity, it can also improve run-time application performance.

However, the choice of H should not be static: from classical multiprogramming to virtual machines and cloud computing, there is constant competition for resources and continually shifting memory allocation. In the above example, if $H = 60$MBytes and M changes from 110MBytes to 50MBytes, the number of faults will increase drastically and performance will plummet. H must therefore be dynamically adjusted to suit changes in M. This is the issue addressed by our paper:

How should heap size H vary with memory allocation M?

Given the overwhelming cost of page faults, it would help if we know how the number of faults n incurred by the mutator *and* garbage collector is related to M and H. This relationship is determined by the complex interaction among the operating system (e.g. page replacement policy), the garbage collector (e.g. its memory references change with H) and the mutator (e.g. its execution may vary with input). Nonetheless, this paper models this relationship, and applies it to dynamic heap sizing.

1.2. Our Contribution

The first contribution in this paper is an equation that relates the number of faults n to memory allocation M and heap size H. This equation has several parameters that encapsulate properties of the mutator, garbage collector and operating system. It is a refinement of the Page Fault Equation (for generic, possibly non-garbage-collected workloads) in previous work [8].

Our second contribution is the following

Heap Sizing Rule:

$$H = \begin{cases} \frac{M-b}{a} & \text{for } aH_{\min} + b < M < aH_{\max} + b \\ \\ H_{\max} & \text{otherwise} \end{cases} \tag{1}$$

This rule, illustrated in Figure 2, reflects any change in workload through changes in the values of the parameters a, b, $H_{\min}$ and $H_{\max}$. Once these values are known, the garbage collector just needs minimal knowledge from the operating system — namely, M — to determine H. There is no need to patch the kernel [9], tailor the page replacement policy [10], require notification when memory allocation stalls [2], track page references [4], measure heap utilization [6], watch allocation rate [11] or profile the application [12].

Rule (1) is in closed-form, so there is no need for iterative adjustments [2, 13, 14, 10, 12]. If M changes dynamically, the rule can be used to tune H accordingly, in contrast to static command-line configuration with parameters and thresholds [15, 16, 13, 17].

Most techniques for heap sizing are specific to the collectors' algorithms. In contrast, our rule requires only knowledge of the parameter values, so it can even be used if there is hot-swapping of garbage collectors [18].

1.3. An overview

We begin in Section 2 by introducing the Page Fault Equation. We validate it for some garbage-collected workloads, then refine it to derive the

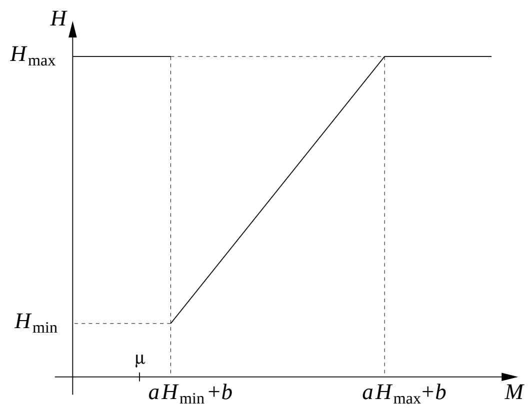


Figure 2: **Heap Sizing Rule. (μ is a lower bound identified in Section 2.5; $\mu \approx 80$ in Figure 1.)**

heap-aware version. This refinement introduces new parameters, which we interpret and inter-relate.

Section 3 derives the Heap Sizing Rule (1) from the Equation, and presents experiments to show its effectiveness for static M, dynamic M and in a multi-mutator mix.

We survey some related work in Section 4, before concluding with a summary in Section 5. Some technical details for the experiments, parameter calibration and parameter sensitivity are contained in an Appendix.

2. Heap-Aware Page Fault Equation

We first recall Tay and Zou's parameterized Page Fault Equation in Section 2.1, and Section 2.2 verifies that it works for garbage-collected workloads. Section 2.3 then derives from it the heap-aware version in Equation (6). This introduces new parameters, which we interpret in Section 2.4. Garbage collection and virtual memory interact to define a lower bound for H, and Section 2.5 examines this bound.

2.1. Page Fault Equation

Suppose an application gets main memory allocation M (in pages or MBytes), and consequently incurs n page faults during its execution. The Page Fault Equation says

$$n = \begin{cases} n^* & \text{for } M \geq M^* \\ \frac{1}{2}(K + \sqrt{K^2 - 4})(n^* + n_0) - n_0 & \text{for } M < M^* \end{cases}$$

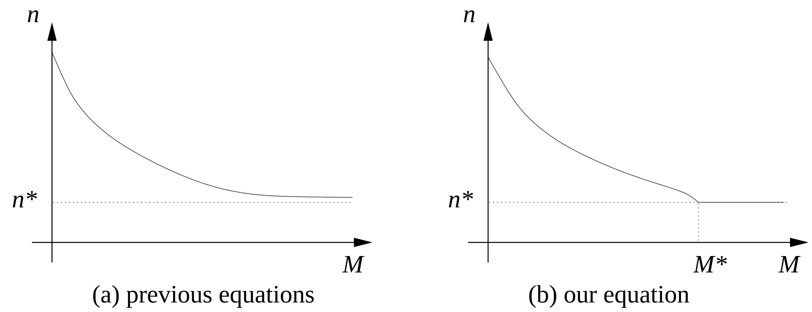


Figure 3: **Generic shape of relationship between n and M. Our equation differs from previous equations in identifying M^*.**

$$\text{where } K = 1 + \frac{M^* + M^o}{M + M^o}. \tag{2}$$

The parameters n^*, M^*, M^o and n_0 have values that depend on the application, its input, the operating system, hardware configuration, etc. Having four parameters is minimal, in the following sense:

- n^* is the number of cold misses (i.e. first reference to a page on disk). It is an inherent characteristic of every reference pattern, and any equation for n must account for it.

- When n is plotted against M, we generally get a decreasing curve. Previous equations for n models this decrease as continuing forever [19, 20, 21], as illustrated in Figure 3(a). This cannot be; there must be some $M = M^*$ at which n reaches its minimum n^*, as illustrated in Figure 3(b). Identifying this M^* is critical to our use of the equation for heap sizing.

- The interpretation for M^o varies with the context [8, 22]. For the Linux experiments in this paper, we cannot precisely control M, so M^o is a correction term for our estimation of M. M^o can be positive or negative.

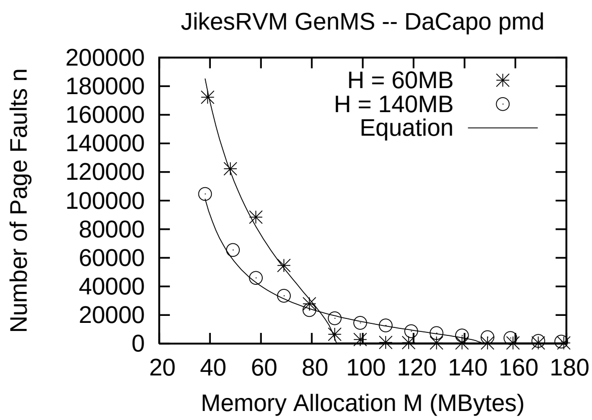


Figure 4: **The Page Fault Equation can fit data from Figure 1 for different heap sizes.**
For $H = 60$MB, $n^* = 480$, $M^* = 89.0$, $M^o = 14.8$ and $n_0 = 64021$ ($R^2 = 0.994$).
For $H = 140$MB, $n^* = 480$, $M^* = 146.2$, $M^o = 22.7$ and $n_0 = 12721$ ($R^2 = 0.993$).

- Like M^o, n_0 is a correction term for n that aggregates various effects of the reference pattern and memory management. For example, dynamic memory allocation increases n_0, and prefetching may decrease n_0 [8]. Again, n_0 can be positive or negative; geometrically, it controls the convexity of the page fault curve.

2.2. Universality: experimental validation

The Page Fault Equation was derived with minimal assumptions about the reference pattern and memory management, and experiments have shown that it fits workloads with different applications (e.g. processor-intensive, IO-intensive, memory-intensive, interactive), different replacement algorithms and different operating systems [8]; in this sense, the equation is **universal**.

Garbage-collected applications are particularly challenging because the heap size determines garbage collection frequency, and thus the reference pattern and page fault behavior. This is illustrated in Figure 1, which shows how heap size affects the number of page faults. Details on the experimental set-up for this and subsequent experiments are given in Appendix A.

Classical page fault analysis is **bottom-up**: it starts with a model of reference pattern and an idealized page replacement policy, then analyzes their interaction. We have not found any bottom-up model that incorporates the impact of heap size on reference behavior.

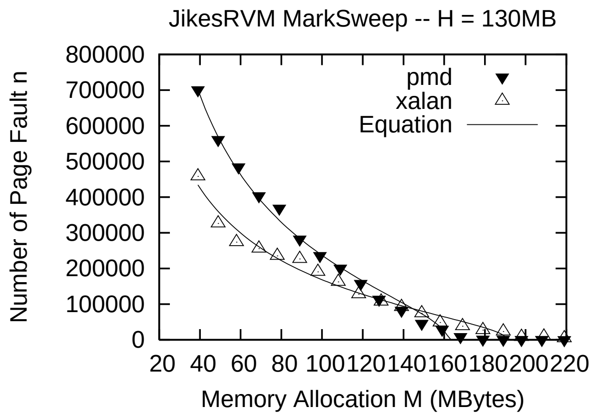


Figure 5: **The Page Fault Equation can fit data for different mutators. For `pmd`, $n^* = 420$, $M^* = 162.4$, $M^o = -12.2$ and $n_0 = 220561$ ($R^2 = 0.995$). For `xalan`, $n^* = 480$, $M^* = 151.6$, $M^o = 23.4$ and $n_0 = 12421$ ($R^2 = 0.997$).**

In contrast, for the Page Fault Equation to fit the result of a change in H, one simply changes the parametric values. Figure 4 illustrates this for the workload of Figure 1: it shows that the equation gives a good fit of the page fault data for two very different heap sizes. The goodness of fit is measured with the widely-used coefficient of determination R^2 (the closer to 1, the better the fit). Appendix B provides details on how we use regression to fit Equation (2) to the data.

A universal equation should still work if we change the mutator itself. Figure 5 illustrates this for `pmd` and `xalan`, using the `MarkSweep` garbage collector and $H = 130$MBytes.

Universality also means the equation should fit data from different garbage collectors. Figure 6 illustrates this for `pmd` run with `MarkSweep` and with another garbage collector, `SemiSpace`, using $H = 90$MBytes.

2.3. Top-down refinement

The Page Fault Equation fits the various data sets by changing the numerical values of n^*, M^o, M^* and n_0 when the workload is changed. In the context of heap sizing, how does heap size H affect these parameters?

The cold misses n^* is a property of the mutator, so it is not affected by H. Although the workload has estimated memory allocation M, it may use more or less than that, and M^o measures the difference. Figure 7 plots the

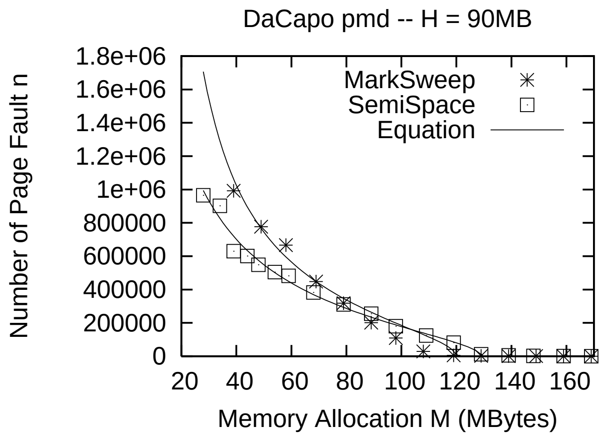


Figure 6: **The Page Fault Equation can fit data for different garbage collectors. For `MarkSweep`, $n^* = 420$, $M^* = 120.6$, $M^o = 7.9$ and $n_0 = 314318$ ($R^2 = 0.997$). For `SemiSpace`, $n^* = 420$, $M^* = 129.0$, $M^o = -5.5$ and $n_0 = 260659$ ($R^2 = 0.992$).**

M^o values when `pmd` is run with `MarkSweep` at various heap sizes. We see some random fluctuation in value, but no discernible trend. Henceforth, we consider M^o as constant with respect to H.

`MarkSweep` accesses the entire heap when it goes about collecting garbage. For such garbage collectors, the memory footprint grows with H, so we expect M^* to increase with H. Figure 8 shows that, in fact, M^* varies linearly with H for all four workloads, i.e.

$$M^* = aH + b \quad \text{for some constants } a \text{ and } b. \tag{3}$$

As for n_0, Figure 9 shows that n_0 decreases linearly with H, then flattens out, i.e.

$$n_0 = \begin{cases} cH + d & \text{for } H < H_{\max} \\ cH_{\max} + d & \text{for } H \geq H_{\max} \end{cases} \tag{4}$$

for some constants c, d and $H_{\max}$. Furthermore, a heap cannot be arbitrarily small; there is a smallest heap size such that, for any smaller H, the workload will run out of memory before completion [23]. There is therefore a bound

$$H_{\min} \leq H \quad \text{for all } H. \tag{5}$$

Equations (2), (3), (4) and (5) together give the following:

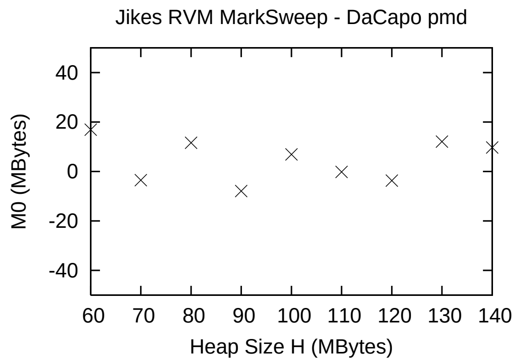


Figure 7: M^o **values for** `pmd` **run with** `MarkSweep`**. They appear to fluctuate randomly.**

Heap-Aware Page Fault Equation

$$n = \begin{cases} n^* & \text{for } M \geq M^* \\ \frac{1}{2}(K + \sqrt{K^2 - 4})(n^* + n_0) - n_0 & \text{for } M < M^* \end{cases}$$

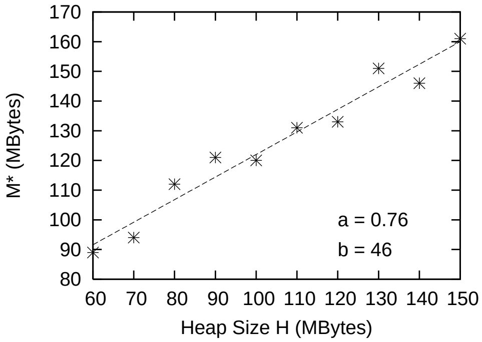
$$\begin{aligned} \text{where } K &= 1 + \frac{M^* + M^o}{M + M^o} \ , \\ M^* &= aH + b, \\ \text{and } n_0 &= \begin{cases} cH + d & \text{for } H_{\min} \leq H < H_{\max} \\ cH_{\max} + d & \text{for } H \geq H_{\max} \end{cases} \end{aligned} \tag{6}$$

Note that, rather than a bottom-up derivation, we have used a **top-down** refinement of the Page Fault Equation to derive the heap-aware version.

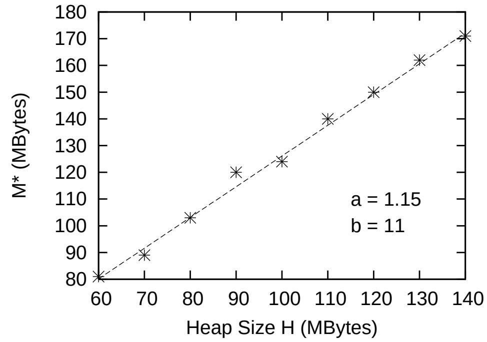
2.4. Interpreting the new parameters

Besides $H_{\min}$, the top-down refinement introduces new parameters a, b, c, d and $H_{\max}$; what do they mean?

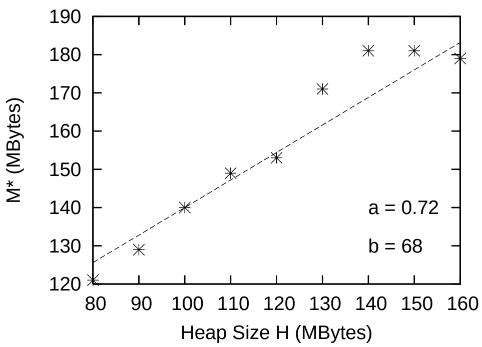
In their work on automatic heap sizing, Yang et al. defined a parameter R to be the minimum real memory required to run an application without substantial paging [4]. Their experiments show that R is approximately linear in H, with a gradient determined by the collection algorithm; in particular, they reasoned that the gradient is 1 for `MarkSweep` and 0.5 for `SemiSpace`. Their



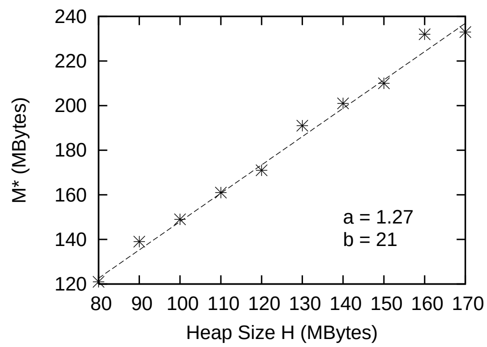
(i) `GenMS` and `pmd` ($R^2 = 0.97$)



(ii) `MarkSweep` and `pmd` ($R^2 = 0.99$)



(iii) `SemiSpace` and `pmd` ($R^2 = 0.95$)



(iv) `MarkSweep` and `xalan` ($R^2 = 0.99$)

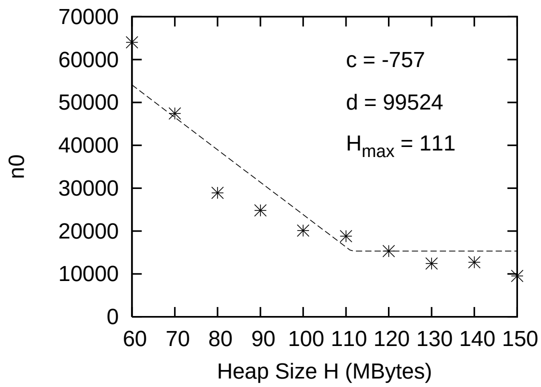
Figure 8: M^* **values vary linearly with** H**.**

R is approximately our M^*, so Equation (3) agrees with their reasoning, even if our values for gradient a in Figure 8 are not as crisp for `MarkSweep` and `SemiSpace`.

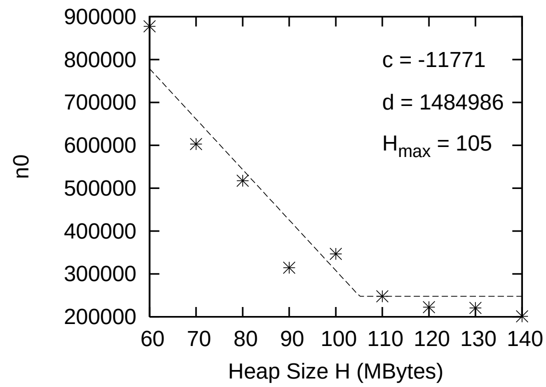
As for the intercept b, this is a measure of the space overhead — for code and stack of the mutator, garbage collector and virtual machine that is outside of the heap; for the garbage collection algorithm; etc. — that is independent of H. To generate no non-cold misses, M^* must accommodate such overheads, in addition to heap-related objects.

What can explain how n_0 varies with H in Figure 9?

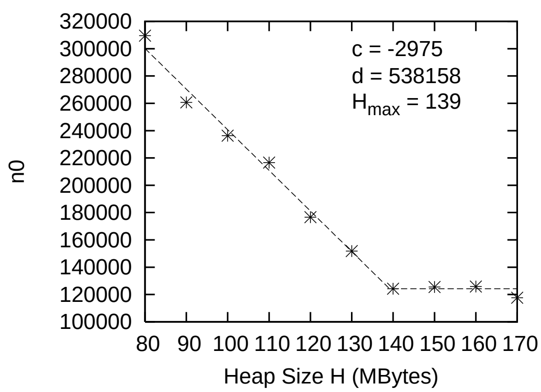
The clue lies in the fact that n_0 is positive: Recall that dynamic memory allocation increases n_0, so n_0 may (largely) measure the memory taken off the freelist during garbage collection. One expects that, if we increase H, then the number of garbage collection would decrease, unless H is so large that it can accommodate all objects created by the mutator and there is no



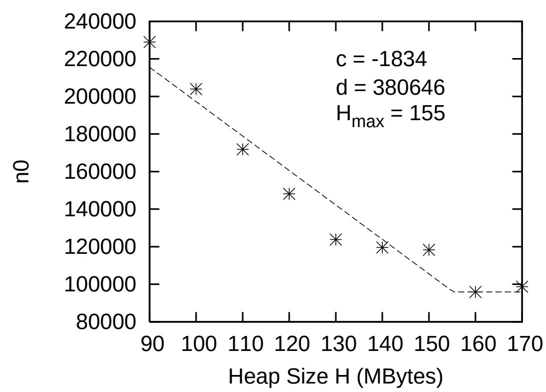
(i) `GenMS` and `pmd` ($R^2 = 0.88$)



(ii) `MarkSweep` and `pmd` ($R^2 = 0.93$)



(iii) `SemiSpace` and `pmd` ($R^2 = 0.99$)



(iv) `MarkSweep` and `xalan` ($R^2 = 0.94$)

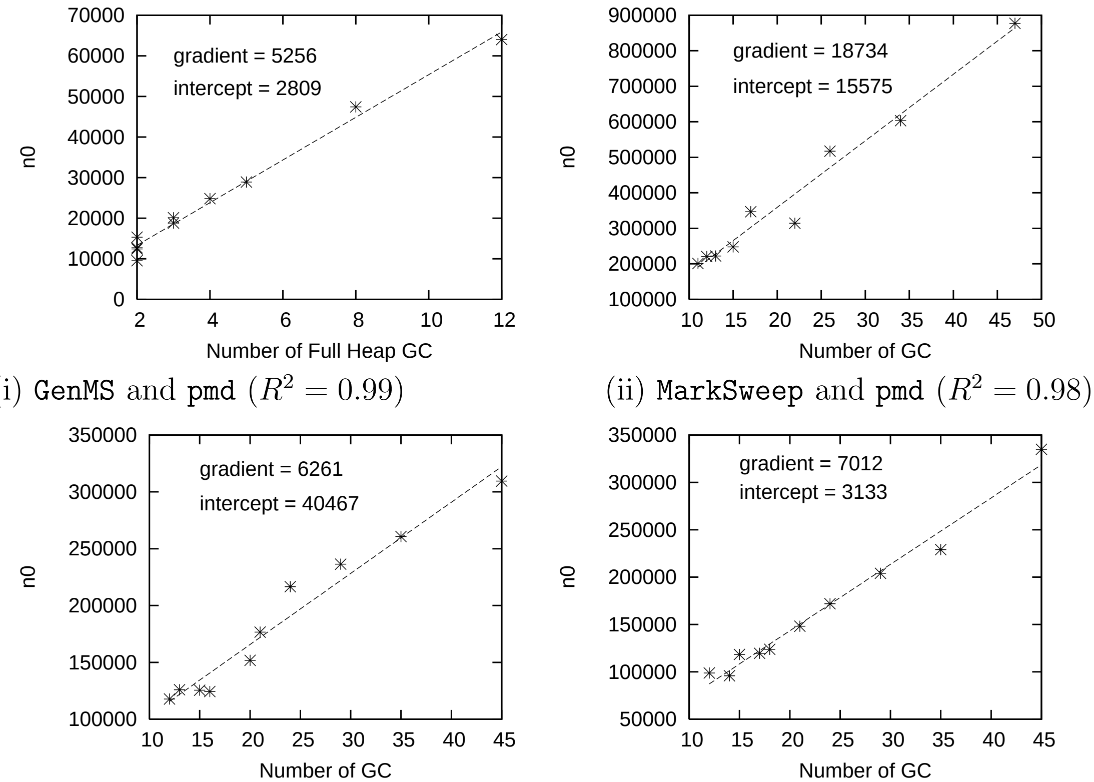
Figure 9: n_0 **decreases linearly with** H**, then flattens out.**

space shortage to trigger collection. This hypothesis matches the n_0 behavior in Figure 9.

To explicitly relate n_0 to garbage collection, we measure the number of garbage collection N_{GC} and plot $\langle n_0, N_{\text{GC}} \rangle$ for various heap sizes in Figure 10. It shows n_0 increasing linearly with N_{GC}, thus supporting the hypothesis.

`GenMS` is a generational garbage collector that has a nursery where objects are first created, and they are moved out of the nursery only if they survive multiple garbage collections. Garbage is collected more frequently from the nursery than from the rest of the heap. Let N'_{GC} be the number of collections from the nursery (not counting the full heap collections) and N_{GC} be the number of full heap collections.

Given our interpretation of n_0 as determined by the number of garbage



(i) `GenMS` and `pmd` ($R^2 = 0.99$) (ii) `MarkSweep` and `pmd` ($R^2 = 0.98$)

(iii) `SemiSpace` and `pmd` ($R^2 = 0.99$) (iv) `MarkSweep` and `xalan` ($R^2 = 0.98$)

Figure 10: n_0 **increases linearly with number of garbage collection. (For `GenMS`, this refers to the number of full garbage collection.) Each data point is generated by one heap size.**

collection, we expect to see a linear relationship among n_0, N'_{GC} and N_{GC}. Indeed regression for our `GenMS` with `pmd` workload gives

$$n_0 = -847N'_{\text{GC}} + 4726N_{\text{GC}} + 71244$$

with $R^2 = 0.99$. It is possible that the negative coefficient -847 for N'_{GC} is a measure for the objects that moved out of the nursery.

On the other hand, it may simply be a statistical reflection of the relationship between N'_{GC} and N_{GC}: Since a full heap collection includes the nursery, the number of times garbage is collected from the nursery is $N'_{\text{GC}} + N_{\text{GC}}$; this should be a constant in our experiment since we fixed the nursery size (at 10MBytes), regardless of H. Table 1 shows that, indeed, $N'_{\text{GC}} + N_{\text{GC}}$ is almost

heap size H (MBytes)	60	70	80	90	100	110	120	130	140	150
N'_{GC} (nursery)	12	8	5	4	3	3	2	2	2	2
N_{GC} (full)	74	74	78	79	80	79	79	79	79	80
$N'_{\text{GC}} + N_{\text{GC}}$	86	82	83	83	83	82	81	81	81	82

Table 1: **For `GenMS` with `pmd`, the number of nursery collections and full collections is almost constant.**

constant as H varies from 60MBytes to 150MBytes.

It follows that n_0 should be directly correlated with N_{GC} for `GenMS`, and regression shows that, in fact,

$$n_0 = 5256 N_{\text{GC}} + 2809,$$

as shown in Figure 10(i).

The interpretation of the other parameters is now clear: As H increases, there is less garbage collection and n_0 decreases. The gradient c is therefore a measure for the memory taken off the freelist during garbage collection.

For a sufficiently large $H = H_{\max}$, the heap suffices to contain all objects created by the workload. We then expect N_{GC} to stabilize, so n_0 flattens out; d is then implicitly determined by c and the kink in Figure 9.

The effect of H on M^* and n_0 explains the impact it has on page faults that we see in Figure 1, which shows that an increased H decreases n for small M, but increases n for large M, i.e. $\frac{dn}{dH} < 0$ for small M and $\frac{dn}{dH} > 0$ for large M. Now

$$\begin{aligned}\frac{dn}{dH} &= \frac{\partial n}{\partial n_0}\frac{dn_0}{dH} + \frac{\partial n}{\partial M^*}\frac{dM^*}{dH} \\ &= \Big(\frac{K + \sqrt{K^2-4}}{2} - 1\Big)\frac{dn_0}{dH} + \frac{1}{2}\Big(1 + \frac{K}{\sqrt{K^2-4}}\Big)\frac{n^* + n_0}{M + M^o}\frac{dM^*}{dH} \qquad (7)\end{aligned}$$

It is obvious from Equation (7) that $\frac{\partial n}{\partial n_0}$ and $\frac{\partial n}{\partial M^*}$ are both positive, while Figure 9 and Figure 8 show that $\frac{dn_0}{dH} \leq 0$ and $\frac{dM^*}{dH} > 0$. In other words, the page fault curve is largely dominated by the garbage collector for small M (through $\frac{dn_0}{dH}$), and by virtual memory for large M (through $\frac{dM^*}{dH}$).

2.5. A lower bound for H

The two opposing effects in Equation (7) cancel when

$$\frac{dn}{dH} = 0 \text{ at some } M = \mu;$$

in Figure 1, $\mu \approx 80$MBytes. Now, μ partitions the page fault curve so that

$$\begin{aligned} \frac{dn}{dH} &< 0 \quad \text{for } M < \mu \\ \text{and} \quad \frac{dn}{dH} &> 0 \quad \text{for } M > \mu, \end{aligned}$$

and M^* is in the latter segment. We hence have

$$\mu < M^* \quad \text{for all } M^*.$$

Since $M^* = aH + b$, we have $\mu < aH + b$ for all H. In particular, since $H_{\min}$ is the smallest feasible H (Equation (5)), we get

$$\mu < aH_{\min} + b \quad \text{(see Figure 2)}.$$

This imposes a lower bound on H, i.e.

$$H_{\min} > \frac{\mu - b}{a}.$$

Unfortunately, we failed to derive a closed-form for μ from Equation (7); otherwise, we could express this bound in terms of the other parameters.

3. Heap Sizing

How large should a heap be? A larger heap would reduce the number of garbage collections, which would in turn reduce the application execution time, unless the heap is so large as to exceed (main) memory allocation and incur page faults. Heap sizing therefore consists in determining an appropriate heap size H for any given memory allocation M.

The results in Section 2 suggest two guidelines for heap sizing, which we combine into one in Section 3.1. For static M, Section 3.2 compares this Rule to that used by `JikesRVM`'s default heap size manager. In Section 3.3, we do another comparison, but with M changing dynamically. Dynamic changes in M can occur when a mutator is part of a heavily-loaded multiprogramming mix. Section 3.4 therefore presents experiments where there are multiple mutators, and compares our Rule to Poor Richard's Memory Manager [24].

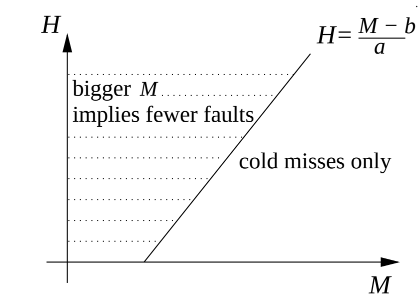


Figure 11: **Guideline for heap sizing from the Heap-Aware Page Fault Equation (6).**

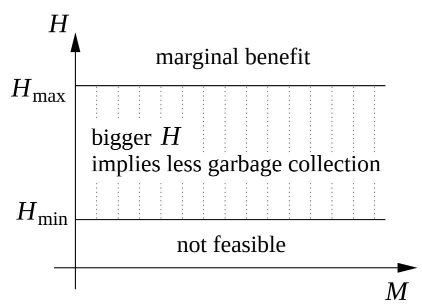


Figure 12: **Guideline for heap sizing from Figure 9.**

3.1. Heap Sizing Rule

The Heap-Aware Page Fault Equation says that, for a given H (so M^* and n_0 are constant parameters), the number of page faults decreases with increasing M for $M < M^*$, and remains constant as cold misses for $M \geq M^*$. Since $M^* = aH + b$ (Figure 8), the boundary $M = M^*$ is $H = \frac{M-b}{a}$. We thus get one guideline for heap sizing, as illustrated in Figure 11.

Recall that the workload cannot run with a heap size smaller than $H_{\min}$. For $H > H_{\min}$, a bigger heap would require less garbage collection. Since garbage collection varies linearly with n_0 (Figure 10), and Figure 9 shows that n_0 stops decreasing when $H > H_{\max}$, the heap should not grow beyond $H_{\max}$: the benefit to the mutator is marginal, but more work is created for the garbage collector. We thus get another guideline for heap sizing, as illustrated in Figure 12.

The two guidelines combine to give the Heap Sizing Rule (1) that is illustrated in Figure 2: Figure 12 requires $H > H_{\min}$. Therefore, for $M < aH_{\min} + b$, the diagonal line in Figure 11 does not apply, and H should be large to reduce the faults from garbage collection, so we get $H = H_{\max}$ from Figure 12. For $aH_{\min} + b < M < aH_{\max} + b$, H should be large to minimize garbage collection and its negative impact (delays, cache corruption, etc.), but the size should not exceed the diagonal line in Figure 11. For $M > aH_{\max} + b$, Figure 12 again sets $H = H_{\max}$; going beyond that would create unnecessary work for the garbage collector.

3.2. Experiments with static M

We first test the Heap Sizing Rule for a static M that is held fixed throughout the run of the workload. We wanted to compare the effectiveness of the Rule against previous work on heap sizing [2, 9, 10, 12]. However, we have no access to their implementation, some of which require significant changes to the kernel or mutator (see Section 4).

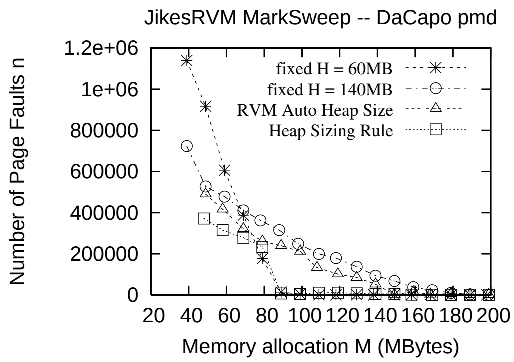
We therefore compare the Rule to `JikesRVM`'s heap sizing policy, which dynamically adjusts the heap size according to heap utilization during execution. This adjustment is done even if M is fixed, since an execution typically goes through phases, and its need for memory varies accordingly.

Figure 13(i) shows that, for `pmd` run with `MarkSweep`, `JikesRVM`'s automatic heap sizing indeed results in fewer faults than if H is fixed at 60MBytes or at 140MBytes for small M; for large M ($\geq$ 80MBytes), however, its dynamic adjustments fail to reduce the number of faults below that for H = 60MBytes.

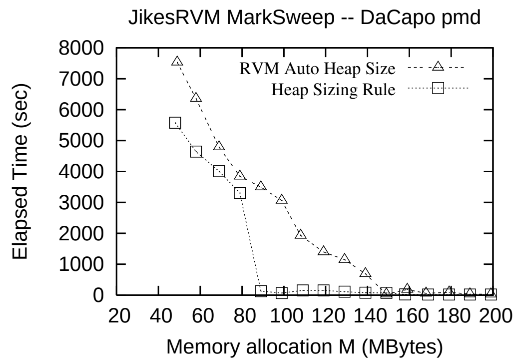
It is therefore not surprising that, although our Rule fixes H for a static M, it consistently yields less faults than `JikesRVM`; i.e. it suffices to choose an appropriate H for M, rather than adjust H dynamically according to `JikesRVM`'s policy. Notice that, around M = 80MBytes, page faults under the Rule drop sharply to just cold misses. This corresponds to the discontinuity in Figure 2 at $M = aH_{\min} + b$.

Since disks are much slower than processors, one expects page faults to dominate execution time. Figure 13(ii) bears this out: the relative performance in execution time between the two policies is similar to that in Figure 13(i). The cold miss segments in the two plots illustrate how, by trading less page faults for more garbage collection, the Rule effectively reduces execution time.

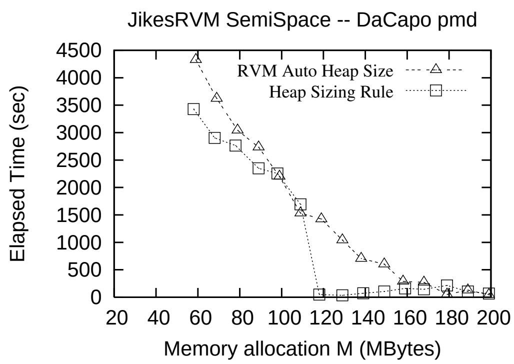
Figure 13(iii) and Figure 13(iv) show similar results for `pmd` run with `SemiSpace` and for `xalan` run with `MarkSweep`.



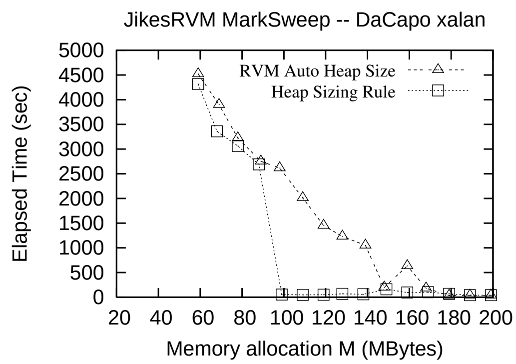
(i) Page faults (`MarkSweep` and `pmd`)



(ii) Execution time (`MarkSweep` and `pmd`)



(iii) Execution time (`SemiSpace` and `pmd`)



(iv) Execution time (`MarkSweep` and `xalan`)

Figure 13: **Comparison of the Heap Sizing Rule to `JikesRVM`'s dynamic heap sizing policy. M is fixed for each run. The steep drop for the Rule's data reflects the discontinuity in Figure 2.**

3.3. Experiments with dynamic M

We next test the Heap Sizing Rule in experiments where M is changing dynamically.

To do so, we modify the garbage collectors so that, after each collection, they estimate M by adding `Resident Set Size RSS` in `/proc/pid/stat` and free memory space `MemFree` in `/proc/meminfo` (the experiments are run on Linux). H is then adjusted according to the Rule.

To change M dynamically, we run a background process that first `mlock` enough memory to start putting pressure on the workload, then loop infinitely as follows:

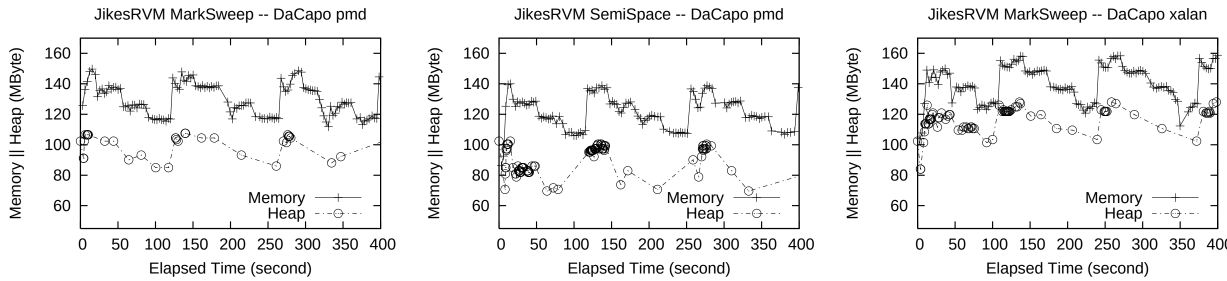


Figure 14: **How the Heap Sizing Rule adjusts H at each garbage collection when M varies dynamically. (To plot the points for M, we run a background process that measures M every 3 seconds.)**

```
repeat{
        sleep 30sec; mlock 10MBytes;
        sleep 30sec; mlock 10MBytes;
        sleep 30sec; mlock 10MBytes;
        sleep 30sec; munlock 30MBytes;
}
```

To prolong the execution time, we run the mutator 5 times in succession.

Figure 14 shows how H responds to such changes for three of the workloads in our experiments. Since we do not modify the operating system to inform the garbage collector about every change in M, adjustments in H occur less frequently (only when there is garbage collection). Consequently, there are periods during which H is different from that specified by the Rule for the prevailing M.

Even so, Table 2 shows that page faults under the Rule is an order of magnitude less than those under `JikesRVM`'s automatic sizing policy. The gap for execution time is similar. These indicate the effectiveness of the Rule for dynamic heap sizing.

3.4. Experiments with multiple mutators

We now examine the Rule's efficacy in a multiprogramming mix, and compare it to Poor Richard's Memory Manager (PRMM), which is specifically designed to manage garbage collection when memory is shared [24].

PRMM is *paging-aware*, in that each process tracks the number of page faults it has caused. It has three strategies: *Communal*, *Leadered* and *Selfish*. The Communal and Leadered strategies require processes to commu-

		MarkSweep pmd	SemiSpace pmd	MarkSweep xalan
page faults	RVM	425828	680575	352338
	Rule	36228	36470	64580
execution time (sec)	RVM	4762	8362	4202
	Rule	419	404	761

Table 2: **Automatic heap sizing when M changes dynamically: a comparison of JikesRVM's default policy and our Heap Sizing Rule.**

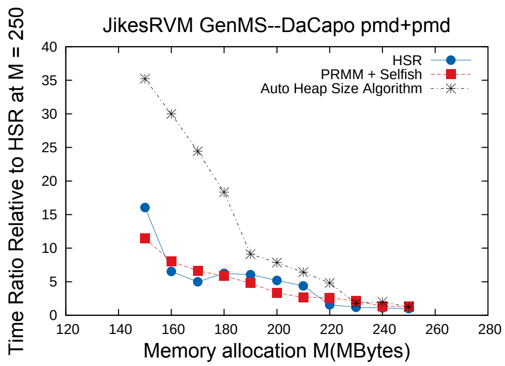
nicate with a "whiteboard" in shared memory, and thus coordinate their garbage collection. For the Selfish strategy, processes independently initiate whole-heap garbage collection if they detect memory pressure. Hertz et al.'s experiments show that Leadered and Selfish outperform Communal, and Selfish is comparable to Leadered. Our experiments therefore compare our Rule to the Selfish strategy, and similarly use GenMS and GenImmix as garbage collectors and pmd and xalan as mutators.

Figure 15 presents results for the execution for two concurrent mutators. The plots show that, as in Figure 13 and Table 2, our Rule outperforms JikesRVM's dynamic heap sizing policy. They also show that our Rule is comparable to the Selfish strategy, except for Figure 15(ii), where our Rule is noticeably worse for small M but better for large M.

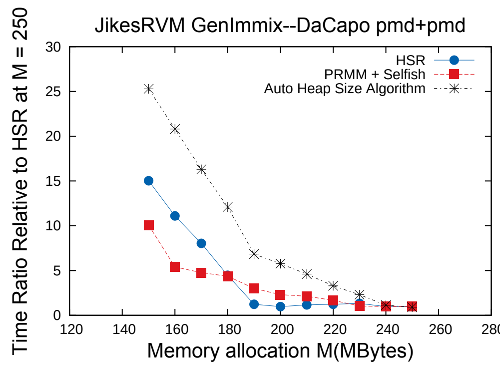
Why does our Rule not do as well as Poor Richard's in some cases? The reason lies in Fig. 14: In a multiprogramming mix, the operating system may change M more frequently than garbage collection; consequently, the Rule is often violated, thus causing extra page faults. For the Rule to work effectively in a multiprogramming mix, it is therefore necessary that the garbage collector keep track of changes in M. If H deviates too far from the heap size specified by the Rule for M, this should trigger garbage collection and heap resizing. Note that details for this trigger mechanism concern effective application of the Rule, which is an issue separate from this paper's focus on the correctness of the Rule.

4. Related Work

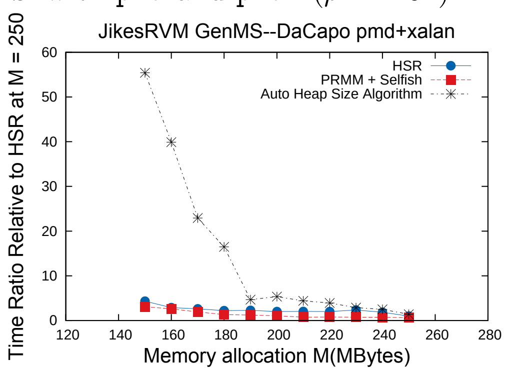
Early work on the interaction between garbage collection and virtual memory were for languages like Lisp and Standard ML. For example, Moon



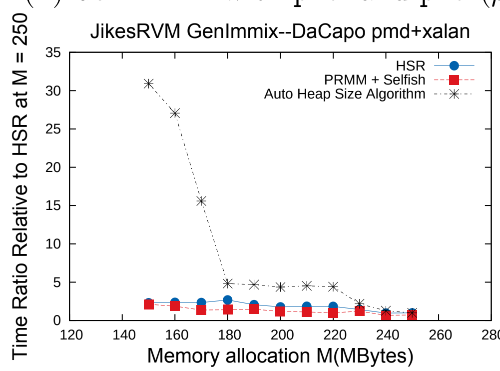
(i) `GenMS` with `pmd` and `pmd` ($\rho = 1.02$)



(ii) `GenImmix` with `pmd` and `pmd` ($\rho = 1.06$)



(iii) `GenMS` with `pmd` and `xalan` ($\rho = 1.99$)



(iv) `GenImmix` with `pmd` and `xalan` ($\rho = 1.49$)

Figure 15: **Comparison of Heap Sizing Rule (HSR) to `JikesRVM`'s dynamic heap sizing policy and Poor Richard's Selfish strategy. The elapsed time depends on how many times each mutator runs (see Appendix A). We therefore follow Hertz et al. and report the ratio of elapsed time instead, using as base the run time for our Rule at $M = 250$MBytes. ρ is the average ratio of HSR runtime to Selfish runtime.**

and Cooper et al. observed that garbage collection can cause disk thrashing [25, 7].

Recent work is mostly focused on Java. Kim and Hsu noted that a garbage collector can generate more hardware cache misses than the mutator, and interfere with the latter's temporal locality [3]. They pointed out that (where possible) heap size should not exceed available main memory. Also, since most objects are short-lived, a heap page evicted from physical memory may only contain dead objects; so, rather than incur unnecessary IO through garbage collection for such pages, it may be better to grow the heap. Their observation is reflected in our Heap Sizing Rule at $M = aH_{\min} + b$, where a reduction in M prompts a discontinuity in H, raising it to $H_{\max}$ (see Figure 2).

Yang et al. [4] had observed a linearity similar to Figure 8. Their dynamic heap sizing algorithm relies on changes to the virtual memory manager to track recent page accesses, shuffle pages between hot and cold sets, construct LRU histograms for mutator and collector references, decay the histograms exponentially to reflect phase changes, etc. The hot set size is adjusted through weighting with minor faults and setting targets with constants (1% and 0.2).

In follow-up work [10], the authors modified and extended their system to handle generational collectors, provide per-process and per-file page management, etc.

Rather than reduce page faults by sizing the heap, Hertz et al. [9] designed a garbage collector to eliminate such faults (assuming memory allocation is sufficiently large). It requires extending the virtual memory manager to help the collector bookmark evicted pages with summary information, so it can avoid recalling them from disk.

Another possibility is to modify the mutator. Zhang et al. [12] proposed a PAMM controller that uses program instrumentation to extract phase information that is combined with data (heap size and fault count) polled from the virtual machine and operating system, and thus used to trigger garbage collection. Various constants (step size, soft bound, etc.) are used in a binary search for an optimal heap size. In contrast, our Heap Sizing Rule is in closed-form, and does not require an iterative search.

Whereas PAMM uses the mutator's phase boundary to trigger garbage collection, Grzegorczyk et al. used a stall in memory allocation as the signal [2], thus delaying collection till when it is actually necessary. Like PAMM, their IV heap sizing is also iterative, using an additive constant to grow the heap and a multiplicative constant to shrink it.

Another iterative policy for growing the heap was proposed by Xian et al. for the `HotSpot` virtual machine [14]. They set a threshold (75%) for H/M to switch between two growth rates. This proposal was in the context of their study of which factors cause throughput degradation, how they do so, and what can be done.

There is recent interest in heap sharing among multiple applications. Choi and Han proposed a scheme for dynamically managing heap share with hardware support and application profiling [26]. Sun et al.'s Resonant Algorithm is more narrowly focused on determining a heap partition that equalizes collection frequencies among applications [27].

Tran et al. have demonstrated how the Page Fault Equation can be used to dynamically partition a database buffer among tasks [22], and we believe our heap-aware Equation (6) can be similarly used for heap sharing.

5. Conclusion

Many programming languages and systems now provide garbage collection. Garbage collection increases programmer productivity but degrades application performance. This run-time effect is the result of interaction between garbage collection and virtual memory. The interaction is sensitive to heap size H, which should therefore be adjusted to suit dynamic changes in main memory allocation M.

We present a Heap Sizing Rule (Figure 2) for how H should vary with M. It aims to first minimize page faults (Figure 11), then garbage collection (Figure 12), as disk retrievals impose a punishing penalty on execution time. Comparisons with `JikesRVM`'s automatic heap sizing policy shows that the Rule is effective for both static M (Figure 13), dynamic M (Table 2) and in a multi-mutator mix (Figure 15). This Rule can thus add a run-time advantage to garbage-collected languages: execution time can be improved by exchanging less page faults for more garbage collection (Figure 13(i) and Figure 13(ii)).

The Rule is based on a Heap-Aware Page Fault Equation (6) that models the number of faults as a parameterized function of H and M. The Equation fits experimental measurements with a variety of garbage collectors and mutators (Figures 4, 5, 6), thus demonstrating its universality. Its parameters have interpretations that relate to the garbage collection algorithm and the mutators' memory requirements (Section 2.4). We also demonstrate how this Equation can be used to examine the interaction between garbage collection and virtual memory through a relationship among these parameters (Section 2.5).

The Rule is based on four parameters (a, b, H_{min} and H_{max}) that characterize the workload (garbage collector and mutator). Once these values are known, the Rule just needs the value of M. In particular, there is no need to change the operating system, and a hot swap of garbage collector just requires a change in the parameter values.

Our experiments suggest that choosing H to suit M is more effective than changing H to suit mutator phases. In any case, if the mutator has multiple phases with very different behavior, one can make the Rule itself adaptive by changing the parameter values to suit each phase. The Rule does not require "stop-the-world" garbage collection; in principle, it can be implemented in any garbage collector that allows dynamic heap sizing.

Our application of the Equation is focused on M^*. Although M^* is partly determined by the rest of the page fault curve, we have not used the latter. Tran et al. have demonstrated how the curve, in its entirety, can be applied to fairly partition memory and enforce performance targets when there is memory pressure among competing workloads [22]. In future work, we plan to demonstrate a similar application of the Equation for dynamic heap sharing.

Acknowledgement

We are most grateful to Prof. Matthew Hertz for his generous help with Poor Richard's Memory Manager. We also thank him and anonymous researchers in `JikesRVM`'s mailing list for their suggestions on reducing the nondeterminism in the experiments.

APPENDIX

Appendix A. Experimental set-up

The hardware for our experiments has an Intel Core 2 Duo CPU E6550 (2.33GHz each), with 4MByte L2 cache, 3.25GByte RAM, 8GByte swap space, and a 250GByte disk that has 11msec average seek time and 7200rpm spindle speed. The operating system is `linux-2.6.20-15-generic SMP`, and the page size is 4KBytes.

To reduce noise and nondeterminism [28], we run the experiments in single-user mode, disconnect the network and shut down unnecessary background processes. We set `lowmem_reserve_ratio=1` to reserve the entire low memory region for the kernel, so it need not compete with our workload for memory.

Linux does not provide any per-process memory allocation utility. Like previous work [10], we therefore vary M by running a background process that claims a large chunk of memory, pins those pages so the virtual memory manager allocates RAM space for them, then `mlock` them to prevent eviction. The remaining RAM space (after `lowmem_reserve_ratio=1` and `mlock`) is shared by our workload and the background processes (e.g. device drivers). There is thus some imprecision in determining M, and M^o corrects for this inaccuracy.

After each run of a mutator, we clear the page cache so the cold misses n^* remains constant. There is also a pause for the system to settle into an equilibrium before we run the next experiment.

Our first experiments used `HotSpot` Java Virtual Machine [13]. However, to facilitate tests with various garbage collectors, we switched to the `Jikes` Research Virtual Machine (Version 3.0.1) [6]. The collectors `GenMS`, `MarkSweep` and `SemiSpace` were chosen from its `MMTk` toolkit as representatives of the three major techniques for collecting garbage. (There is another technique that *slides* the live objects; however, previous work has shown that `Mark-Compact`, which uses this technique, does not perform as well as `MarkSweep` and `SemiSpace` [29].) For the multi-mutator experiments, we follow Hertz et al. in using `GenMS` and `GenImmix` [24].

Our first mutator was `ipsixql` from the `DaCapo` benchmark suite [5]. However, for the `JVM/ipsixql` combination, measurements show that heap size has little effect on page faults. We then limited the other experiments to `pmd` (a source code analyzer for Java), `xalan` (an XSLT transformer for XML documents) [28], and `fop` (which parses and formats an XSL-FO file, and generates a PDF file).

We prefer a larger range of mutators, but are limited by the need to make comparisons between garbage collectors, space constraint for presenting the results, and time constraint for finishing the experiments. The workloads are IO-intensive, so a set of data points like Figure 1 can take two or three days to generate.

`JikesRVM` uses adaptive compilation, which makes its execution nondeterministic. We therefore (like previous work [10]) logged the adaptive compilation of 6 runs of each workload, select the run with best performance, then direct the system to compile methods according to the log from that run. Such a replay of the compilation is known to yield performance that is similar to an adaptive run.

For the multi-mutator and sensitivity experiments (Sections 3.4 and Appendix C), we use an Intel Core i5 machine with 4MByte L2 cache, 4GByte RAM and a 500GByte disk. It runs a vanilla Linux 2.6.36-rc3 with 4KByte pages. We make no change to the operating system, and implemented the Heap Sizing Rule with some 50 lines of code in `MMTk/src/org/mmtk/utility/heap/HeapGrowthManager.java`.

Appendix B. Parameter calibration

For the Page Fault Equation (2), an experiment with a fixed H yields a data set

$$\Omega_k^n = \{\langle M_i, n_i\rangle \mid M_{i-1} < M_i \text{ for } i = 1, \ldots, k\}.$$

We use regression to fit the equation to Ω_k^n, and thus calibrate the parameters M^*, M^o and n_0. (The cold misses n^* can be determined separately, by running the workload at some sufficiently large M.)

Equation (2) has an equivalent linear form

$$M = (M^* - M^o)x + M^o \text{ where } x = \left(\frac{n + n_0}{n^* + n_0} - 1 + \frac{n^* + n_0}{n + n_0}\right)^{-1}. \quad \text{(B.1)}$$

Software for linear regression is readily available, but there are three issues:

(i) Transforming Ω_k^n into a corresponding

$$\Omega_k^x = \{\langle M_i, x_i\rangle \mid M_{i-1} < M_i \text{ for } i = 1, \ldots, k\}$$

requires a value for n_0.

(ii) The nontrivial part of the equation is valid for $M \leq M^*$ only, so the flat tail in Ω_k^n must be trimmed off. There is no obvious way of trimming a point $\langle M_i, n_i\rangle$ since M^* is unknown, and a point may belong to that tail although n_i is driven from n^* by some statistical fluctuation.

(iii) Although regression is done with $\langle M_i, x_i\rangle$ data, the objective is to get a good fit for the original $\langle M_i, n_i\rangle$ data.

These issues are addressed in the calibration algorithm in Figure B.16.

In practice, calibration may be done in two ways:

(Offline) Some workloads are repetitive, so that calibration can be done with data from previous runs. Examples may include batch workloads, transaction processing and embedded applications.

(Online) Many workloads are ad hoc or vary with input [30], so on-the-fly calibration is necessary.

$k' \leftarrow k$;   //start with the entire data set
repeat{
  $n_0 \leftarrow -n^* + 1$;
  for each candidate n_0 {   //(i) iteratively search for best n_0 value

    $x_i \leftarrow (\frac{n_i+n_0}{n^*+n_0} - 1 + \frac{n^*+n_0}{n_i+n_0})^{-1}$ for $i = 1, \ldots, k'$;
    $\Omega^x_{k'} \leftarrow \{\langle M_i, x_i \rangle \mid i = 1, \ldots, k'\}$;
    fit $\Omega^x_{k'}$ with Equation (B.1);
    record sum of square errors SSE for Ω^n_k;   //(iii) instead of $\Omega^x_{k'}$
    if SSE decreases then increment n_0
      else adopt previous n_0 value and corresponding M^o and M^* values;

  }
  record coefficient of determination R^2 for Ω^n_k;
  if R^2 increases then $k' \leftarrow k' - 1$;   //(ii) trim off the last point
            else exit with n_0, M^o and M^* from previous k' value
}

Figure B.16: **Algorithm for calibrating the parameters through linear regression.**

The algorithm in Figure B.16 can be used for offline calibration. For online calibration, one cannot wait to measure the number of page faults n for the entire run, so we need to use another version of the Page Fault Equation, namely

$$P^{\text{miss}} = \begin{cases} P^* & \text{for } M \geq M^* \\ \frac{1}{2}(K + \sqrt{K^2 - 4})(P^* + P_0) - P_0 & \text{for } M < M^* \end{cases}$$

where P^{miss} is the probability that a page reference results in a page fault.

Tran et al. have demonstrated how the parameters P_0, M^*, etc. can be calibrated dynamically, using moving windows for measurements of P^{miss}, if this is the probability of missing a database buffer [22].

In our case, measuring P^{miss} is difficult because page hits are not seen by the operating system. We are aware of only two techniques for making such measurements with software [10, 31]; both require significant changes to the operating system. They also use the Mattson stack [32], which assumes an LRU-like inclusion property that disagrees with the looping behavior in most garbage collectors.

garbage collector	mutator	a	b	$H_{\min}$	$H_{\max}$
`GenMS`	`pmd`	0.77	88.0	36	143
	`xalan`	0.78	78.0	32	100
	`fop`	0.74	68.9	28	84
`MarkSweep`	`pmd`	1.18	52.9	48	131
	`xalan`	1.00	58.9	32	165
	`fop`	1.01	55.0	35	96
`SemiSpace`	`pmd`	0.75	63.5	64	106
	`xalan`	0.71	61.9	44	114
	`fop`	0.73	61.5	52	110

Table C.3: **How the parameters in the Rule vary with garbage collector and mutator.**

The difficulty in measuring P^{miss} with software has led to designs for measurement with hardware [33, 10], but implementing such designs are harder still. However, a major hardware vendor is extending their metering architecture to memory usage [34, 35], so P^{miss} measurements with built-in hardware may be possible soon.

Appendix C. Parameter sensitivity

Our Heap Sizing Rule (1) has four parameters: a, b, $H_{\min}$ and $H_{\max}$. How sensitive are they to the workload, and what is their impact on performance?

Table C.3 shows how these four parameters vary with different garbage collectors and mutators. (The values for $H_{\max}$ are somewhat different from those in Fig. 9 because the experiments were run with different hardware and software.) We see that, if we fix the garbage collector, then the values for a are similar for different mutators. This is consistent with the interpretation in Section 2.4 that a is determined by the garbage collector. For a garbage collector that is heavily used, one can therefore collect a lot of data to accurately determine a.

The other parameters (b, $H_{\min}$ and $H_{\max}$) have values that depend on the mutator. The value for b determines the H-intercept for the diagonal in the Rule (Figure 2). An overestimate of this parameter for space overhead will shift the diagonal down; the Rule would then underestimate H and cause garbage collection frequency to increase, but there will be no increase in page faults beyond cold misses (see Figure 11). In using the Rule, one should therefore prefer an overestimate of b, rather than an underestimate.

Similarly, underestimating $H_{\max}$ would lower the horizontal line in Figure 2, and reduce H as determined by the Rule. For large M, garbage collection would be more frequent but no page faults are added. For $M < aH_{\min}+b$, the smaller H would cause more page faults (see Figure 4). Indeed, the big increase in page faults from cold misses shown in Figure 13 happens at the discontinuity in the Rule at $M = aH_{\min} + b$.

The discontinuity happens where M is too small to accommodate the minimal footprint $aH_{\min}+b$, so underestimating $H_{\min}$ is not an option. Overestimating $H_{\min}$ will cause the jump in page faults from cold misses to occur at some $M > aH_{\min} + b$, but Figure 13 shows that the resulting performance would still be better than using, say, `JikesRVM`'s policy.

References

[1] Y. C. Tay, X. R. Zong, A page fault equation for dynamic heap sizing, in: Proc. 2010 Joint WOSP/SIPEW Int. Conf. Performance Engineering, pp. 201–206.

[2] C. Grzegorczyk, S. Soman, C. Krintz, R. Wolski, Isla Vista heap sizing: using feedback to avoid paging, in: CGO '07: Proceedings of the International Symposium on Code Generation and Optimization, IEEE Computer Society, Washington, DC, USA, 2007, pp. 325–340.

[3] J.-S. Kim, Y. Hsu, Memory system behavior of Java programs: methodology and analysis, SIGMETRICS Perform. Eval. Rev. 28 (2000) 264–274.

[4] T. Yang, M. Hertz, E. D. Berger, S. F. Kaplan, J. E. B. Moss, Automatic heap sizing: taking real memory into account, in: ISMM '04: Proceedings of the 4th International Symposium on Memory Management, ACM, New York, NY, USA, 2004, pp. 61–72.

[5] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, 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, B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, B. Wiedermann, The DaCapo benchmarks: Java benchmarking development and analysis, SIGPLAN Not. 41 (2006) 169–190.

[6] B. Alpern, S. Augart, S. M. Blackburn, M. A. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. J. Fink, D. Grove, M. Hind, K. S. McKinley, M. F. Mergen, J. E. B. Moss, T. A. Ngo, V. Sarkar, M. Trapp, The Jikes Research Virtual Machine project: Building an open-source research community, IBM Systems Journal 44 (2005) 399–418.

[7] D. A. Moon, Garbage collection in a large LISP system, in: LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming, ACM, New York, NY, USA, 1984, pp. 235–246.

[8] Y. C. Tay, M. Zou, A page fault equation for modeling the effect of memory size, Perform. Eval. 63 (2006) 99–130.

[9] M. Hertz, Y. Feng, E. D. Berger, Garbage collection without paging, SIGPLAN Not. 40 (2005) 143–153.

[10] T. Yang, E. D. Berger, S. F. Kaplan, J. E. B. Moss, CRAMM: virtual memory support for garbage-collected applications, in: OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation, pp. 103–116.

[11] T. Brecht, E. Arjomandi, C. Li, H. Pham, Controlling garbage collection and heap growth to reduce the execution time of Java applications, ACM Trans. Program. Lang. Syst. 28 (2006) 908–941.

[12] C. Zhang, K. Kelsey, X. Shen, C. Ding, M. Hertz, M. Ogihara, Program-level adaptive memory management, in: ISMM '06: Proceedings of the 5th International Symposium on Memory Management, ACM, New York, NY, USA, 2006, pp. 174–183.

[13] JavaSoft, J2SE 1.5.0 documentation: Garbage collector ergonomics, `http://java.sun.com/j2se/1.5.0/docs/guide/vm/gc-ergonomics.html`, 2010.

[14] F. Xian, W. Srisa-an, H. Jiang, Investigating throughput degradation behavior of Java application servers: a view from inside a virtual machine, in: PPPJ '06: Proc. Int. Symp. on Principles and Practice of Programming in Java, pp. 40–49.

[15] Oracle, Oracle JRockit JVM, `http://www.oracle.com/technology/products/jrockit/index.html`, 2010.

[16] S. M. Blackburn, P. Cheng, K. S. McKinley, Oil and water? High performance garbage collection in Java with MMTk, in: ICSE '04: Proceedings of the 26th International Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA, 2004, pp. 137–146.

[17] Novell, NetWare 6: Optimizing Garbage Collection, `http://www.novell.com/documentation/nw6p/?page=/documentation/nw6p/smem_enu/data/ht0rfodz.html`, 2010.

[18] T. Printezis, Hot-swapping between a mark&sweep and a mark&compact garbage collector in a generational environment, in: Java Virtual Machine Research and Technology Symposium, USENIX Association, Berkeley, CA, USA, 2001, pp. 20–20.

[19] L. A. Belady, A study of replacement algorithms for virtual storage computer, IBM System J. 5(2) (1966) 78–101.

[20] W. W. Hsu, A. J. Smith, H. C. Young, I/O reference behavior of production database workloads and the TPC benchmarks — an analysis at the logical level, ACM Trans. Database Syst. 26 (2001) 96–143.

[21] A. J. Storm, C. Garcia-Arellano, S. S. Lightstone, Y. Diao, M. Surendra, Adaptive self-tuning memory in DB2, in: VLDB '06: Proceedings of the 32nd International Conference on Very Large Data Bases, VLDB Endowment, 2006, pp. 1081–1092.

[22] D. N. Tran, P. C. Huynh, Y. C. Tay, A. K. H. Tung, A new approach to dynamic self-tuning of database buffers, ACM Trans. Storage 4 (2008) 1–25.

[23] S. Soman, C. Krintz, D. F. Bacon, Dynamic selection of application-specific garbage collectors, in: ISMM '04: Proceedings of the 4th International Symposium on Memory Management, ACM, New York, NY, USA, 2004, pp. 49–60.

[24] M. Hertz, S. Kane, E. Keudel, T. Bai, C. Ding, X. Gu, J. E. Bard, Waste not, want not: resource-based garbage collection in a shared environment, in: Proc. 2011 Int. Symp. on Memory Management, pp. 65–76.

[25] E. Cooper, S. Nettles, I. Subramanian, Improving the performance of SML garbage collection using application-specific virtual memory management, in: Proc. 1992 ACM Conf. on LISP and Functional Programming, pp. 43–52.

[26] Y. Choi, H. Han, Shared heap management for memory-limited Java virtual machines, ACM Trans. Embed. Comput. Syst. 7 (2008) 1–32.

[27] K. Sun, Y. Li, M. Hogstrom, Y. Chen, Sizing multi-space in heap for application isolation, in: Dynamic Languages Symposium, ACM, New York, NY, USA, 2006, pp. 647–648.

[28] S. M. Blackburn, K. S. McKinley, R. Garner, C. Hoffmann, A. M. Khan, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. L. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, B. Wiedermann, Wake up and smell the coffee: evaluation methodology for the 21st century, Commun. ACM 51 (2008) 83–89.

[29] S. M. Blackburn, K. S. McKinley, Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance, in: Proc. PLDI 2008, pp. 22–32.

[30] F. Mao, E. Z. Zhang, X. Shen, Influence of program inputs on the selection of garbage collectors, in: VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, ACM, New York, NY, USA, 2009, pp. 91–100.

[31] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, S. Kumar, Dynamic tracking of page miss ratio curve for memory management, SIGPLAN Not. 39 (2004) 177–188.

[32] R. L. Mattson, J. Gecsei, D. R. Slutz, I. L. Traiger, Evaluation techniques for storage hierarchies, IBM System J. 9(2) (1970) 78–117.

[33] R. Azimi, L. Soares, M. Stumm, T. Walsh, A. D. Brown, Path: page access tracking to improve memory management, in: Proc. ISMM '07, pp. 31–42.

[34] R. Iyer, R. Illikkal, L. Zhao, D. Newell, J. Moses, Virtual platform architectures for resource metering in datacenters, SIGMETRICS Perform. Eval. Rev. 37 (2009) 89–90.

[35] V. Sekar, P. Maniatis, Verifiable resource accounting for cloud computing services, in: Proc. ACM Workshop on Cloud Computing Security Workshop, CCSW '11, pp. 21–26.