A Page Fault Equation for Dynamic Heap Sizing

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.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems—*Modeling techniques*; D.4.2 [**Operating Systems**]: Storage Management—*garbage collection*

General Terms

Performance, Languages

Keywords

garbage collection, heap size, page faults, 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. 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 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 [7, 10].

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 can cause more page faults than mutator execution when the heap extends beyond main memory [20].

Fig. 1 presents measurements from running mutator `pmd` (from the `DaCapo` benchmark suite [5]) 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. 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 = 140MBytes in Fig. 1. If M = 50MBytes, then shrinking the heap to H = 60MBytes would trigger more garbage collection and double the number of page faults. If M = 110MBytes, however, setting H = 60MBytes would reduce the faults to just cold misses, and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
WOSP/SIPEW'10, January 28–30, 2010, San Jose, California, USA.
Copyright 2009 ACM 978-1-60558-563-5/10/01 ...$10.00.

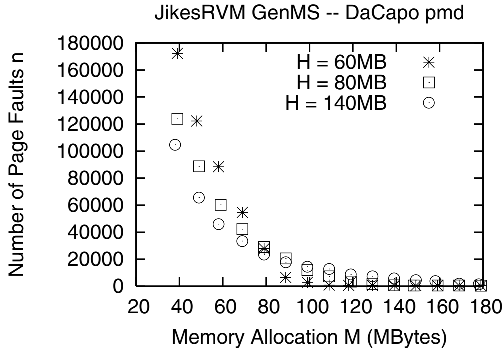


Figure 1: How heap size H and memory allocation M affect the number of page faults n. The garbage collector is `GenMS` and the mutator is `pmd` from the `Dacapo` benchmark suite.

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 [11]). 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 [16].

Our second contribution is the following

Heap Sizing Rule:

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

This rule, illustrated in Fig. 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 [8], tailor the page replacement policy [19], require notification when memory allocation stalls [7], track page references [20], measure heap utilization [1], watch allocation rate [6] or profile the application [21].

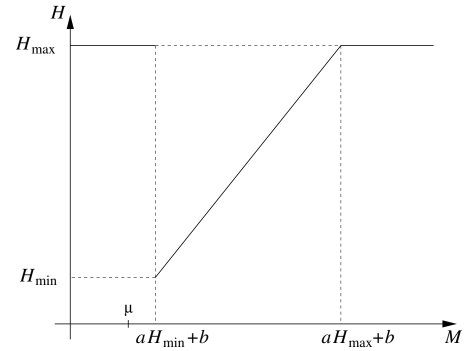


Figure 2: Heap Sizing Rule. (μ is a lower bound for M^* in Eqn. (6); $\mu \approx 80$ in Fig. 1.)

Rule (1) is in closed-form, so there is no need for iterative adjustments [7, 18, 19, 21]. If M changes dynamically, the rule can be used to tune H accordingly, in contrast to static command-line configuration with parameters and thresholds [2, 4, 9, 12].

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 [13].

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.

Section 3 derives the Heap Sizing Rule (1), and presents experiments to show its effectiveness for static M and dynamic M. We conclude with a summary in Section 4.

Due to space constraint, we omit some details (experimental set-up, parameter calibration, etc.) but they can be found in the full paper [15].

2. HEAP-AWARE PAGE FAULT EQUATION

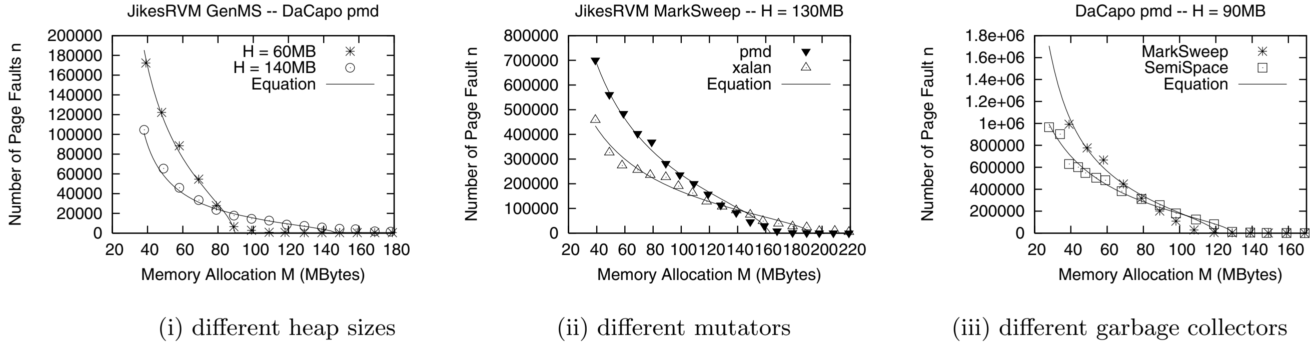
We first recall Tay and Zou's parameterized Page Fault Equation in Section 2.1, and Section 2.2 verifies that it works for garbage-collected workloads. Section 2.3 then derives from it the heap-aware version.

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

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

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



(i) different heap sizes (ii) different mutators (iii) different garbage collectors

Figure 3: The Page Fault Equation can fit data for different heap sizes, mutators and garbage collectors. (i) 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 $n_0 = 12721$ ($R^2 = 0.993$). (ii) 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 $n_0 = 12421$ ($R^2 = 0.997$). (iii) 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 $n_0 = 260659$ ($R^2 = 0.992$).

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 these 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 property 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 [3]. This cannot be; there must be some $M = M^*$ at which n reaches its minimum n^*. Identifying this M^* is critical to our use of the equation for heap sizing.

• The interpretation for M^o varies with the context [16, 17]. 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 [16]. 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 [16]; in this sense, the equation is **universal**.

Garbage-collected applications are particularly challenging because the heap size affects garbage collection frequency, and thus the reference pattern and page fault behavior. This is illustrated in Fig. 1, which shows how heap size affects the number of page faults. Details on the set-up for this and subsequent experiments are given in the full paper [15].

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 Eqn. (2) to fit the result of a change in H, one simply changes the parametric values. Fig. 3(i) illustrates this for the workload of Fig. 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). Details on how we use regression to fit Eqn. (2) to the data are in the full paper [15].

A universal equation should still work if we change the mutator itself. Fig. 3(ii) 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. Fig. 3(iii) illustrates this for `pmd` run with `MarkSweep` and with another garbage collector, `SemiSpace`, using $H = 90$MBytes.

2.3 Top-down refinement

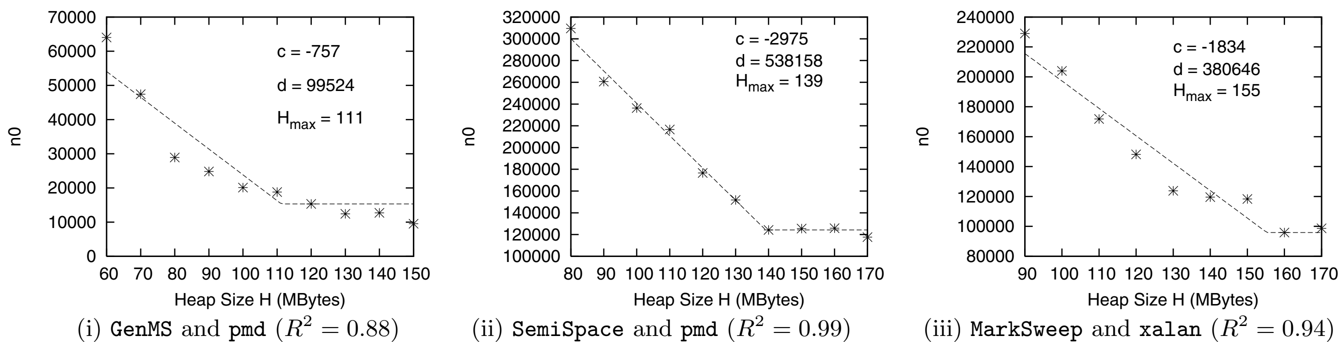
The Page Fault Equation fits the various data sets by changing the numerical values of n^*, M^o, M^* and n_0 when the workload is changed. In the context of heap sizing, how does heap size H affect these parameters?

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. Our experiments show that, for a given workload, M^o varies somewhat randomly when heap size is changed, with no discernible trend. Henceforth, we consider M^o as constant with respect to H.

Garbage collectors like `MarkSweep` access the entire heap when they go about collecting garbage; their memory footprint thus grows with heap size, so we expect M^* to increase with H. Our experiments show that, in fact, M^* varies linearly with H for all our workloads, i.e.

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

Yang et al. defined a metric R that is the minimum real memory required to run an application without substantial paging [20], and found that R is approximately linear in H. Their R is approximately our M^*, so Eqn. (3) agrees with their observation.



(i) GenMS and pmd ($R^2 = 0.88$) (ii) SemiSpace and pmd ($R^2 = 0.99$) (iii) MarkSweep and xalan ($R^2 = 0.94$)

Figure 4: n_0 decreases linearly with H, then flattens out.

As for n_0, Fig. 4 shows that n_0 decreases linearly with H, then flattens out, i.e.

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

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 [14]. There is therefore a bound

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

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

Heap-Aware Page Fault Equation

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

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

$$\text{and } n_0 = \begin{cases} cH + d & \text{for } H_{\min} \leq H < H_{\max} \\ cH_{\max} + d & \text{for } H \geq H_{\max} \end{cases} \quad (6)$$

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.

Besides, $H_{\min}$, the refinement introduces new parameters a, b, c, d and $H_{\max}$; what do they mean? We agree with Yang et al. that the gradient a is a property of the the collection algorithm. As for the intercept b, it is a measure of space overhead that is independent of H.

As H increases, there is less garbage collection and n_0 decreases; in fact, our experiments show that n_0 varies linearly with the number of garbage collection. The gradient c is a measure of the memory taken from the freelist during garbage collection, $H_{\max}$ is the smallest H that suffices to contain all objects created by the workload, and d is implicitly determined by c and the kink in Fig. 4. The full paper [15] describes these interpretations in greater detail; it also identifies a lower bound μ for M^*, indicated in Fig. 2.

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 Fig. 5(i).

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, and Fig. 4 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 Fig. 5(ii).

The two guidelines combine to give the Heap Sizing Rule (1) that is illustrated in Fig. 2

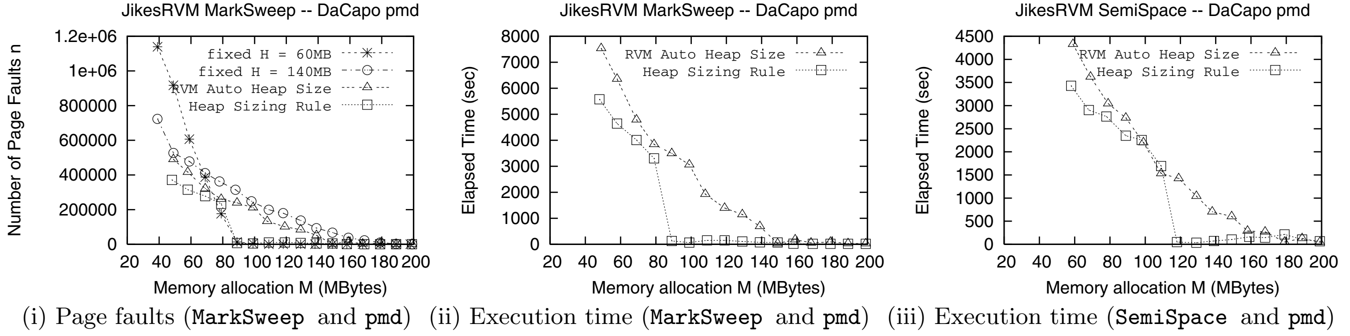
3.2 Experiments with static M

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 [7, 8, 19, 21]. However, we have no access to their implementation, some of which require significant changes to the kernel or mutator.

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.

Fig. 6(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 hence unsurprising 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.



(i) Page faults (`MarkSweep` and `pmd`) (ii) Execution time (`MarkSweep` and `pmd`) (iii) Execution time (`SemiSpace` and `pmd`)

Figure 6: 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 Fig. 2.

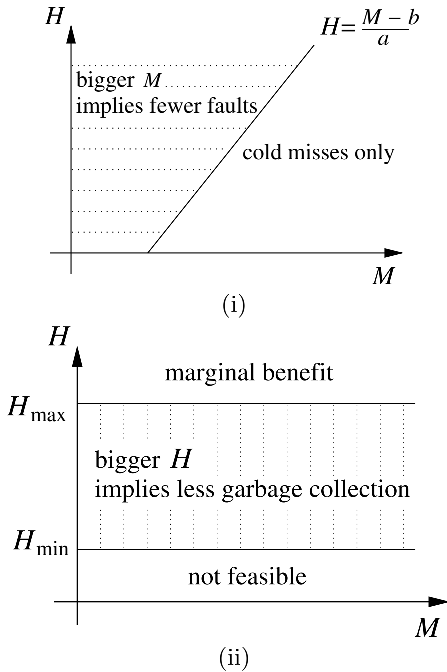


Figure 5: Guideline for heap sizing from (i) the Heap-Aware Page Fault Equation (6) and (ii) Fig. 4.

Notice that, around M = 80MBytes, page faults under the Rule drop sharply to just cold misses. This corresponds to the discontinuity in Fig. 2 at $M = aH_{\min} + b$.

Since disks are much slower than processors, one expects page faults to dominate execution time. Fig. 6(ii) bears this out: the relative performance in execution time between the two policies is similar to that in Fig. 6(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. Fig. 6(iii) shows similar results for `pmd` run with `SemiSpace`.

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 executes a loop that repeatedly locks 30MBytes (in 10MByte increments) and unlocks them. To prolong the execution time, we run the mutator 5 times in succession.

Fig. 7 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 1 shows that page faults under the Rule is an order of magnitude less than those under `JikesRVM`'s automatic sizing. The gap for execution time is similar. These indicate the Rule's effectiveness for dynamic heap sizing.

4. 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 (Fig. 2) for how H should vary with M. It aims to first minimize page faults (Fig. 5(i)), then garbage collection (Fig. 5(ii)), 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 (Fig. 6) and dynamic M (Table 1). 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 (Fig. 6(i) and Fig. 6(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 (Fig. 3), thus demonstrating its universality. Its parameters have interpretations that relate to the garbage collection algorithm and the mutators' memory requirements (Section 2.3).

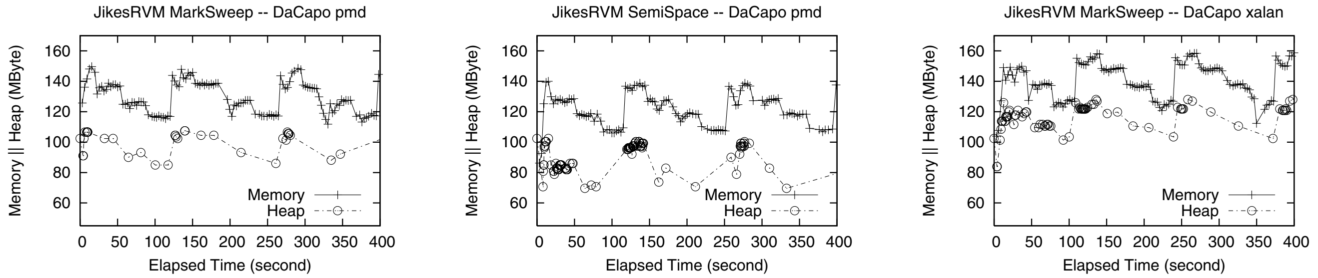


Figure 7: How the Heap Sizing Rule adjusts H at each garbage collection when M varies dynamically. (To plot the points for M, we run a background process that measures M every 3 seconds.)

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

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 [17]. In future work, we plan to similarly apply the Equation to dynamic heap sharing.

5. ACKNOWLEDGMENTS

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

6. REFERENCES

[1] B. Alpern et al. The Jikes Research Virtual Machine project: Building an open-source research community. *IBM Systems Journal*, 44(2):399–418, 2005.

[2] BEA WebLogic. JRockit: Java for the enterprise. *White Paper (online)*.

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

[4] 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.

[5] S. M. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, pages 169–190, 2006.

[6] 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.

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

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

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

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

[11] F. Mao, E. Z. Zhang, and X. Shen. Influence of program inputs on the selection of garbage collectors. In *VEE*, pages 91–100, 2009.

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

[13] T. Printezis. Hot-swapping between a mark&sweep and a mark&compact garbage collector in a generational environment. In *JVM Research and Technology Symp.*, pages 20–20, 2001.

[14] S. Soman, C. Krintz, and D. F. Bacon. Dynamic selection of application-specific garbage collectors. In *ISMM*, pages 49–60, 2004.

[15] Y. C. Tay and X. R. Zong. A page fault equation for dynamic heap sizing. *http://www.math.nus.edu.sg/∼mattyc/HeapSizing.pdf*.

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

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

[18] 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.

[19] 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.

[20] 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.

[21] C. Zhang et al. Program-level adaptive memory management. In *ISMM*, pages 174–183, 2006.