

# A Page Fault Equation for Dynamic Heap Sizing

(A shorter, 6-page version will appear in WOSP/SIPEW 2010)

Y.C. Tay  
National University of Singapore  
tay@acm.org

X.R. Zong  
Duke University  
xrz@cs.duke.edu

## 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  $M$ .

## 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. 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.

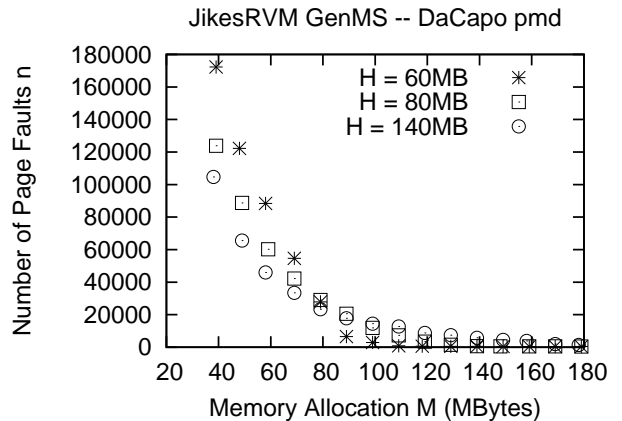


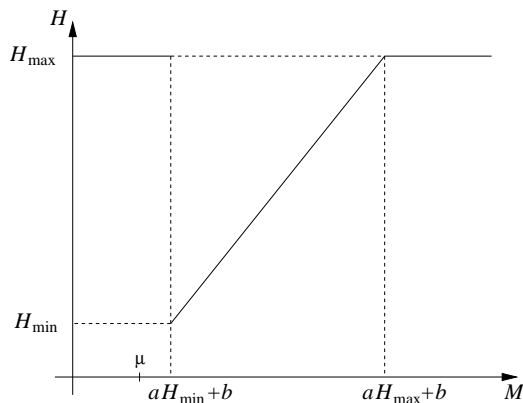
Figure 1: How heap size  $H$  and memory allocation  $M$  affect the number of page faults  $n$ . The garbage collector is GenMS and the mutator is pmd from the DaCapo benchmark suite.

### 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 [11, 16].

While a larger heap size can reduce garbage collection and its negative impact,  $H$  cannot be arbitrarily large either. A process will only get some fraction of main memory allocated to it. 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 causes more page faults than mutator execution when the heap extends beyond main memory [27].



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

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

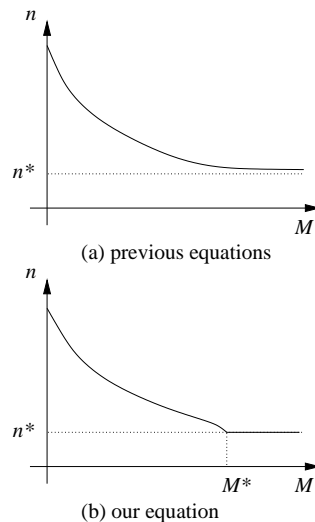
In the worst case,  $H > M$  can cause page thrashing [18]. Even if the situation is not so dire, page faults are costly — reading from disk is several orders of magnitude slower than from main memory — and should be avoided. It is thus clear that performance tuning for garbage-collected applications requires a careful choice of heap size.

Consider the case  $H = 140\text{MBytes}$  in Figure 1. If  $M = 50\text{MBytes}$ , then shrinking the heap to  $H = 60\text{MBytes}$  would trigger more garbage collection and double the number of page faults. If  $M = 110\text{MBytes}$ , however, setting  $H = 60\text{MBytes}$  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  $H$  cannot 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  $H = 60\text{MBytes}$  and  $M$  changes from  $110\text{MBytes}$  to  $50\text{MBytes}$ , performance would drop drastically.  $H$  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.



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

## 1.2 Our Contribution

The first contribution in this paper is the following equation that relates number of faults  $n$  to memory allocation  $M$  and heap size  $H$ :

**Heap-Aware Page Fault Equation**

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

$$\text{where } K = 1 + \frac{M^* + M^o}{M + M^o},$$

$$M^* = aH + b,$$

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

(1)

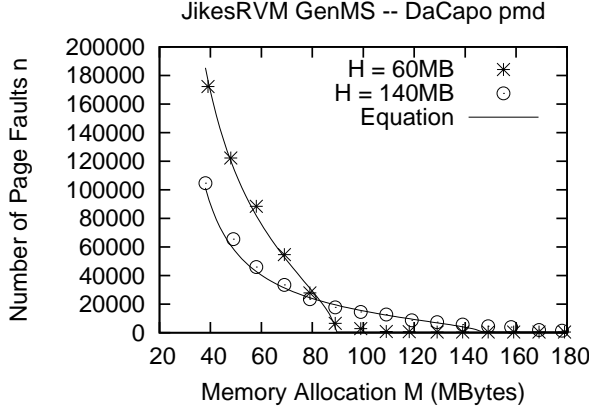
We refer to  $n^*$ ,  $n_0$ ,  $M^*$ ,  $M^o$ ,  $H_{\min}$ ,  $H_{\max}$ ,  $a$ ,  $b$ ,  $c$  and  $d$  as **parameters**; they encapsulate properties of the mutator, garbage collector and operating system. This equation is a refinement of the Page Fault Equation (for generic, possibly non-garbage-collected workloads) in previous work [23].

Our second contribution is the following

**Heap Sizing Rule:**

$$H = \begin{cases} \frac{M-b}{a} & \text{for } aH_{\min} + b < M < aH_{\max} + b \\ H_{\max} & \text{otherwise} \end{cases} \quad (2)$$

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 [12], tailor the page replacement policy [26], require notification when memory allocation stalls [11], track page references [27], measure heap utilization [1], watch allocation rate [8] or profile the application [28].



**Figure 4: The Page Fault Equation can fit data from Figure 1 for different heap sizes.**

For  $H = 60\text{MB}$ ,  $n^* = 480$ ,  $M^* = 89.0$ ,  $M^o = 14.8$  and  $n_0 = 64021$  ( $R^2 = 0.994$ ).

For  $H = 140\text{MB}$ ,  $n^* = 480$ ,  $M^* = 146.2$ ,  $M^o = 22.7$  and  $n_0 = 12721$  ( $R^2 = 0.993$ ).

Rule (2) is in closed-form, so there is no need for iterative adjustments [11, 15, 25, 26, 28]. If  $M$  changes dynamically, the rule can be used to tune  $H$  accordingly, in contrast to static command-line configuration with parameters and thresholds [3, 5, 15, 19].

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 [20].

### 1.3 An overview

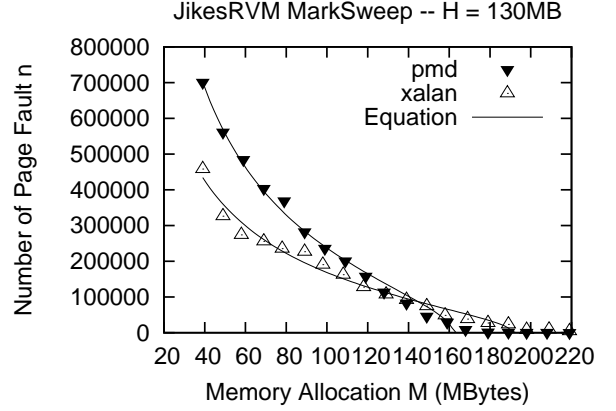
We begin in Section 2 by introducing the Page Fault Equation. We validate it for some garbage-collected workloads, then refine it to derive the heap-aware version (1). This refinement introduces new parameters, which we interpret and inter-relate.

Section 3 derives the Heap Sizing Rule (2) from the Equation, and presents experiments to show its effectiveness for static  $M$  and dynamic  $M$ .

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

## 2. HEAP-AWARE PAGE FAULT EQUATION

We first recall Tay and Zou's 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 (1), by expressing  $M^*$  and  $n_0$  in terms of  $H$ . This introduces new parameters  $a$ ,  $b$ ,  $c$  and  $d$ , 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.



**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  $n_0 = 12421$  ( $R^2 = 0.997$ ).

### 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

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

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

(3)

The values of  $n^*$ ,  $M^*$ ,  $M^o$  and  $n_0$  depend on the application, its input, the operating system, hardware configuration, etc. These four parameters are 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 [4, 13, 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 [23, 24]. 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.
- 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  $n_0$  [23]. Again,  $n_0$  can be positive or negative; geometrically, it controls the convexity of the page fault curve.

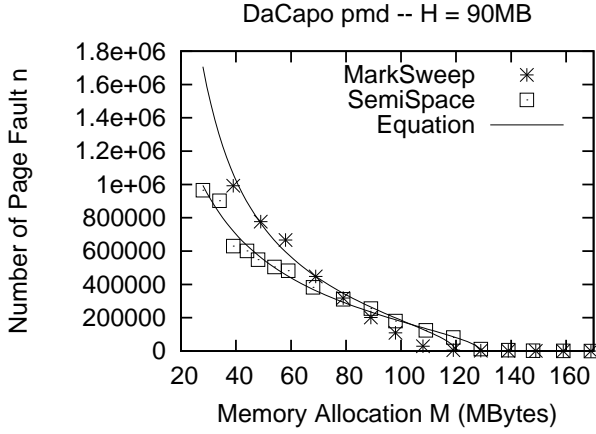


Figure 6: The Page Fault Equation can fit data for different garbage collectors.

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  $n_0 = 260659$  ( $R^2 = 0.992$ ).

## 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 [23]; 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 setup for this and subsequent experiments are given in Appendix A.1.

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.

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 A.2 provides details on how we use regression to fit Equation (3) 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.

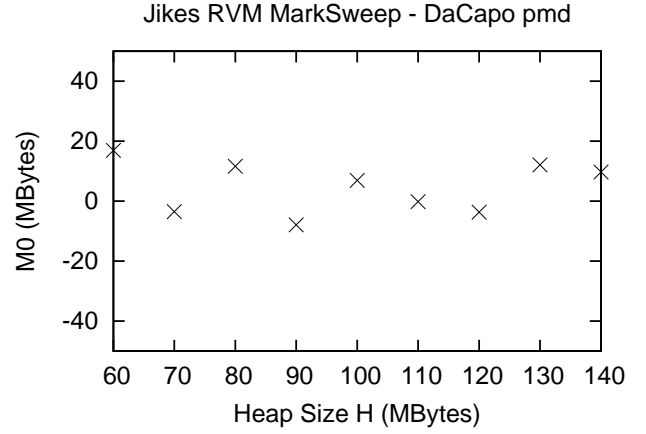


Figure 7:  $M^o$  values for `pmd` run with `MarkSweep`. They appear to fluctuate randomly.

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  $M^o$  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.

$$M^* = aH + b \quad \text{for some constants } a \text{ and } b. \quad (4)$$

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

$$n_0 = \begin{cases} cH + d & \text{for } H < H_{\max} \\ cH_{\max} + d & \text{for } H \geq H_{\max} \end{cases} \quad (5)$$

for some constants  $c$ ,  $d$  and  $H_{\max}$ . Equations (3), (4) and (5) together constitute the Heap-Aware Page Fault Equation (1).

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.

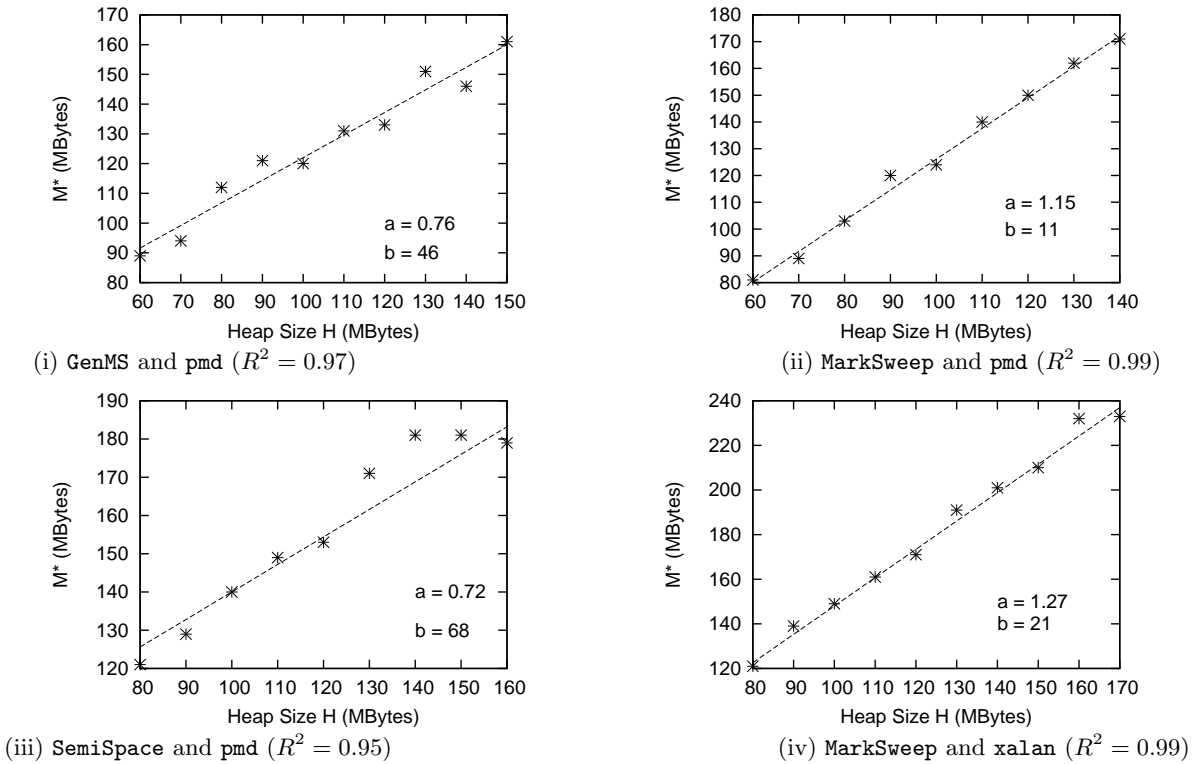


Figure 8:  $M^*$  values vary linearly with  $H$ .

## 2.4 Interpreting the new parameters

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  $R$  to be the minimum real memory required to run an application without substantial paging [27]. 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  $R$  is approximately our  $M^*$ , so Equation (4) 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 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  $\langle n_0, N_{GC} \rangle$  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 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

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

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  $N'_{GC} + N_{GC}$ ; this should be a constant in our experiment since we fixed the nursery size (at 10MBytes), regardless of  $H$ . Table 1 shows that, indeed,  $N'_{GC} + N_{GC}$  is almost constant as  $H$  varies from 60MBytes to 150MBytes.

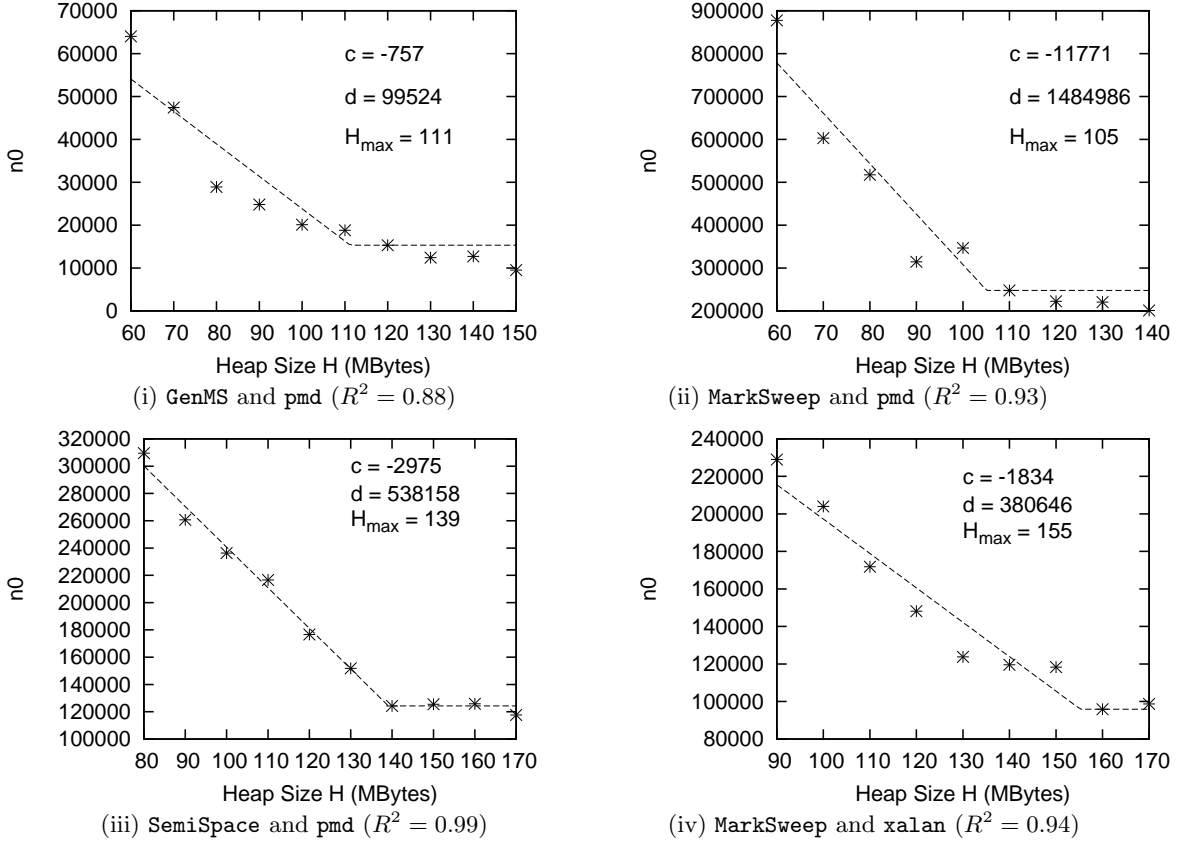


Figure 9:  $n_0$  decreases linearly with  $H$ , then flattens out.

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
$N'_{GC} + N_{GC}$	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.

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

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

as shown in Fig. 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  $M$ . Now

$$\begin{aligned} \frac{dn}{dH} &= \frac{\partial n}{\partial n_0} \frac{dn_0}{dH} + \frac{\partial n}{\partial M^*} \frac{dM^*}{dH} \\ &= \left( \frac{K + \sqrt{K^2 - 4}}{2} - 1 \right) \frac{dn_0}{dH} \\ &\quad + \frac{1}{2} \left( 1 + \frac{K}{\sqrt{K^2 - 4}} \right) \frac{n^* + n_0}{M + M^o} \frac{dM^*}{dH} \end{aligned} \quad (6)$$

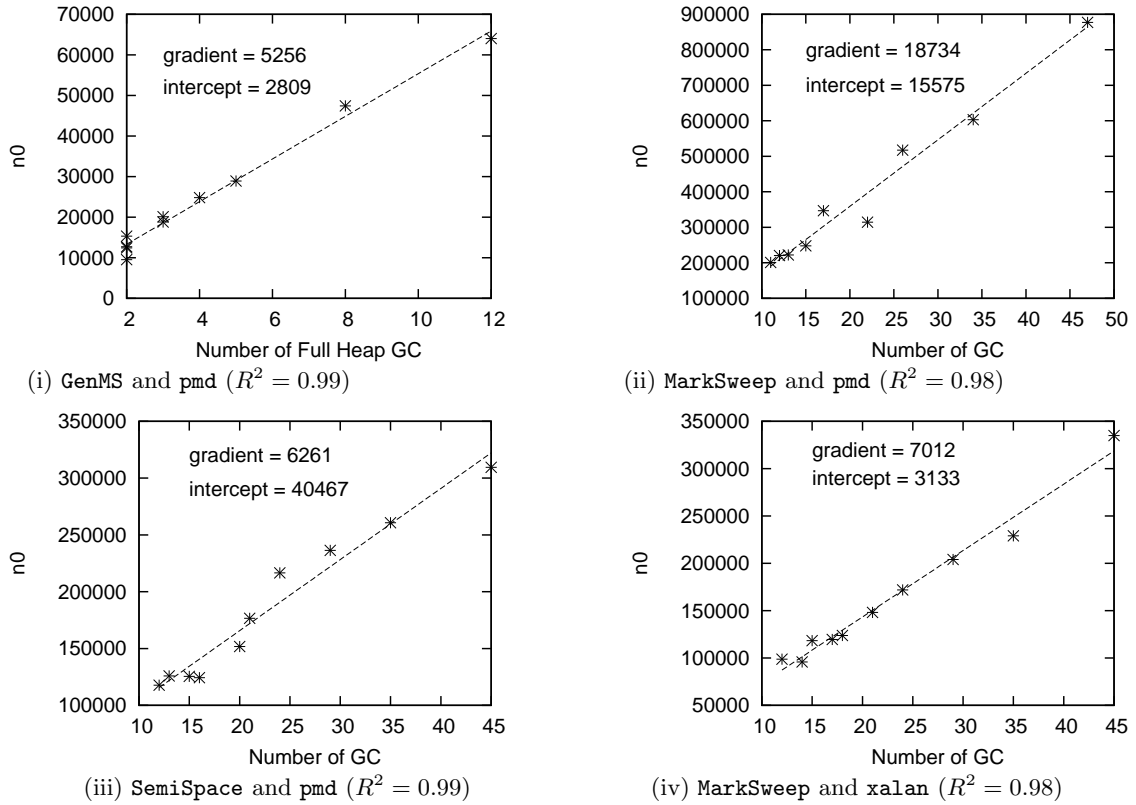
It is obvious that  $\frac{\partial n}{\partial n_0}$  and  $\frac{\partial n}{\partial M^*}$  are both positive, while Figure 9 and Figure 8 show that  $\frac{dn_0}{dH} \leq 0$  and  $\frac{dM^*}{dH} > 0$ . In other words, the page fault curve is largely dominated by the garbage collector for small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

## 2.5 A lower bound for $H$

The two opposing effects in Equation (6) cancel when

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

in Figure 1,  $\mu \approx 80$ MBytes. A heap cannot be arbitrarily small; there is a smallest heap size such that, for any smaller



**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.

$H$ , the workload will run out of memory before completion. There is therefore a bound

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

Now,  $\mu$  partitions the page fault curve so that

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

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, 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  $\mu$  from Equation (6); 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.

#### 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  $M$  for  $M < M^*$ , and remains constant as cold misses for  $M \geq M^*$ . Since  $M^* = aH + b$ , 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 (2) that is illustrated in Figure 2

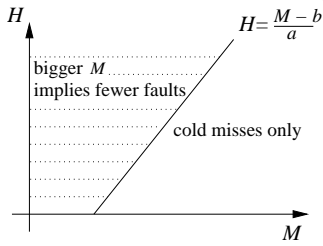


Figure 11: Guideline for heap sizing from the Heap-Aware Page Fault Equation (1).

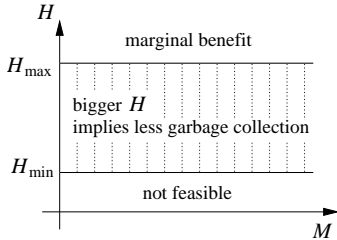


Figure 12: Guideline for heap sizing from Figure 9.

### 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 [11, 12, 26, 28]. 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  $M$  ( $\geq 80$ MBytes), however, its dynamic adjustments fail to reduce the number of faults below that for  $H = 60$ MBytes.

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  $M = 80$ MBytes, 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`.

		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.

### 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:

```
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.

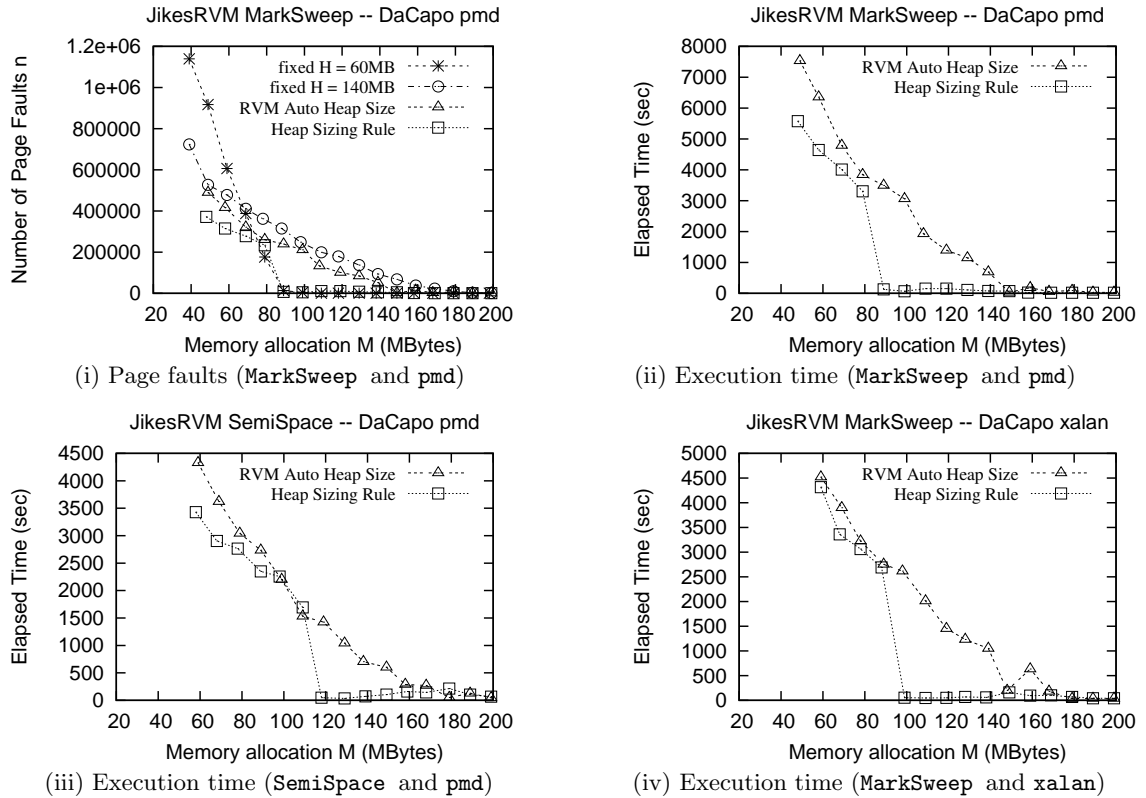
Figure 14 shows how  $H$  responds to such changes for three of the workloads in our experiments. Since we do not modify the operating system to inform the garbage collector about every change in  $M$ , adjustments in  $H$  occur less frequently (only when there is garbage collection). Consequently, 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.

## 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 and Cooper et al. observed that garbage collection can cause disk thrashing [10, 18].

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 [16]. 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



**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.**

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. [27] 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 [26], 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. [12] 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. [28] proposed a PAMM controller that uses program in-

strumentation 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, Grzegorzczuk et al. used a stall in memory allocation as the signal [11], 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 [25]. 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 [9]. Liu et al.’s Resonant Algorithm is more narrowly focused on determining a heap partition that equalizes collection frequencies among applications [22].

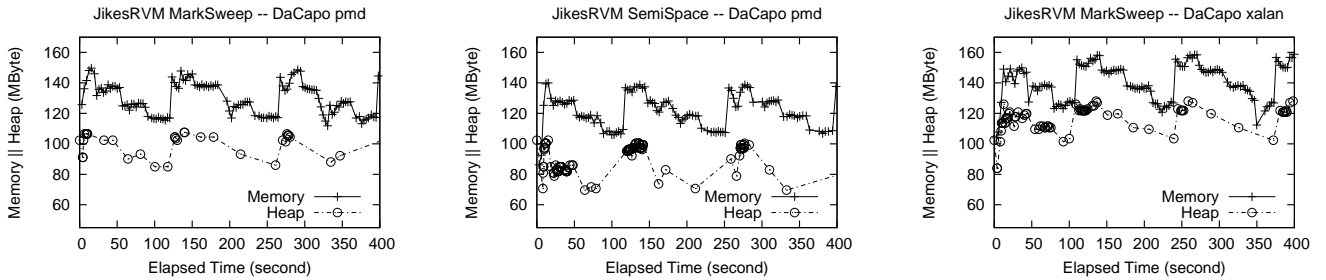


Figure 14: How the Heap Sizing Rule adjusts  $H$  at each garbage collection when  $M$  varies dynamically. (To plot the points for  $M$ , we run a background process that measures  $M$  every 3 seconds.)

Tran et al. have demonstrated how the Page Fault Equation can be used to dynamically partition a database buffer among tasks [24], and we believe our heap-aware Equation (1) can be similarly used for heap sharing.

## 5. CONCLUSION

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) and dynamic  $M$  (Table 2). 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 (1) 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).

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. In future work, we plan to demonstrate a similar application of the Equation for dynamic heap sharing.

## 6. ACKNOWLEDGMENTS

We thank Prof. Matthew Hertz and anonymous researchers in JikesRVM's mailing list for their suggestions on reducing

the nondeterminism in the experiments.

## 7. REFERENCES

- [1] 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, and M. Trapp. The Jikes Research Virtual Machine project: Building an open-source research community. *IBM Systems Journal*, 44(2):399–418, 2005.
- [2] R. Azimi, L. Soares, M. Stumm, T. Walsh, and A. D. Brown. Path: page access tracking to improve memory management. In *ISMM '07: Proceedings of the 6th international symposium on Memory management*, pages 31–42, New York, NY, USA, 2007. ACM.
- [3] BEA WebLogic. JRockit: Java for the enterprise. *White Paper (online)*.
- [4] L. A. Belady. A study of replacement algorithms for virtual storage computer. *IBM System J.*, 5(2):78–101, July 1996.
- [5] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? High performance garbage collection in Java with MMTk. In *ICSE*, pages 137–146, 2004.
- [6] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. L. Hosking, M. Jump, H. B. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, pages 169–190, 2006.
- [7] 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, and B. Wiedermann. Wake up and smell the coffee: evaluation methodology for the 21st century. *Commun. ACM*, 51(8):83–89, 2008.
- [8] T. Brecht, E. Arjomandi, C. Li, and H. Pham. Controlling garbage collection and heap growth to reduce the execution time of Java applications. *ACM Trans. Program. Lang. Syst.*, 28(5):908–941, 2006.
- [9] Y. Choi and H. Han. Shared heap management for memory-limited Java virtual machines. *ACM Trans. Embed. Comput. Syst.*, 7(2):1–32, 2008.
- [10] E. Cooper, S. Nettles, and I. Subramanian. Improving

the performance of SML garbage collection using application-specific virtual memory management. In *LISP and Functional Programming*, pages 43–52, 1992.

[11] C. Grzegorzczak, S. Soman, C. Krintz, and R. Wolski. Isla Vista heap sizing: using feedback to avoid paging. In *CGO*, pages 325–340, 2007.

[12] M. Hertz, Y. Feng, and E. D. Berger. Garbage collection without paging. In *PLDI*, pages 143–153, 2005.

[13] W. W. Hsu, A. J. Smith, and 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(1):96–143, 2001.

[14] R. Iyer, R. Illikkal, L. Zhao, D. Newell, and J. Moses. Virtual Platform Architectures: a framework for efficient resource metering in datacenter servers. In *SIGMETRICS*, 2009.

[15] JavaSoft. J2SE 1.5.0 documentation: Garbage collector ergonomics.

[16] J.-S. Kim and Y. Hsu. Memory system behavior of Java programs: methodology and analysis. In *SIGMETRICS*, pages 264–274, 2000.

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

[18] D. A. Moon. Garbage collection in a large Lisp system. In *LISP and Functional Programming*, pages 235–246, 1984.

[19] Novell. NetWare 6: Optimizing Garbage Collection.

[20] 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*, pages 20–20, Berkeley, CA, USA, 2001. USENIX Association.

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

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

[23] Y. C. Tay and M. Zou. A page fault equation for modeling the effect of memory size. *Perform. Eval.*, 63(2):99–130, 2006.

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

[25] F. Xian, W. Srisa-an, and H. Jiang. Investigating throughput degradation behavior of Java application servers: a view from inside a virtual machine. In *PPPJ*, pages 40–49, 2006.

[26] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. CRAMM: virtual memory support for garbage-collected applications. In *OSDI*, pages 103–116, 2006.

[27] T. Yang, M. Hertz, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. Automatic heap sizing: taking real memory into account. In *ISMM*, pages 61–72, 2004.

[28] C. Zhang, K. Kelsey, X. Shen, C. Ding, M. Hertz, and

M. Ogihara. Program-level adaptive memory management. In *ISMM*, pages 174–183, 2006.

[29] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In *ASPLOS*, pages 177–188, 2004.

## APPENDIX

### A.1 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 [7], 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 [26], 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 [15]. However, to facilitate tests with various garbage collectors, we switched to the `Jikes` Research Virtual Machine (Version 3.0.1) [1]. The collectors `GenMS`, `MarkSweep` and `SemiSpace` were chosen from its `MMTk` toolkit as representatives of the three major techniques for collecting garbage.

Our first mutator was `ipsixql` from the `DaCapo` benchmark suite [6]. 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) and `xalan` (an XSLT transformer for XML documents) [7].

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 [26])

```

 $k' \leftarrow k$ ; //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

     $x_i \leftarrow \left(\frac{n_i+n_0}{n^*+n_0} - 1 + \frac{n^*+n_0}{n_i+n_0}\right)^{-1}$  for  $i = 1, \dots, k'$ ;
     $\Omega_{k'}^x \leftarrow \{\langle M_i, x_i \rangle \mid i = 1, \dots, k'\}$ ;
    fit  $\Omega_{k'}^x$  with Equation (7);
    record sum of square errors SSE for  $\Omega^n$ ; // (iii) instead of  $\Omega_{k'}^x$ 
    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  $\Omega^n$ ;
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 15: Algorithm for calibrating the parameters through linear regression.**

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.

## A.2 Parameter calibration

For the Page Fault Equation (3), an experiment with a fixed  $H$  yields a data set

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

We use regression to fit the equation to  $\Omega^n$ , 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 (3) has an equivalent linear form

$$M = (M^* - M^o)x + M^o \text{ where } x = \left(\frac{n + n_0}{n^* + n_0} - 1 + \frac{n^* + n_0}{n + n_0}\right)^{-1}. \quad (7)$$

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

(i) Transforming  $\Omega^n$  into a corresponding

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

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  $\Omega^n$  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  $\Omega^x$ , the objective is to get a good fit for  $\Omega^n$ .

These issues are addressed in the calibration algorithm in Figure 15.

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, so on-the-fly calibration is necessary.

The algorithm in Figure 15 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  $P^{\text{miss}}$  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  $P^{\text{miss}}$ , if this is the probability of missing a database buffer.

In our case, measuring  $P^{\text{miss}}$  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 [26, 29]; both require significant changes to the operating system. They also use the Mattson stack [17], which assumes an LRU-like inclusion property that disagrees with the looping behavior in most garbage collectors.

The difficulty in measuring  $P^{\text{miss}}$  with software has led to designs for measurement with hardware [2, 26], but implementing such designs are harder still. However, a major hardware vendor is extending their metering architecture to memory usage [14], so  $P^{\text{miss}}$  measurements with built-in hardware may be possible soon.