Assessing the Limits of Program-Specific Garbage Collection Performance

Nicholas Jacek   Meng-Chieh Chiu   Benjamin Marlin   Eliot Moss

College of Information and Computer Sciences
University of Massachusetts Amherst
Amherst, MA 01003 USA
{njacek,joechiu,marlin,moss}@cs.umass.edu

Abstract

We consider the ultimate limits of program-specific garbage collector performance for real programs. We first characterize the GC schedule optimization problem using Markov Decision Processes (MDPs). Based on this characterization, we develop a method of determining, for a given program run and heap size, an *optimal* schedule of collections for a non-generational collector. We further explore the limits of performance of a *generational* collector, where it is not feasible to search the space of schedules to prove optimality. Still, we show significant improvements with Least Squares Policy Iteration, a reinforcement learning technique for solving MDPs. We demonstrate that there is considerable promise to reduce garbage collection costs by developing program-specific collection policies.

Categories and Subject Descriptors D.3.3 Language Constructs and Features: Dynamic storage management

Keywords Optimal garbage collection, Markov decision processes, least squares policy iteration

1. Introduction

Garbage collection (GC), that is, automatic storage reclamation, has been a feature of some programming languages from the early days of high-level languages, and now it is *expected* for many major languages (and intentionally avoided by others). While GC technology has advanced greatly, reaching a stage of maturity where its overheads are acceptable for casual, and some heavy production use, and even in real-time systems, we know little about the limits of its possible performance. There are a few worst-case bounds on explicit allocation and freeing schemes [Robson, 1971, 1974, 1980] for compacting GC [Bendersky and Petrank, 2012], and for reference counting GC [Boehm, 2004]. These bounds assume that one first fixes a storage management algorithm, and then presents it with a program whose allocation behavior the algorithm must carry out. Such bounds are proved by considering an adversarial program.

In practice, however, we are generally concerned with how a storage management algorithm performs when presented with a particular program: the program we wish to run now. We can measure the performance of various GC algorithms on this program, but we have no idea how good they are in an absolute sense. Put another way, to date there are no published results on the lowest possible GC cost achievable for particular programs under particular circumstances.[1] We seek here to fill this gap by developing and evaluating methodology for assessing the limits of program-specific garbage collection performance.

What we present here is a *limit study*. Our goal is to determine the best possible GC performance (here measured in terms of bytes traced or copied). To accomplish this goal, we present methods for optimizing over the space of all possible sequences of decisions of when to collect that are consistent with not overflowing a given heap size. We refer to an optimal sequence of collection points as an optimal GC *schedule*.

When considering optimality of GC, we find it helpful to distinguish between an optimal *schedule* and an optimal *policy*. An optimal GC schedule is a GC schedule of lowest cost in a specific situation, e.g., for a specific heap size. Note that optimality of a GC schedule is defined with respect to a specific execution, i.e., a trace of a particular program executing in a particular situation. For a deterministic program and language implementation, an execution may be defined

PLDI'16, June 13–17, 2016, Santa Barbara, CA, USA
ACM. 978-1-4503-4261-2/16/06...$15.00
http://dx.doi.org/10.1145/2908080.2908120

[1] Hence we provide no "related work" section since we have not found any comparable work.

by the combination of a program and its input / environment. A schedule says nothing about how that schedule is obtained.

A GC *policy*, in contrast to a schedule, uses some information about the state of execution of the program, often some current and recent allocation and collection statistics, to determine when to collect. An *optimal* GC policy (for a particular execution) is one that leads to an optimal schedule. Note that a policy, as used here, is situation-specific. It need not perform well or even be applicable in other situations. In our larger research plan, we are interested in program-specific, but situation-general, policies, but here the focus is on the ultimate limits of performance, which can be characterized in terms of minimum cost schedules. These limits give a yardstick for measuring how well more general policies perform.

To develop optimal GC schedules, we work from traces of allocation and heap behavior acquired using the Elephant Tracks (ET) tool [Ricci et al., 2011, 2013]. We consider first the simplest case: non-generational (NG) GC with a fixed heap size. Since a given schedule is optimal only for a specific trace and heap size, we compute optimal schedules for a variety of heap sizes. In principle we could explore schedules that allow collection at *any* point in the trace, i.e., just before any allocation. For practicality, and to mirror more typical storage management behavior, we limit the points we consider to ones spaced about 256KB[2] apart, i.e., we assume a block-oriented scheme with a block size of 256KB.[3] We show that NG GC can be modeled as a first-order Markov Decision Process (MDP), and then show that given the information available from an ET trace after some simple post processing, we can exactly solve the MDP using dynamic programming, yielding the absolute minimum cost schedule.

The baseline against which we compare is simple: the schedule produced by collecting only when the heap is full, i.e., when the next block of allocation will not fit. We call that the *default* policy. In all cases we use as our measure of cost the number of bytes remaining after collection, with one caveat: large objects that contain only bytes (no pointer fields/elements) incur no cost to retain. (This is typical of modern systems, which segregate large objects and avoid any significant work on them if they contain no pointers.) Here "large" means consuming at least one 256KB block.

At this point we should mention that, because we use ET, which offers fine-grained analysis of when objects "die" (become unreachable), and ET is Java-specific, our results pertain to Java programs. Specifically, we use traces from programs in the DaCapo benchmark suite [Blackburn et al., 2006, 2008], a suite designed to reveal various interesting storage use and GC behaviors.

After presenting results on NG GC, we turn to generational (Gen) GC. We use the same traces from Java programs. However, we use some novel trace post-processing techniques that allow us to group objects into what we call *cohorts*. All objects in a given cohort have the same behavior with respect to Gen GC, which allows us to simulate a schedule without having to simulate the behavior of individual objects in the heap. As with NG GC, we used a size for old space, and to that we add a fixed young space size. Even with these constraints, the space of possible schedules is extremely large. We characterize Gen GC as an MDP and show that it in fact has variable Markov order, making the resulting MDP extremely difficult to solve exactly. We introduce an approximate solution approach based on Least Squares Policy Iteration (LSPI), a reinforcement learning technique for solving MDPs. The LSPI results for Gen GC are not provably optimal due to the approximation we introduce, so we refer to these resulting schedules as *optimized*. We still expect the cost of these optimized schedules to serve as a useful benchmark for methods that learn general policies in the generational setting.

In summary, this limit study provides insights about several facets of GC design. First, it can tell us whether GC is essentially a solved problem in practice or whether further performance gains may be possible. Second, if we have a mechanism that gives some improvement, it can tell us how much of the potential improvement the mechanism is achieving. Third, it may be that improvement is possible for some programs and not others, so a limit study of a particular program can guide whether developing a program-specific GC mechanism might be worthwhile. The techniques we propose can guide future effort to decrease GC cost, and may offer deeper insight into the nature and difficulty of the GC problem.

2. MDP Models of Garbage Collection

We now present Markov Decision Process (MDP) [Bellman, 1957; Puterman, 2014] formulations of both NG and Gen GC. We consider models with three primary components: a program P, a heap H, and a controller C. We assume that the model evolves in discrete time where the state variable t represents the current time step. On each time step, the program P does one of three things: (1) it allocates an object; (2) it causes an object to die; (3) it terminates. If P causes an object to die, it notifies H, and H adjusts its state (not visible to C). If P terminates, the MDP stops. If P allocates an object, the allocation request goes to C. C optionally performs a garbage collection, notifying H to update its state, and then forwards the allocation request to H. C may maintain state and some parts of H's state may also be visible to C.

In this section, we begin by describing the the mathematical structure of an MDP. We then describe the state variables and dynamics of NG and Gen collection in terms of P, C, and H. Finally, we provide an MDP formulation of NG and Gen collection.

[2] We use K and M for powers of 1024; B for bytes.

[3] We say "about 256KB" because of the need to align on object boundaries, giving the specifics later.

2.1 Markov Decision Processes

A first-order deterministic, non-discounted Markov Decision Process (MDP) is defined by a tuple: $(\mathscr{S}, \mathscr{A}, \mathscr{C}, \mathscr{T})$ [Bellman, 1957; Puterman, 2014]. Here, $\mathscr{S}$ is a finite set of states, $\mathscr{A}$ is a finite set of actions, $C : \mathscr{S} \times \mathscr{A} \mapsto \mathbb{R}$ is a cost function that maps state-action pairs to real-valued costs, and $\mathscr{T} : \mathscr{S} \times \mathscr{A} \mapsto \mathscr{S}$ is a transition operator that maps state-action pairs into new states. A higher-order MDP allows the cost function and the transition operator to depend on not only the current state and action but also a history consisting of one or more past states and actions.

2.2 Modeling Non-generational Collection

From the point of view of the GC system, a running program can be described by a sequence of allocation requests from the Program P for objects, one per time point t.[4] An object o_t is defined as a tuple (b, d, s) where b represents the time the object is born, d represents the time the object dies, and s represents the size of the object in bytes. We let Ω be the set of objects that are allocated during a program run. If we number the objects in the order in which they are allocated, $o_i.b = i$.

In the NG case, the heap itself can be described by a tuple of three state variables (L_t, D_t, F_t) at each time step t, as well as a static state variable S. L_t represents the total size of live objects in the heap, D_t represents the total size of dead objects in the heap, F_t represents the total amount of free space in the heap, and S represents the size of the heap. The heap maintains the invariant that $S = L_t + D_t + F_t$ at all times t. (We consider only fixed size heaps in this study; varying-size heaps will almost certain lead to higher order MDPs.)

The controller implements a GC policy π. This policy is a function that maps the visible state of the rest of the system into the choice of an action a. The set of possible actions $\mathscr{A}$ is a property of the controller, C. In NG collection, the possible actions are `collect` and `no-collect`. However, some actions may not be available at a given time step t. The set of available actions is denoted by $\mathscr{A}_t \subseteq \mathscr{A}$. For example, the choice not to collect on a given time step t is not valid if $o_t.s$ (the size of the object to be allocated at time step t) is larger than the remaining free heap space F_t. The visible state of the system at time t is denoted V_t. The policy π has access only to V_t, not the entire state. The contents of V_t depend on the specific controller, but may contain the current time step t, the current amount of free space in the heap F_t, the allocated space A_t (defined as $L_t + D_t$), and a feature vector X_t containing additional information about the state of the system obtained from suitably instrumenting the running program. (Not both of A_t and F_t are strictly necessary, but we find having both to be convenient.) In this study we take X_t as being empty. Thus, π maps visible state to legal actions: $\pi(V_t) \rightarrow \mathscr{A}_t$. We observe that since we include t in V_t and π is defined in the context of a single program execution, π can be structured as a lookup table base solely on t. We emphasize that this is *not* what a systems builder would call a policy! It is an *oracle*, *not* a mechanism. Thus π chooses a sequence of actions a_t, one for each time step t. We call that set of actions a (GC) *schedule*.

The evolution of the state variables that define the system is driven by the sequence of allocation and death requests issued by the program P. The initial state of the heap prior to any allocations is $(0, 0, S)$, indicating that all space is free. At each time step t, the program requests the allocation of $o_t.s$ bytes of storage for the object o_t: `alloc`$(o_t.s)$. As part of the same time step, it announces the death of all objects that died on time step t (i.e., since the previous allocation) by issuing the set of notifications $\{$`death`$(o_{t'}) | o_{t'} \in \Omega$ and $o_{t'}.d = t\}$. The controller C receives the `alloc`$(o_t.s)$ request and must fulfill the allocation by issuing an `halloc`$(o_t.s)$ request to the heap H. However, before forwarding the request to the heap, the controller selects an action $a_t \in \mathscr{A}_t$ by applying the policy function $\pi()$ to the visible state vector V_t.

To update the state of the heap, we first account for the objects announced as dead on time step t. We let $\delta_t = \{t' | o_{t'} \in \Omega$ and $o_{t'}.d = t\}$ be the set of time points whose objects were announced as dead. We then let $\sigma_t = \sum_{t' \in \delta_t} o_{t'}.s$ be the total number of bytes of all objects that were reported dead on time step t. If the state of the heap on time step t is (L_t, D_t, F_t), then after accounting for objects that were announced as dead, we have that the state of the heap is $(L_t - \sigma_t, D_t + \sigma_t, F_t)$.

Now we take into account the action selected by the controller. We first note that the set of legal actions $\mathscr{A}_t$ is defined to be $\{$`collect`, `no-collect`$\}$ if $F_t \geq o_t.s$ and $\{$`collect`$\}$ otherwise. This forces a collection at time step t if there is insufficient space to allocate the object o_t. Once object o_t has been allocated, the time step is incremented. If the action a_t selected by the controller is `collect`, the state of the heap is further updated to reclaim the space allocated to all dead objects. The updated state of the heap becomes $(L_t - \sigma_t, 0, F_t + D_t + \sigma_t)$. Following the allocation of object o_t, the state of the heap is $(L_t - \sigma_t + s_t, 0, F_t + D_t + \sigma_t - s_t)$. If the action a_t selected by the controller is `no-collect`, the updated state of the heap following the allocation of object o_t becomes $(L_t - \sigma_t + s_t, D_t + \sigma_t, F_t - s_t)$.

The visible state consists of: t, $A_t = L_t + D_t$, and F_t.

In summary, we thus have:

$$L_{t+1} \leftarrow \begin{cases} L_t - \sigma_t + o_t.s & \text{collect} \\ L_t - \sigma_t + o_t.s & \text{no-collect} \end{cases} \tag{1}$$

$$D_{t+1} \leftarrow \begin{cases} 0 & \text{collect} \\ D_t + \sigma_t & \text{no-collect} \end{cases} \tag{2}$$

$$F_{t+1} \leftarrow \begin{cases} F_t + D_t + \sigma_t - o_t.s & \text{collect} \\ F_t - o_t.s & \text{no-collect} \end{cases} \tag{3}$$

[4] When describing MDPs we define time in terms of steps of the state machine, i.e., 0, 1, etc. In other words, we count time in terms of objects allocated, as opposed to bytes allocated (typical in GC research) or some other time unit.

It is now plain that the evolution of the state of the heap depends only on the three sequences $o_t.s$, σ_t, and a_t. Furthermore, we can see that the sequence of live size values L_t is actually independent of a_t. As a corollary of the heap size invariant, we obtain the result that the free space F_{t+1} obtained following a collection at time t is always equal to $S - L_{t+1}$ since $D_{t+1} = 0$. Since the sequence of values L_t is independent of a_t, this means that the free space F_{t+1} following a collection at time t is independent of the sequence of prior actions. Finally, since $D_{t+1} = 0$ following a collection, we obtain the result that the full state of the heap $(L_{t+1}, 0, S - L_{t+1})$ following a collection is independent of the sequence of actions before time t. This fact will play an important role in designing algorithms for determining optimal GC schedules for the NG case.

We can now define the Markov decision process for NG collection. Recall that an MDP is a tuple $(\mathscr{S}, \mathscr{A}, \mathscr{C}, \mathscr{T})$. In the NG case, the set of states $\mathscr{S}$ consists of all tuples (t, L_t, D_t, F_t) that are accessible via valid sequences of actions. The actions are `collect` and `no-collect`. The transition operator is deterministic and is given by Equations 1-3. The cost of taking the action `collect` in the state (t, L_t, D_t, F_t) is simply L_t since only live objects must be copied/processed during collection. On the other hand, the cost associated with not collecting is 0. Formally, we have:

$$\mathscr{C}((t, L_t, D_t, F_t), a_t) = \begin{cases} L_t & \text{collect} \\ 0 & \text{no-collect} \end{cases} \tag{4}$$

We note that we assume a *tracing* collector in determining the cost of GC to be L_t. A copying collector certainly incurs cost determined strongly by L_t, and the same can be said about the mark phase of a mark-sweep collector. The cost of sweeping also conforms somewhat to this cost model: the cost of sweeping the live objects is determined by L_t, and the cost of sweeping dead objects can be viewed at part of the cost related to allocating them in the first place. However, objects already on a free list will be swept repeatedly, making this cost model more approximate.

In practice we refine the model just presented to account for large objects that contain no pointers (loosely, large byte arrays; the "class" pointer in object-oriented languages here does not count as a pointer since most systems arrange for classes not to be moved or reclaimed). Collectors usually handle large byte arrays specially, allocating them with a block-based allocator and not moving or copying them. This avoids any significant collector work to process these objects. We can model this by splitting the size request $o_t.s$ into two portions, $o_t.s$ and $o_t.s'$, where $o_t.s$ is handled as before and $o_t.s'$ is the no-collection-cost amount. (For a single allocation, one of these will be 0.) We likewise split L_t, D_t, and A_t, tracking the two kinds of objects separately. The cost values are as in Equation 4, notably excluding L'_t. Since the extension is straightforward but notationally tedious, we omit the full model here.

Another refinement captures the fact that large objects that *do* contain pointers may be allocated with the block-base allocator and thus consume total space that is rounded up to an integer number of blocks, while the collector processing costs remain proportional to the inherent size of the object. This we handle by allowing both $o_t.s$ and $o_t.s'$ to be non-zero with $o_t.s'$ representing the padding to the end of a block.

2.3 Modeling Generational Collection

The Gen case requires several modifications to the previous model. First, the model for the heap is extended to include separate young and old sections with sizes S^Y and S^O. We require $S^O \geq S^Y$ so that the young space can be fully used. The set of actions that the controller can perform is augmented to allow collection of only the young section of the heap, or collection of the full heap. The action set is thus $\mathscr{A} = \{$`young-collect, full-collect, no-collect`$\}$.

Note that allocations, except for very large objects, are into young space, with collections *promoting* certain young space objects to old space. An object larger than young space will be allocated directly to old space. We force objects to be considered for promotion in age order (oldest first). Therefore allocating a very large object requires young space to be empty, and thus may force a collection first. Also, young space is managed in terms of *blocks*—a group always consumes an integer number of blocks in young space. However, when (non-large) objects are promoted to old space, they are packed together. Our rationale is that this keeps GC decisions on block boundaries as considered from the point of view of the entire trace. The management of old space assumes it is configured as a contiguous sequence of blocks, while that need not be the case for the nursery. If we disallowed small objects from crossing block boundaries in *old* space, then we would have to specify the order in which objects are promoted, which over-specifies the problem. While obviously any detail can affect the exact numeric result in a particular case, we feel these choices are reasonable in order to obtain estimates of optimal GC cost in a practical amount of time.

The heap is modeled by the tuple $(L_t^Y, L_t^O, D_t^Y, D_t^O, F_t^Y, F_t^O, B_t, T_t^Y, T_t^O)$. The superscript Y indicates state variables related to the young section of the heap and O those related to the old section. The L variables represent the live size of a section, the D variables represent the dead size, and the F variables the free space. B_t is the total amount of "baggage" in the young section of the heap. The baggage is the total number of bytes of dead objects that will be promoted to the old section of the heap if only the young section of the heap is collected. The variables T_t^Y and T_t^O indicate the time step on which the young space and the old space were last collected (note that a young collection updates T_t^Y only, while a full collection updates both T_t^Y and T_t^O). This expanded model includes two heap invariants, one for each section of the heap: $L_t^O + D_t^O + F_t^O = S^O$ and $L_t^Y + D_t^Y + F_t^Y + B_t = S^Y$.

The dynamics of full collections work identically to the single-heap case. However, the dynamics of a young collection are more nuanced. When a young collection is performed, all live objects in the young section of the heap are copied to the old section of the heap. However, dead objects in the young generation that are pointed to by objects in the old generation appear still to be live during a young collection and are *also* copied to the old section of the heap. To support this reasoning accurately, the model of an object must be expanded to include a pre-birth time[5] $o_t.p$: $o_t = (p, b, d, s)$ where p is the pre-birth time, b is the birth time (t when time is measured in MDP steps), d is the death time, and s is the size of the object.

The initial conditions for the heap are $L_t^Y = 0$, $L_t^O = 0$, $D_t^Y = 0, D_t^O = 0, F_t^Y = S^Y, F_t^O = S^O, B_t = 0$. We need several auxiliary sequences to keep track of the objects that die at each time step. First we let $\delta_t^O = \{t' | o_{t'} \in \Omega, o_{t'}.d = t, o_{t'}.b < T_t^Y\}$. This set contains the indices of all objects that die at time t, and were born before the last young collection. This means these objects were promoted to the old space. Their total size is given by $\sigma_t^O = \sum_{t' \in \delta_t^O} o_{t'}.s$. Next, we let $\delta_t^B = \{t' | o_{t'} \in \Omega, o_{t'}.s = t, o_{t'}.b \geq T_t^Y, o_{t'}.p < T_t^Y\}$. This set contains the indices of all objects that die at time t, were born after the last young collection, but have a pre-birth time before the last young collection. These objects are pointed to by objects in the old space at the time they die, and thus contribute to the baggage in the young generation. Their total size is $\sigma_t^B = \sum_{t' \in \delta_t^B} o_{t'}.s$. Finally, let $\delta_t^Y = \{t' | o_{t'} \in \Omega, o_{t'}.d = t, o_{t'}.b \geq T_t^Y, o_{t'}.p \geq T_t^Y\}$. This set contains the indices of all objects that die at time t, were born after the last young collection, and have a pre-birth time after the last young collection. These objects are correctly identified as being dead within the young space. Their total size is $\sigma_t^Y = \sum_{t' \in \delta_t^Y} s_{t'}^o.s$. We further define $\sigma_t = \sigma_t^O + \sigma_t^B + \sigma_t^Y$.

The visible state consists of: $t, A_t^Y = L_t^Y + D_t^Y + B_t^Y, A_t^O = L_t^O + D_t^O, F_t^Y, F_t^O, T_t^Y$, and T_t^O.

We can now specify the full dynamics of generational collection:

$$L_{t+1}^O \leftarrow \begin{cases} L_t^O + L_t^Y - \sigma_t^O & \text{full/young-collect} \\ L_t^O - \sigma_t^O & \text{no-collect} \end{cases} \tag{5}$$

$$D_{t+1}^O \leftarrow \begin{cases} 0 & \text{full-collect} \\ D_t^O + B_t + \sigma_t^O & \text{young-collect} \\ D_t^O + \sigma_t^O & \text{no-collect} \end{cases} \tag{6}$$

$$F_{t+1}^O \leftarrow \begin{cases} F_t^O - L_t^Y + D_t^O + \sigma_t^O & \text{full-collect} \\ F_t^O - L_t^Y - B_t & \text{young-collect} \\ F_t^O & \text{no-collect} \end{cases} \tag{7}$$

$$L_{t+1}^Y \leftarrow \begin{cases} o_t.s & \text{full/young-collect} \\ L_t^Y - \sigma_t^Y - \sigma_t^B + o_t.s & \text{no-collect} \end{cases} \tag{8}$$

$$D_{t+1}^Y \leftarrow \begin{cases} 0 & \text{full/young-collect} \\ D_t + \sigma_t^Y & \text{no-collect} \end{cases} \tag{9}$$

$$F_{t+1}^Y \leftarrow \begin{cases} S^Y - o_t.s & \text{full/young-collect} \\ F_t^Y - o_t.s & \text{no-collect} \end{cases} \tag{10}$$

$$B_{t+1} \leftarrow \begin{cases} 0 & \text{full/young-collect} \\ B_t + \sigma_t^B & \text{no-collect} \end{cases} \tag{11}$$

$$T_{t+1}^Y \leftarrow \begin{cases} t & \text{full/young-collect} \\ T_t^Y & \text{no-collect} \end{cases} \tag{12}$$

$$T_{t+1}^O \leftarrow \begin{cases} t & \text{full-collect} \\ T_t^O & \text{young/no-collect} \end{cases} \tag{13}$$

We can now define the Markov decision process for the Gen case. Recall that an MDP is a tuple $(\mathscr{S}, \mathscr{A}, \mathscr{C}, \mathscr{T})$. In the Gen case, the set of states $\mathscr{S}$ consists of all tuples $(L_t^Y, L_t^O, D_t^Y, D_t^O, F_t^Y, F_t^O, B_t, T_t^Y, T_t^O)$ that are accessible via valid sequences of actions. The actions are `young-collect`, `old-collect`, and `no-collect`. The transition operator is deterministic and is given by Equations 5-13. The cost of taking the action `old-collect` is simply $L_t^Y + L_t^O$. The cost of taking the action `young-collect` is $L_t^Y + B_t$. The cost of performing no collection is 0.

There are two refinements we make to this model, but whose details we omit here. First is the handling of large objects, both ones containing no pointers and ones with pointers, as previously described. The no-cost objects/space must be tracked as part of L^Y, D^Y, or B in the young space, but add nothing to the cost of collection, and must be tracked in L^O and D^O in old space. The other refinement is needed to handle objects larger than S^Y. In that case we require that the young space be empty *and* there be adequate free space in the old space. This mandates a full collection if $S^Y - F^Y > F^O - s_t$ or a young collection if $F^Y \neq S^Y$. The large object is then allocated directly into old space.

What makes Gen collection more complex than NG is the fact that the cost function is only partially first-order Markov. As in the NG case, the cost of a full collection is simply the total number of live bytes in the young and old spaces, and this number is independent of the past sequence of actions. The cost of no collection is 0, which is clearly also independent of the sequence of actions. However, the cost of a young collection is $L_t^Y + B_t$, and the value of B_t depends on the sequence of young collections that occurs from the last full collection up to time t, and the specifics of these collections can vary. The cost function as it relates to young collections thus has variable Markov order, which is a significantly more complex structure than the NG case.

3. Optimizing GC Schedules

In the previous section we provided detailed characterizations of NG and Gen collection as deterministic non-discounted Markov Decision Processes. The one element of these descriptions that was left unspecified is the policy function $\pi()$ that maps states to actions. We begin this section by describing standard default policies for the NG and Gen cases. We

[5] Please pardon the forward reference; this concept is defined carefully in Section 4.1.

then describe methods for optimizing GC schedules based on dynamic programming and reinforcement learning.

3.1 Default Policies

In the NG case, a simple default policy is obtained by allocating objects until the heap fills, and then collecting. Mathematically, this policy can be described as shown below:

$$a_t \leftarrow \begin{cases} \text{no-collect} & F_t \geq o_t.s \\ \text{collect} & \text{otherwise} \end{cases} \tag{14}$$

In the generational case, a similar default policy can be constructed. This policy allocates objects to the young space until the young space fills, at which point only the young space is collected, if possible. If there is insufficient heap left in the old space to accommodate the possible promoted objects from the young space, then a full collection is performed instead. This policy is shown below:

$$a_t \leftarrow \begin{cases} \text{no-collect} & F_t^Y \geq o_t.s \\ \text{young-collect} & F_t^Y < o_t.s \text{ and } F_t^O \geq L_t^Y + D_t^Y + B_t \\ \text{full-collect} & \text{otherwise} \end{cases} \tag{15}$$

3.2 Dynamic Programming

Given an MDP that describes NG collection (this includes a heap size), and a program trace, it is possible to find an exact minimum cost collection schedule for the program trace using dynamic programming [Bertsekas, 1995]. This is possible because the MDP for single heap collection is deterministic and first-order Markov. We give the dynamic program for computing the cost of the optimal schedule below. Recall that the cost of collecting at time point t is equal to L_t, and there is no cost associated with not collecting. To facilitate the recursive description of the algorithm, we define the cost of collecting after the program ends to be 0.

$$\kappa_t \leftarrow \begin{cases} \min\limits_{t' \in f(t)} L_{t'} + \kappa_{t'+1} & t \leq N \\ 0 & t = N+1 \end{cases} \tag{16}$$

Here, N is the total number of time steps. The terms κ_t give the minimum total collection cost between time t and the end of the program, assuming that a collection occurred (or the program started) at time $t-1$. To compute this cost, the dynamic program considers all collection points in the set $f(t)$ (f for "feasible"). The function $f(t)$ includes all time points between t and the next time point at which a collection is forced. In terms of the MDP description of the process, $f(t)$ is given by:

$$f(t) = \{t' | t' \geq t,\ F_{t-1} \geq \sum_{i=t}^{t'-1} o_i.s\} \tag{17}$$

The total cost of the optimal schedule is obtained from the dynamic programming variables as κ_1. The actual optimal sequence of actions can be recovered by determining the sequence of split points t', which correspond to the time points of the collections. This requires one backward pass through the dynamic programming table to compute.

3.3 Reinforcement Learning

The previous dynamic programming algorithm is possible because the MDP for the NG case is deterministic and first-order Markov. As mentioned in the previous section, the MDP for the Gen case is significantly more complex due to the fact that the cost function is first-order Markov as it relates to full collections and no collections, but variable-order as it relates to young collections.

While the process is certainly still deterministic, the presence of variable-order Markov structure requires the use of approximations when solving the MDP. In this work, we elect to approximate the variable-order cost function with a partially stochastic second-order cost function. Essentially, this approximation views the cost of performing a young collection at time t as a random variable that conditions on the current state and the state at the last collection. The variation due to different sequences of young collections is viewed as random variation in the value of the cost of a young collection at time t.

In terms of deriving an optimal schedule for a given program run, it suffices to define the state space of the MDP to be $[1,N] \times [1,N]$ where the state $s = (t,t')$ indicates that the last collection occurred at time t' and the current time step is t. This effectively converts the second order MDP back to a first-order MDP with augmented state so that standard reinforcement learning (RL) [Sutton and Barto, 1998] techniques for first-order MDPs can be used to learn a policy.

With this approximation in hand, we can turn to the use of RL techniques to solve the MDP. In reinforcement learning, an agent in a given state $s \in \mathscr{S}$ selects an action $a \in \mathscr{A}_s$ according to a policy $\pi : \mathscr{S} \mapsto \mathscr{A}_s$, where $\mathscr{A}_s \subseteq \mathscr{A}$ denotes the actions that are available in state s. The agent chooses a new action according to its policy, and the process repeats until the system reaches a terminal state. The goal is to find an optimal policy π^* that minimizes the total sum of costs that the agent incurs: $c_0 + c_1 + \cdots + c_N$.[6]

An important concept in reinforcement learning is the state-action value function for a policy, which can be written in recursive form as shown below [Sutton and Barto, 1998]. This form of the state-action value function is known as the Bellman Equation. Recalling that $\mathscr{T}$ is the MDP's state transition function, we simplify notation by letting $s' = \mathscr{T}(s,a)$.

$$Q^\pi(s,a) = c_0 + Q^\pi(s', \pi(s')). \tag{18}$$

Using matrix notation, the Bellman Equation becomes:

$$Q^\pi = C + \Pi^\pi Q^\pi \tag{19}$$

[6] MDPs are usually given in the equivalent form where we wish to maximize the rewards that the agent receives. We present the cost minimization form in order to simplify the connection between MDPs and GC cost minimization.

where Q^π and C are vectors of size $|\mathscr{S}||\mathscr{A}|$. Π^π is a matrix of size $(|\mathscr{S}||\mathscr{A}| \times |\mathscr{S}||\mathscr{A}|)$ where $\Pi^\pi((s',a'),(s,a))$ is 1 if $s' = \mathscr{T}(s,a)$ and $a' = \pi(s')$ and is 0 otherwise.

Policy Iteration (PI) [Howard, 1960] is an algorithm that generates a series of monotonically improving policies $\pi_0, \pi_1, \cdots$. Once $\pi_m = \pi_{m+1}$, the algorithm has converged to an optimal policy, so $\pi_m = \pi^*$. PI uses two alternating steps: In the policy evaluation step, the state-action value function Q^{π_m} is calculated. Then, in the policy improvement step, the next policy is defined greedily in terms of the previous state-action value function:

$$\pi_{m+1}(s) = \arg\min_{a \in \mathscr{A}_s} Q^{\pi_m}(s,a). \qquad (20)$$

In practice, for large factored state spaces, PI breaks down and it is preferable to use some form of function approximation [Sutton et al., 1999]. Least-Squares Policy Iteration (LSPI) modifies the PI procedure to use linear function approximation [Lagoudakis and Parr, 2003]. In this case, we are given a set of fixed basis functions $\phi_1(s,a)$, $\phi_2(s,a)$, $\cdots, \phi_N(s,a)$ and approximate Q^π as a linear combination of these functions:

$$Q^\pi(s,a) = \sum_{i=1}^{N} \phi_i(s,a)\theta_i \quad \text{or equivalently} \quad Q^\pi = \Phi\theta^\pi \quad (21)$$

Here, θ is a vector of size N known as the weight vector. When we use this approximation, the goal of the policy evaluation step becomes calculating θ^π.

To achieve this, LSPI uses a modified form of the Bellman Equation that includes an orthogonal projection onto the space spanned by the basis functions. The policy update is obtained as follows:

$$A = \Phi^T(\Phi - \Pi^\pi\Phi) \qquad B = \Phi^T C \qquad \theta^\pi = A^{-1}B \quad (22)$$

A and B can be calculated from a set of samples of states, actions, and resulting costs and next states from the MDP. Given one such set, in no particular order, $((s_1,a_1,c_1,s'_1)$, $(s_2,a_2,c_2,s'_2), \cdots, (s_L,a_L,c_L,s'_L))$, we can calculate them as

$$A = \frac{1}{L}\sum_{i=1}^{L} \phi(s_i,a_i)(\phi(s_i,a_i) - \phi((s'_i),\pi(s'_i))^T \qquad (23)$$

$$B = \frac{1}{L}\sum_{i=1}^{L} \phi(s_i,a_i)c_i. \qquad (24)$$

Finally, the policy improvement step defines each policy greedily in terms of the previous state-action value function, as before: $\pi_{m+1}(s) = \arg\min_{a \in \mathscr{A}_s} \phi(s,a)^T \theta^{\pi_m}$.

To apply LSPI to GC, we first need to select a set of basis functions. We consider each triple consisting of the current time step, the time step where the last collection occurred, and action at the current time step. Given that there are n distinct triples, we give each one a unique index in $[1,n]$. Our basis functions are the n-element vectors where one element is 1 and the rest are 0. (This is called a "one-hot" encoding of the triples.) Clearly the space of possible triples is very large, but the occurring triples are sparse in that space, giving an encoding that is $O(N)$ in practice rather than $O(N^2)$.

Next, we generate a collection of samples from the MDP. For every possible triple of time steps t' and t and actions a, we run the default policy until we reach time step t', at which point we perform a nursery collection. We then perform no collections until time step t, where we perform action a. We record these two time steps, as well as the cost incurred and the next state reached after taking action a. Some triples may not be feasible if t' and t are too far apart, in which case the triple is simply not recorded. Once we have this information, we run LSPI and compute an optimal schedule under these approximations. The time complexity of LSPI is approximately $O(N^3)$ per iteration due to the need to solve the matrix equation $\theta = A^{-1}B$ on each iteration.

4. Traces, Blocks, and Cohorts

As described by Ricci, Guyer, and Moss [2011, 2013], Elephant Tracks (ET) traces consist of a sequence of event records, each recording an event associated with control (method calls, returns, etc.) or with memory management (object allocation, pointer updates, object death). Like its intellectual predecessor Merlin [Hertz et al., 2006], ET computes precise death times for each object. Here we are interested in the subsequence of records pertaining to memory management, and these are entirely precise, lacking only a very small number of events that occur before ET can start or due to small deficiencies in the particular Java virtual machine's reporting of corner cases.

An object allocation record gives the object a unique ID (essentially a sequence number) and indicates its class and size, and for arrays, its number of elements. For our purposes we develop object sizes using a uniform model of Java objects for a 64-bit platform. Specifically, scalar (non-array) objects have a two-word header (one word = 8 bytes) plus their fields, packed as tightly as possibly while obeying alignment constraints, rounded up to a multiple of 8 bytes. Each Java primitive type is aligned to a boundary equal to its size (`char` and `short` to 2 bytes, `int` and `float` to 4 bytes, `long` and `double` to 8 bytes). Arrays consist of a three-word header and their elements, also rounded up to a multiple of 8 bytes.

We aggregate object allocations into 256KB blocks. We start a new block when the next allocation would result in the current block being larger that the block size. Thus, objects whose size is 256KB or more are allocated into (one or more) blocks by themselves. We consider such *large objects* to consume the whole sequence of blocks, i.e., their effective size is their original size rounded up to a multiple of 256KB. We distinguish large objects that contain pointers (in Java object arrays of objects can have this property) from those that don't, as previously mentioned. Allocation into the young generation is in terms of whole blocks, so if a particular block is less than 256KB long, there is some fragmentation waste at the end. We assume that old space is organized not to do this. Performing allocation in terms of blocks restricts possible GC points to block boundaries, significantly reducing computational cost in our optimization algorithms. We feel this is reasonable since any actual policy cannot perform a complex policy computation at each object allocation. Computing the policy after every block is more realistic.

ET object death records indicate the object that died, giving its ID. From the allocation and death records we can determine the volume of allocation in each block and the volume of live objects as of the beginning of the next block. This information is sufficient for computing our NG GC results.

For Gen GC we need also information about how objects refer to each other. ET's pointer update records enable us to model that, since they indicate the referring object, the target object, and the slot of the referring object that is being updated.

4.1 Generational GC Computation

There are three aspects of our approach to computing the cost of Gen GC that we now treat in detail. The first concerns a method to precompute information related to the object graph so as to simplify computation of the cost of Gen GC. In particular we introduce what we call the *pre-birth time* of an object. An object's pre-birth time, along with its birth and death time, determines when that object will be promoted by a young GC, *without needing to build or traverse the object graph*. The second aspect is the notion of *cohorts* of objects that will have the same behavior under Gen GC, and how to compute those cohorts. Cohorts group objects according to blocks and thus also reduce the cost of computing GC costs in both space and time. The final aspect is the details of how we model the young and old generations, the cost of young and full GCs, and when specific steps are allowed in a schedule.

4.1.1 Pre-birth Times

Given the object birth and death time information available from our traces, it is simple to compute the set of live objects at a full GC. Starting from sorted birth and death records, one can process a trace in order and track the live size at all time steps with constant work per trace record. As will be seen, our cost model for full GC is based on the volume of live objects, so it is therefore cheap to compute the cost of full GC at any time step.

Unfortunately, the situation is not so straightforward with young GCs. Of course a live object in young space will be promoted to old space, and a live object in old space will be retained. However, *dead* objects in old space will also be retained, and if they refer to dead objects in young space, those objects will also be promoted. We use the term *baggage* for such promoted dead objects, and they increase the cost of a young GC. They also take up room in the old space, possibly forcing a full GC sooner than would otherwise be necessary.

Consider an object x with birth time $x.b$ and death time $x.d$. Our unit of time is the number of objects allocated so far, so $x.b$ is the number of the objects allocated before x, and $x.d$ the number of objects allocated before x dies. Necessarily, $x.d > x.b$. Note: it will be helpful to consider the birth as occurring infinitesimally after $x.b$, i.e., at $x.b^+$, and the death as occurring infinitesimally before $x.d$, i.e., at $x.d^-$. This will clarify how a GC that occurs at time t affects any given object.

Given a dead object x, consider all objects that died at or before $x.d$, and in particular consider their pointers one to another. We observe that this part of the object graph will never change after x's death (the mutator cannot access dead objects and so cannot change them). Now consider the subset of these dead objects from which there are paths to x. We call these *predecessors* of x in the dead object graph. Let o be the predecessor of x whose birth time is earliest.

Claim: A young collection after $x.d$ will promote x *if and only if* there is a collection (young or full) after $o.b$ and before $x.b$, and no collection between $x.b$ and $x.d$. *Argument:* First suppose there is no collection between $o.b$ and $x.d$. Then collection after $x.d$ will reclaim, not promote, x and all its predecessors in the dead object graph. To see this we note that if y is dead and refers to x, then $x.d \geq y.d$, since if $y.d > x.d$, the lifetime of x would be extended. In other words, object x cannot die before another object y that refers to x at the time y dies. This property hold recursively. Thus all unreclaimed predecessors of x in the object graph were born at or after $o.b$ and died at or before $x.d$. They must all lie in young space and will be reclaimed by any collection that occurs after $x.d$.

Now we tackle the converse. Suppose there is a collection between $o.b$ and $x.b$ and no collection while x is live. We argue that there is some object y in old space that, via a chain of one or more references, refers to x and forces x to be promoted if there a young collection after $x.d$. Consider a chain of objects $o = o^0$, o^1, ..., o^n, $o^{n+1} = x$ where each of these objects refers to the next in the dead object graph. Since o^i refers to o^{i+1} (for $i = 0, \ldots n$), o^i and o^{i+1} have overlapping lifetimes (the mutator made o^i refer to o^{i+1}; they were both live at that moment). Therefore the chain of objects together spans lifetimes covering the range $o.b$ through $x.d$. Therefore, for any time t' such that $o.b \leq t' \leq x.b$, at least one object in the chain will be live at time t' and be promoted. This applies specifically to the latest such t'. Note that our argument depends on restricting collection to occur in *age order*. That is, if $x.b < y.b$ then y cannot be collected before x. Most collectors work this way because of faith in the generational hypothesis [Hayes, 1991], though collecting in this order is not a logical necessity.

A minor part of the converse is that collection between $x.b$ and $x.d$ will promote x, but this is obvious since x is live in that interval.

In our argument using the dead object graph, we considered the oldest dead object o from which the target object x is reachable. We call $o.b$ the *pre-birth time* of x, written $x.p$. The end result of this argument is that if the previous collection occurred after $x.p$, then young collection with x in young space will cause x to be promoted, *whether or not x is live*. The point is that a precomputation over the dead object graph can provide the additional per-object datum, $x.p$, sufficient for calculating young object promotions. We do not need to build the object graph and traverse it, simulating GC directly; we can just look at $x.p$, $x.b$, and $x.d$ to determine what will happen to x in any sequence of collections.

Furthermore, sets of objects can be sorted in terms of these times to make simulation of the effects of collection very efficient. Here is how such an efficient per-object simulation can proceed. We keep track of the time c when the most recent collection occurred. When a new object x is allocated, if $x.p < c$ then we add x to a baggage set, which is sorted by death time. If not, we add it to a non-baggage set, also sorted by death time. When a young collection occurs at time t, the baggage set objects are promoted, as well as non-baggage set objects whose death time is greater than t. The old space object set is also maintained in death time order. A full GC discards objects in all three sets (young baggage, young non-baggage, and old) whose death time is no more than t and puts the remaining objects in the old space set. Each object is handled at most three times (allocation, promotion, death), with each operation's cost logarithmic in the number of objects that can fit in the heap, i.e., logarithmic in heap size. This gives a total simulation cost of $O(n \log H)$ where n is the number of objects and H is the heap size. Simulating actual GC directly requires building and traversing object graphs with worst case cost $O(n \cdot H)$. So, for repeated simulations our precomputation can save a lot.

Here is how to compute pre-birth times efficiently. When processing an ET trace, maintain a model of the objects allocated in the heap. Associate with each object its pre-birth, birth, and death times. When processing an allocation event record, we allocate the new object in the model of the heap, assign its birth time according to the bytes allocated so far, and set its initial pre-birth time to its birth time. When processing a pointer update event, we update the heap model to reflect the pointer update. When processing a death record, we propagate the pre-birth time of the dying object to all objects to which the dying object refers. By "propagate" we mean to set the target object's pre-birth time to the minimum of its current pre-birth time and the pre-birth time being propagated, and if that reduces the target object's pre-birth time, propagate recursively to the objects to which the target refers. Since a number of objects may die at the same time, it is necessary to do all these propagations *before* removing any of the dying objects from the heap, since another object's death might cause pre-birth values to propagate through an object dying at the same time. Once all the deaths at a given time have been processed, the newly dead objects have stable pre-birth times, and their three times (pre-birth, birth, and death) can be recorded and the objects removed from the heap model. This will result in a trace of object information in death order. This trace will need to be sorted into birth order to be suitable for that heap simulation previously described.

4.1.2 Block-based Collection: Cohorts

As already revealed, a cohort is a group of objects that will behave the same under Gen GC. However, we need to unpack what we mean by "behave the same". We now describe how our Gen GC scheme works. Understanding Gen GC is necessary to understanding cohorts and how we compute them.

Our plan is to restrict collection to occur only at block boundaries. A primary motivation is that this will reduce dramatically the computation necessary to determine optimal costs, by reducing the problem size. For example, we have typically used a block size of 256KB. If the average object size is 50 bytes, this reduces the size of our optimization problem by a factor of about 5000. Furthermore, as we argue more carefully later, the optimization problem is not one that is easily decomposed into fully parallel subproblems, so we cannot simply throw 5000 times as many compute nodes at the problem—problem size determines the time required. Also, that time is not simply linear in the problem size. For example, the exact dynamic programming solution for the NG case has quadratic cost, and it is clear that the Gen case can only be worse. We can expect the cost to be $O(B^{-k})$ in term of block size B, where $k \geq 2$. A similar factor applies to the space needed by the dynamic programming computation.

If collection occurs only at block boundaries, we can group together objects according to blocks. In particular, we use the term *group* to refer to the set of objects allocated in a particular block (or a single object that consumes all of one or more blocks). We can map object pre-birth, birth, and death times to the group within whose allocation times those object times fall. We call these the pre-birth, birth, and death *groups* of the object. A *cohort* is a set of objects that share the same pre-birth, birth, and death groups. Given our previous result on pre-birth times, and the restriction that collection occur only on block boundaries, we can compute collection costs from a small number of facts about each cohort, greatly speeding simulation. In our argument concerning pre-birth times we explained how GC behavior could be simulated handling each object at most three times. This same strategy applies to cohorts, giving algorithms that handle each cohort only three times.

It is clear that restricting collection to occur only at block boundaries does not necessarily find the very lowest cost GC schedules. While we do not do so here, we believe that it is possible to develop bounds on the gap between optimal collection without restriction and with the block boundary restriction. For example, for NG GC, the error is less than one block per collection, and can be tightened further by considering the minimum live size within each block. It is rather more complex for Gen GC, so we leave detailed investigation to future work. Suffice it to say that it appears that getting an exact result in the unrestricted case appears infeasible for traces large enough to be interesting.

While collecting only at block boundaries is pragmatic, it is also realistic in that many collectors work on a block basis. A thread acquires one block at a time from a global pool, and then allocates sequentially (so-called "bump pointer" allocation) within the block. Making collection decisions at block boundaries is thus natural, though admittedly not all collectors work this way.

In sum, our precomputed pre-birth times allow us to reduce GC simulation cost from an $O(H)$ factor to $O(\log H)$, where H is the heap size, and the block boundary restriction reduces costs by a factor of about $(B/s)^2$, where B is the block size and s is the average object size.

4.2 Summary of Approximations

At this point it may be helpful to review the approximations and idealizations involved in our approach to determining optimal GC costs.

- We use bytes traced/copied as a proxy for actual GC cost. It is generally held by GC researchers that costs are largely proportional bytes copied, but sometimes other effects can be significant, which may include stopping threads and scanning their stacks, cache and TLB misses and page faults, etc. These effects would be very difficult to capture in an analytical model such as what we develop here.
- We allow collection only at block boundaries. This was discussed in more detail just above.
- For Gen GC our MDP is not exact in that it does not consider all possible schedules of young collections since the most recent full collection. This approximation appears necessary in order to make the optimization computations tractable. Put another way, we have approximated a deterministic variable-order MDP by a stochastic fixed-order MDP.
- We observe that our particular basis functions used in framing Gen GC optimization as an LSPI problem are *not* an approximation—they exactly render the MDP in a linear form. However, alternative basis functions could introduce an additional approximation.

5. Experiments

We now describe the experimental methodology we use to assess the proposed GC schedule optimization methods. We begin by describing the general experimental protocols and then summarize the set of program traces that we used in our evaluation.

5.1 Experimental Protocols

The primary question of interest in this work is how much better can the optimized GC collection schedules produced by the dynamic program and LSPI methods be, compared to schedules produced by the corresponding default policies? To answer this question, we evaluate each method on a benchmark suite consisting of multiple programs. Each program can be run with a variety of input files, producing a collection of program traces. The program traces we use are described below.

In the NG case, a particular program trace combined with a choice of heap size defines a unique deterministic MDP that we solve using the proposed dynamic programming algorithm. We evaluate the performance of methods for each trace using a range of heap sizes from the maximum live size of the trace to 5 times the maximum live size. The DP gives the absolute minimum cost achievable for each trace and heap size combination.

In the Gen case, each combination of a particular program trace combined with a choice of old and young heap sizes defines a unique variable-order MDP that we solve approximately using the proposed LSPI algorithm. We evaluate the performance of methods for each trace using a range of old heap sizes from the maximum live size of the trace to 5 times the maximum live size. We evaluate a number of absolute sizes for the young heap including 1MB, 2MB, 4MB, 8MB, and 16MB.

We compute results for single traces in terms of the mark/cons ratio of the default and optimized schedules over the full range of heap values. When reporting results over multiple traces or heap sizes, we aggregate the percent decrease in GC cost achieved by the optimized schedule compared to the corresponding default schedule.

5.2 Traces Used

We created traces using mostly programs from the DaCapo benchmark suite (version 9.12) [Blackburn et al., 2006, 2008]. For a number of these programs we developed additional inputs, namely `avrora`, `fop`, `jython`, `luindex`, `lusearch`, and `pmd`. We had difficulty getting some programs to run on our trace generation framework (Elephant Tracks [Ricci et al., 2011, 2013]) or with some "sizes" (inputs), particularly `tradesoap` and larger "sizes" of `tomcat` and `tradebeans`, so these traces were left out of the evaluation.

We added an additional benchmark, `javac`, which runs the HotSpot Java compiler. We wanted to synthesize behavior similar to a long-running server program, so we presented the compiler with a substantial number of Java source files to compile and forced it to compile them separately rather than together. We further manipulated things so that the compiler would release cached soft references between compilations, making it reload—as a server would have to do. We achieved our goal of a highly cyclic pattern of live sizes, growing

Benchmark	# of inputs	Max live (MB)		MB allocated		Groups (256KB Blocks)		Cohorts (256KB Blocks)		Groups (64KB Blocks)		Cohorts (64KB Blocks)	
		min	max	min	max	min	max	min	max	min	max	min	max
avrora	4	3	11	60	428	203	1678	1020	20238	809	6702	4328	61337
fop	14	4	34	36	1882	109	7160	1115	41180	439	29554	3363	124806
javac	6	11	17	185	3653	731	14366	10354	197875	2867	55158	28324	577961
jython	7	7	29	147	17954	554	71981	6445	181377	2198	290816	20522	698449
luindex	5	2	5	26	62	67	211	413	956	265	811	1389	3245
pmd	19	2	81	30	1178	83	4856	713	202940	325	18967	2261	618627
tomcat	1	10		496		1956		21976		7932		54895	
xalan	3	3	3	160	13280	623	55076	4233	321669	2646	235306	13328	1064039

Table 1: Summary of benchmarks used

during each compilation then dropping back to a baseline in between. We believe this server-like benchmark is an interesting case for which to find optimal GC policies. We ran the benchmarks using the IBM Java 6 Java virtual machine.

From the ET traces we extracted object allocation, mutation, and death events, which we consolidated into 256KB groups. We computed pre-birth times and cohorts. The cohorts were then used with the dynamic program to find absolutely optimal NG collection schedules and with LSPI to find approximately optimal ones for the Gen case. As previously mentioned, we used whole heap / old generation sizes from 1.0 to 5.0 times the maximum live size (rounded up to a whole block), with about 100 points in between to develop fairly smooth curves.

Since our collector requires the old space size to be at least as large as the young space size, if a program run has maximum live size that is particularly small, then we cannot produce results for our chosen range of young generation and old generation sizes. This causes several programs not to be included in our analysis, notably `batik`.

Our GC cost measure is the number of bytes marked / copied (not counting large non-pointer objects). As mentioned before, we generally report the "mark/cons" ratio, i.e., bytes marked/copied divided by bytes allocated.

Table 1 offers some summary information about each of the traces we used. Since a given program usually has a number of different inputs resulting in a number of different traces, we report sizes by giving the range of values.

6. Results

We now report the results of the experiments described in the previous section. We begin by illustrating optimized and default schedules in the Gen case for a single combination of program trace, young heap size, and old heap size. The results shown in Figure 1 are based on a single trace for `javac`. The young generation size was 8MB, and the old generation size was 3.25 times the maximum live size (approximately 39MB). Both panels show the live size curve for the trace. The markers in the top panel correspond to the collection locations for the schedule optimized using LSPI. The bottom panel shows the schedule obtained using the default policy. The solid markers indicate full collections while the hollow markers indicate young collections. The vertical axis has been trimmed to focus on the behavior of the program after the brief start-up phase. The horizontal axis is in terms of 256KB blocks of allocation.

In this case, the optimized schedule performs 77 young collections while the default schedule performs 56 young collections. We can also see that the optimized schedule performs only two full collections while the default schedule performs four. The optimized schedule achieves a 37% reduction in GC cost in this example. The reduction in cost of the optimized schedule is due to a combination of performing the young collections at locations with lower live size, and adjusting the number and location of young collections to avoid performing more costly full collections. In general, the schedules optimized by both LSPI and the dynamic programming algorithm achieve the largest gains by reducing the number of full collections.

To give a broader sense of the performance of the methods, Figure 2 shows the cost of the default and optimized schedules as a function of the full/old heap size. The same program and trace were used as in Figure 1. Figure 2(a) shows the default policy cost and the cost of the optimal schedule as found by dynamic programming in the NG case. Figures 2(b) and 2(c) show the default Gen cost and the cost of the schedule obtained using LSPI for two settings of the young heap size (4MB and 8MB). The most evident trend is that the total cost of all schedules decreases as the full/old heap size increases. This is due to the fact that a larger heap requires less collection. We can also see that the gap between the default schedule and the optimized schedule in the NG case tends to be quite small for this program trace over all heap sizes. In the Gen case, we can see that the gap widens as the size of the young heap increases, and we observed this trend across all traces.

Figure 3 summarizes the distribution of percent decrease in GC cost of optimized schedules compared to the corresponding default schedules in the NG (Figure 3(a)) and Gen (Figure 3(b-c)) cases. The results are broken out by program, but aggregated over all full/old heap sizes and all inputs to

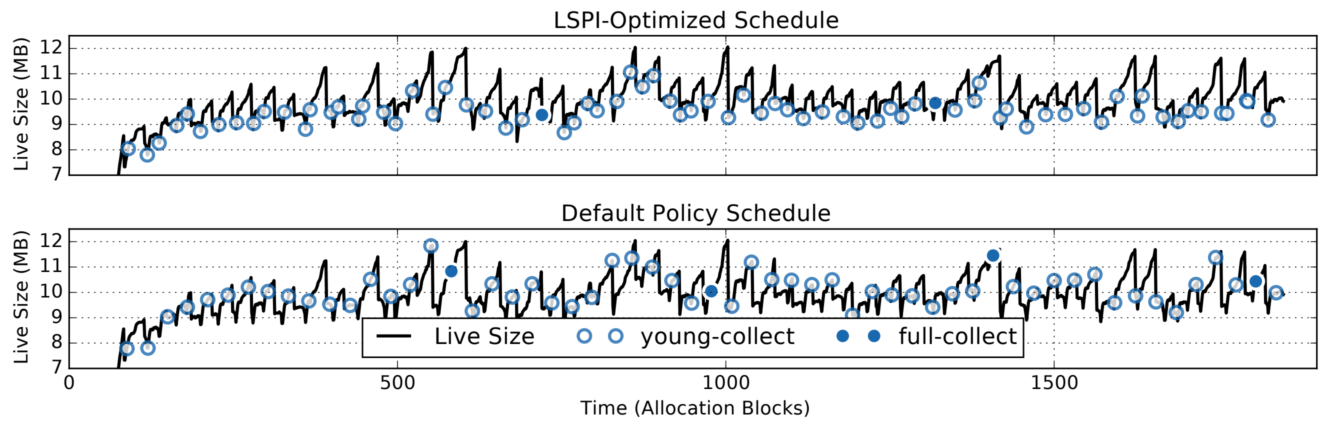
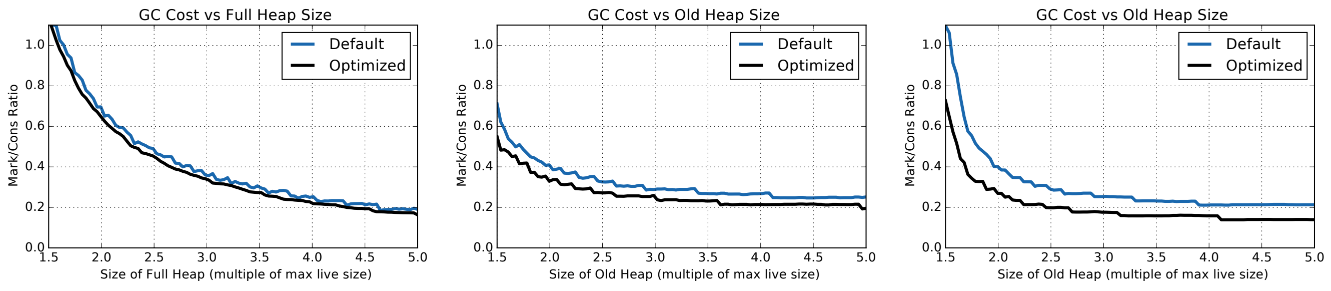
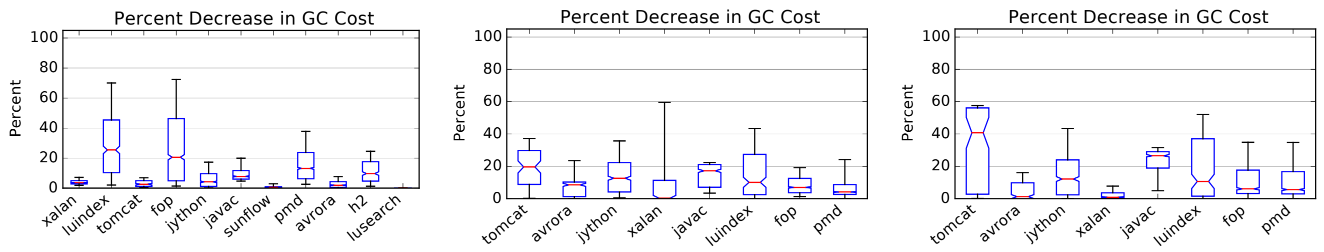


Figure 1: Example Gen collection schedules obtained using LSPI and the default policy



(a) Non-Generational (b) Generational ($Y^S = 4$MB) (c) Generational ($Y^S = 8$MB)

Figure 2: Examples of the relationship between cost and old/full heap size for the program run `javac-analyzetrace`.

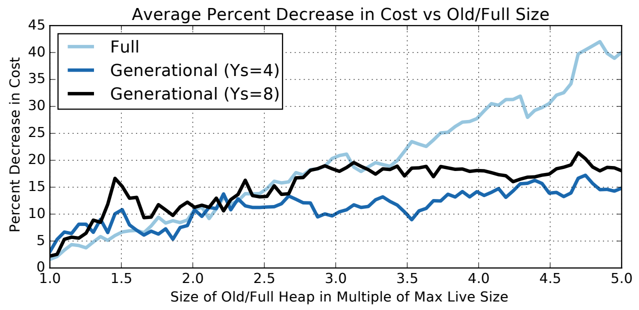


(a) Non-Generational (b) Generational ($Y^S = 4$MB) (c) Generational ($Y^S = 8$MB)

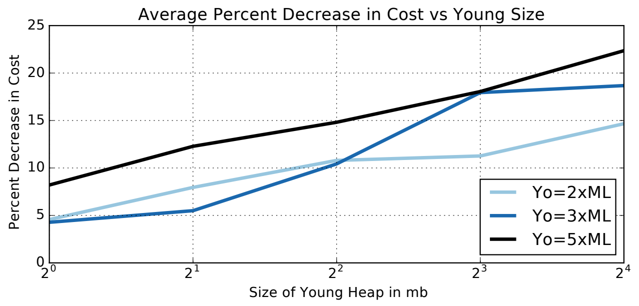
Figure 3: The percent decrease in cost relative to the default policy for the non-generational and generational cases

each program. The red lines in the box plot show the median percent decrease in cost, and the programs are ordered according to median decrease in cost for the NG case, with higher decreases to the right. The top and bottom of each box correspond to the 25*th* and 75*th* percentiles. The whiskers show the 0*th* and 100*th* percentiles. The NG results show that the median decrease in cost is modest for most programs (in the range of 5% reaching toward 10%), but that in some cases the savings can be much larger (20% median decrease in cost for `luindex`, `fop`, and `pmd`, with maximum decrease in cost about 75% for `fop`). We can also see that the median decrease in cost is sometimes larger and sometimes smaller in the Gen case, and that the median decrease in cost is larger when the young heap size is larger. We can also see an example of one program (`tomcat`), where the median percent decrease in cost obtained by the optimized schedule is in excess of 40%.

Figure 4 explores these trends further. Figure 4(a) shows the geometric mean of the percent decrease in cost over all program traces as a function of the full/old heap size. Results are included for the NG case and for the Gen case with two different young sizes. We can see that in the NG case we obtain larger average decreases in cost at larger heap sizes. This is sensible since the dynamic program has more room to optimize the schedule when the heap size is larger. We can also see a larger average decrease in cost when the young



(a) Decrease in cost vs old/full heap size



(b) Decrease in cost vs young heap size

Figure 4: The geometric mean of the percent decrease in cost over all program traces as a function of the full/old heap size (left), and as a function of the young heap size (right)

heap size is larger. However, for a fixed young heap size, the cost decrease profile does not increase much for heap sizes larger than 2 to 3 times the maximum live size. (The *absolute* improvements will tend to be larger for small heap sizes, of course; recall that these plots are of relative improvements.) Finally, Figure 4(b) performs a similar aggregation over all program traces, but shows the average results as a function of the young heap size for several values of the old heap size. The general trend is that performance gain increases as both the young heap and old heap sizes increase.

6.1 Smaller Block Sizes

Intuitively, one would think that using smaller block sizes would result in better performance—after all, smaller blocks means there are more points at which collection may occur. However, there is an opposing effect: there will be more fragmentation (space lost at the ends of blocks in the young space, and in old space in the case of objects larger than one block). Figure 5a shows a two-dimensional histogram of the reduction in cost. For each trace, old space size, and young space size, we plot a point corresponding to the optimizer improvement for block size 256KB (y axis) and 64KB (x axis) and accumulate a histogram, displayed here using colors, where darker colors indicate more points in a bin. We see that most points cluster along the $x = y$ diagonal. Notice that the color scale is logarithmic, so the visual spread of fainter colors represents a relatively small number of cases. (When plotted on a linear scale they nearly disappear.) There is no systematic improvement with a smaller block size. This suggests that, across a range of heap sizes, etc., the block size 256KB results are representative of what this optimization approach achieves for any (reasonable) block size. Qualitative examination of detailed plots for each trace and combination of heap sizes bears this out. The biggest differences seem to occur on short traces, where small changes tend to have larger impacts.

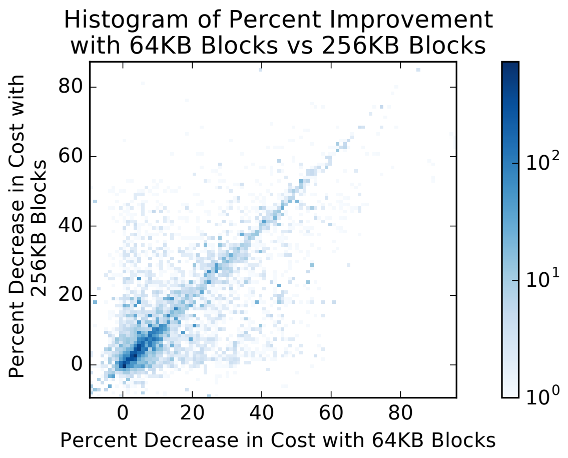
6.2 Cost of the Optimizer

How important was our choice to use blocks as opposed to allowing collection before any allocation? Figure 5b shows the time required to run our optimization procedure versus the length of the trace in blocks. This includes results for all blocks sizes (4KB, 16KB, 64KB, and 256KB) as well as all young space sizes (1MB, 2MB, 4MB, 8MB, and 16MB). Notice that the scales are both logarithmic. The figure also shows the best line fit, separately for each young space size, but stopping at 10^4 blocks since larger runs sometimes timed out, and would skew the plot if we extended the line fit. The lines are a good match to the equation $y = cx^{2.2}$, where c varies by young space size. In fact, c varies almost exactly proportionally to young space size. We speculate that this is because larger young spaces allow more choices of where to "place" young collections in a schedule. Individual points in the scatter plot are colored to correspond with the trend lines, and they show qualitatively the same increase in computation time for larger nurseries. In any case, these data show that using blocks was critical to making the optimization task feasible.

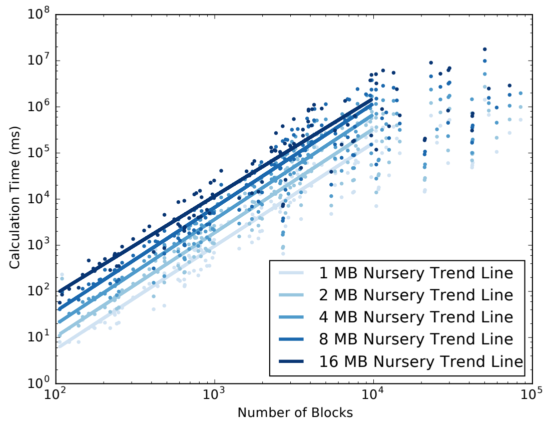
7. Realizing the Benefits

All a lower bound limit study can do is indicate where it is impossible to do better. Where it shows a gap, it *may* be possible to do better, but a limit study does not indicate *how* to get closer to the limit. It is still of value by itself because it indicates where further efforts at performance improvement might be possible, even worthwhile, and where they are not. It may also provide insight to help tackle the problem in new ways. All that said, we do have some ideas about how to create program-specific GC policies (in the broader sense of policy) that might approach the limits we have shown here.

The Elephant Tracks traces we collect contain much richer information than is needed for determining object pre-birth, birth, and death times. In particular, they include events for method calls and returns, and exceptions thrown and caught. Also, allocation events include the code location in the program (allocation site) and allocated class, etc. We have developed *feature vectors* for each 256KB block of allocation where a given feature indicates how many calls there were to a given method m during that window of allocation, etc.



(a) Histogram of improvements for two block sizes



(b) Scatter plot of optimizer time vs groups in trace

Figure 5: This figure shows the effect of block sizes on cost (left), and the effect of trace length on run time (right).

A typical program has around 30,000 features of this kind that are non-zero in one or more blocks. These long feature vectors, one for each block of allocation, appear to be a rich source of information for applying machine learning methods to induce a program-specific GC triggering function. Indeed, we can sometimes obtain interesting improvements when inducing a policy and applying it back to the same situation (same trace, same heap sizes(s)). The challenge is to induce a suitably *general* policy—one that works across many heap sizes and across many different inputs to the same program. We observe that trace information is entirely independent of heap size(s). Thus the rich feature set we have is not enough: there need also to be features relating to the current allocation and collection situation (how much space remains, etc.). Bolstered by the limit study results reported here, we remain hopeful of realizing a good portion of the benefit that appears possible.

8. Conclusions

We believe this is the first work to determine provably optimal GC costs (in terms of bytes marked/copied) for runs of real programs. We accomplished this in a specific, but realistic, block-oriented non-generational collector design. A second contribution is the introduction of pre-birth time, which allows an up-front analysis of the object graph, allocation, and death times to enable rapid determination of when dead young space objects will be promoted. Further, it enables, in a block-oriented scheme, the grouping of many objects together into *cohorts* that will always act the same in collectors that collect in age order (older objects no later than younger ones). Together, cohorts and blocks greatly speed determination of the number of bytes traced/copied for a given GC schedule. A third contribution is the modeling of GC as a Markov Decision Process, which clarifies why it is feasible to compute optimal GC schedules in the NG case (using dynamic programming), and why this is much more difficult in the Gen case. Finally, we introduce the application of Least Squares Policy Iteration to achieve optimized (but not necessarily optimal) schedules in the Gen GC case. Our empirical results show that there is room for program-specific GC policies to reduce GC cost consistently. Program-specific GC policies appear worthwhile for some programs, while for others the impact on the total performance of an application may not be worth the effort. While we do not offer detailed results here, we observed that optimized performance varied more smoothly with heap size than did performance with a default policy. This smoother trade-off of space versus time may be important for some kinds of systems.

Future directions include using machine learning methods to develop program-specific GC policies based on limited, focused instrumentation of a given program, and on methods to determine whether our LSPI approximation of optimal generational collection costs can be improved (or to bound what further improvement is even possible, e.g., by determining tight *lower* bounds on optimal collection cost). More broadly, the MDP modeling approach and approximate optimization techniques might be used to offer program-specific improvements related to other virtual machine services, such as just-in-time compilers.

Acknowledgments

We thank the reviewers for their many helpful suggestions. This material is based upon work supported by the National Science Foundation under Grant No. CCF-1320498.

References

R. Bellman. A Markovian decision process. Technical report, DTIC Document, 1957.

A. Bendersky and E. Petrank. Space overhead bounds for dynamic memory management with partial compaction. *ACM Transactions on Programming Languages and Systems*, 34(3):13, 2012.

D. P. Bertsekas. *Dynamic programming and optimal control*, volume 1. Athena Scientific Belmont, MA, 1995.

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 P. L. Tarr and W. R. Cook, editors, *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pages 169–190. ACM, 2006. ISBN 1-59593-348-4. doi: `10.1145/1167473.1167488`. URL `http://doi.acm.org/10.1145/1167473.1167488`.

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. *Communications of the ACM*, 51(8):83–89, 2008. doi: `10.1145/1378704.1378723`. URL `http://doi.acm.org/10.1145/1378704.1378723`.

H. Boehm. The space cost of lazy reference counting. In N. D. Jones and X. Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pages 210–219. ACM, 2004. ISBN 1-58113-729-X. doi: `10.1145/964001.964019`. URL `http://doi.acm.org/10.1145/964001.964019`.

B. Hayes. Using key object opportunism to collect old objects. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, ACM SIGPLAN Notices 26(11), pages 33–46, Phoenix, AZ, Nov. 1991. ACM Press. doi: `10.1145/117954.117957`.

M. Hertz, S. M. Blackburn, J. E. B. Moss, K. S. McKinley, and D. Stefanovic. Generating object lifetime traces with Merlin. *ACM Transactions on Programming Languages and Systems*, 28(3):476–516, 2006. doi: `10.1145/1133651.1133654`. URL `http://doi.acm.org/10.1145/1133651.1133654`.

R. A. Howard. *Dynamic Programming and Markov Processes.* The M.I.T. Press, 1960.

M. G. Lagoudakis and R. Parr. Least-squares policy iteration. *The Journal of Machine Learning Research*, 4:1107–1149, 2003.

M. L. Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

N. P. Ricci, S. Z. Guyer, and J. E. B. Moss. Tool demonstration: Elephant Tracks—generating program traces with object death records. In *Proceedings of the 9th International Conference on the Principles and Practice of Programming in Java*, pages 39–43, Kongens Lyngby, Denmark, Aug. 2011. ACM.

N. P. Ricci, S. Z. Guyer, and J. E. B. Moss. Elephant Tracks: Portable production of complete and precise GC traces. In P. Cheng and E. Petrank, editors, *International Symposium on Memory Management, ISMM '13, Seattle, WA, USA - June 20 - 20, 2013*, pages 109–118. ACM, 2013. ISBN 978-1-4503-2100-6. doi: `10.1145/2464157.2466484`. URL `http://doi.acm.org/10.1145/2464157.2466484`.

J. M. Robson. An estimate of the store size necessary for dynamic storage allocation. *Journal of the ACM*, 18(3):416–423, July 1971. doi: `10.1145/321650.321658`.

J. M. Robson. Bounds for some functions concerning dynamic storage allocation. *Journal of the ACM*, 21(3):419–499, July 1974. doi: `10.1145/321832.321846`.

J. M. Robson. Storage allocation is NP-hard. *Information Processing Letters*, 11(3):119–125, Nov. 1980. doi: `10.1016/0020-0190(80)90124-6`.

R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

R. S. Sutton, D. A. McAllester, S. P. Singh, Y. Mansour, et al. Policy gradient methods for reinforcement learning with function approximation. In *Neural Information Processing Systems (NIPS)*, volume 99, pages 1057–1063. Citeseer, 1999.