Modeling Heap Data Growth Using Average Liveness

Pengcheng Li, Chen Ding and Hao Luo
Department of Computer Science, University of Rochester, Rochester, NY14627, USA
{pli, cding, hluo}@cs.rochester.edu

Abstract
Most of today’s programs make use of a sizable heap to store dynamic data. To characterize the heap dynamics, this paper presents a set of metrics to measure the average amount of data live and dead in a period of execution. They are collectively called average liveness. The paper defines these metrics of average liveness, gives linear-time algorithms for measurement, and discusses their use in finding the best heap size. The algorithms are implemented in a Java tracing system called Elephant Tracks and evaluated using the DaCapo benchmarks running on the Oracle HotSpot and IBM J9 Java virtual machines.

Categories and Subject Descriptors D.3.4 [Processors]: Memory management (garbage collection); D.2.8 [Metrics]: Performance measures

General Terms algorithms, performance, measurement

Keywords liveness; all-window statistics; GC frequency; heap data growth

1. Introduction
In a heap, an object is live if it is allocated and not yet freed. The amount of live data can be measured at each time point in an execution. The amount is known as the population count. Traditionally, heap dynamics is shown by plotting the population counts over the entire execution. However, we often want to characterize a heap for a shorter period, e.g. a second. A problem arises because the population dynamics differs depending on the time window, and any window we select may not be representative.

This paper generalizes the concept of population count and defines window-based liveness. An execution window is a consecutive series of time points. The set of the live data in a window is the union of the population at all its time points. The liveness of the window is the size of the union set.

The liveness differs depending on which window we count. To have a deterministic metric, we define average liveness \(wl(k)\), which is the average liveness in all windows of length \(k\). The symbol \(wl\) designates window-based liveness or win-live for short. For any sequence of memory allocation and free events, \(wl(k)\) is uniquely defined.

Average liveness provides strictly more information than the traditional population count. Population count is defined for each time point. Average liveness is defined for every window, which includes all time points as unit-length windows. A population includes only the live data. Average liveness counts both live and dead objects. Mathematically there is a fundamental difference: the population count over time is not monotone, but average liveness over increasing window lengths is monotone.

Average liveness can be used to derive optimal heap sizes. First, it avoids a problem known as the measurement bias [19]. When the same program is run multiple times, the GC frequency changes because it depends on the initial condition of the heap. A different initial point can lead to a different GC frequency. Average liveness computes the average frequency, since it assumes that the initial point can anywhere. Second, for full-heap GC, it computes the average GC frequency for all heap sizes, so the most effective size can be identified. Third, for generational GC, it computes the minimal mature heap growth, which can be used to choose the size for the nursery heap to minimize the amount of premature promotion.

Average liveness is a collection of metrics counting objects differently based on the following factors:

- **Liveness and death.** In a liveness metric, we count the objects that are (or have been) live in a window. In a death metric, we count the number of objects that become dead in the window.
- **Window local and global.** An object, live or dead, is considered local in a window if it is allocated inside the window; otherwise it is global.
- **Reachable and actual.** The death of an object may be precipitated immediately when it becomes unreachable or later when its memory is actually reclaimed. We call them the reachable liveness/death and the actual liveness/death respectively.

In this paper we present mainly three novel findings. First, we define seven metrics of average liveness, give the algorithms to compute them precisely in linear time, and show the inclusion relation as a full lattice. Then we use average liveness to find the best application-specific heap and nursery size. Finally, we evaluate the average liveness of DaCapo benchmarks running on two different Java virtual machines.

The study has a few limitations. The goal of the work is formulating a new type of heap metric not optimizing a program or a garbage collector. The treatment of garbage collectors is simplistic and ignores much of the diversity and complexity in a real design. The best GC parameters are derived after running a program (but can be derived for all heap sizes). Next we will show that the new metrics are novel, compact, deterministic, and can be measured efficiently.
2. Average Liveness

The metrics can be divided into three groups, each containing the metrics with similar characteristics. This section first introduces common symbols, then gives the definitions and measurement algorithms for each group and finally formalizes their relationship in the last section.

2.1 Notations

We use the function symbol \( wl \) to denote liveness metrics and \( wd \) death metrics. Each function has two subscripts. The first is either global \( g \) or local \( l \). The second is either reachable \( r \) or actual \( a \). For example, \( wl_{g,r}(k) \) is the function for local reachable death. It counts the average number of objects that are allocated and become unreachable in length-\( k \) windows. We may use only the first subscript, e.g. \( wl_g \), when there is no need to specify whether the liveness is for reachable or actual lifetimes.

The metrics are defined for each execution trace. The length (logical time) of the trace is \( n \). The total number of objects is \( m \). For each object \( i \), the allocation time is \( s_i \), and the death time is \( e_i \).

2.2 Global Liveness

In a time window \( w \), for global liveness we count the number of objects live in \( w \). The average global liveness, \( wl_g(k) \), is the average of all \( n - k + 1 \) length \( k \) windows.

There are two metrics: global reachable liveness \( wl_{g,r}(k) \) and global actual liveness \( wl_{g,a}(k) \). The algorithms to compute the two are identical except that the death time is by reachability in the former and by actual reclamation in the latter. We show the algorithm as one to compute \( wl_g \).

It is too numerous to count window by window. Considering all window length \( k \), the total number of windows is \( \binom{n}{k} \) or \( \frac{n(n+1)}{2} \). First, we transform the problem to make it solvable.

Instead of counting window by window, we count lifetime by lifetime. Take object \( i \) whose lifespan spans \( < s_i, e_i > \). Consider how many \( k \)-length windows in which the object is live. Ignoring boundary conditions (i.e. \( s_i \geq k \) and \( e_i \leq n - k + 1 \)), object \( i \) is live in every \( k \)-length window starting from time \( s_i - k \) to time \( e_i \). In other words, \( A \) contributes to the live count in and only in \( e_i - s_i + k \) windows. Hence the transformation of the counting problem: summing the liveness in all windows is the same as summing the contributions by all objects. More formally, the equivalence means:

\[
wl_g(k) = \frac{\sum_{\text{all window } w} (\text{num live objects in } w)}{n - k + 1} \tag{1}
\]

Counting the contributions is done in four cases, shown in Figure 1. For Object 1, \( s_1 \geq k \) and \( e_1 < n - k + 1 \). It is live in \( e_1 - s_1 + k \) length-\( k \) windows. This is the steady-state case. Others are boundary cases, \( s_2 < k \) for Object 2 and \( e_2 \geq n - k + 1 \) for Object 3. The fourth case is \( s_i < k \) and \( e_i \geq n + k + 1 \) for Object 4. In all four cases, the correct count for the contribution is \( \min(e_i, n - k + 1) - \max(s_i, k) + k \). Therefore, we have

\[
wl_g(k) = \sum_{i=1}^{m} (\min(n - k + 1, e_i) - \max(k, s_i) + k) \tag{2}
\]

Algorithmic Complexity The total time cost to compute \( wl_g(k) \) for all \( k \) is \( O(n) \), where \( n \) is the length of the trace. The space cost is also \( O(n) \). The length of a window is between 1 and \( n \). If we consider a logarithmic rather than linear scale and let \( k \) range from 1 to \( \log n \), the space cost is reduced to \( O(\log n) \).

Monotonicity Average liveness in general and global liveness in particular is monotone. It shows the heap data growth for different window length \( k \). When \( k = 1 \), \( wl(1) \) is the average population (every time point is a length 1 window). When \( k \) increases, \( wl(k) \) increases by including the live data in consecutive time points.

Average Population Size A trivial use of global liveness is to compute the average population size, which is simply \( wl_g(1) \). At each time point, the population size gives the minimal heap size. It shows the degree of cohabitation of objects. The same number of objects may be live at different times or at the same time. The average population size shows the average number of objects that are live at the same time.

Approximation Our earlier paper gives a simple formula to approximate the global liveness by ignoring the boundary conditions [16]. While the approximation is not used in this paper, the formulation may help to understand, for example, the relation between average liveness and the average rate of object allocation and death.

2.3 Local Liveness and Global Death

In the second group of metrics, just one end of the lifetime matters to the calculation. For local liveness, the significant event is the allocation at \( s_i \). For global death, the significant event is the death at \( e_i \). The event is significant because it affects the count in the windows that contain the event.

The formulas are given in Eq. 3 and 4. They are identical except where \( s_i \) is used in Eq. 3, \( e_i \) is used in Eq. 4.

\[
w_l(k) = \frac{\sum_{i=1}^{m} \min(s_i, k, n - s_i + 1, n - k + 1)}{n - k + 1} \tag{3}
\]

\[
w_d(k) = \frac{\sum_{i=1}^{m} \min(e_i, k, n - e_i + 1, n - k + 1)}{n - k + 1} \tag{4}
\]

The enumerators in the two equations are the sums of all object contributions: the total local live counts in Eq. 3 and the total death counts in Eq. 4. For each one, the contribution is the minimum of four numbers. The first three are cases ordered by the event time: earlier than \( k \), between \( k \) and \( n - k + 1 \), and after \( n - k + 1 \) (provided \( k \leq n/2 \)). The last number, \( n - k + 1 \), is the maximal contribution, which means the object is counted in every window of length \( k \).
For example, if the object is allocated before \( k \) (\( s_i < k \)), it appears in all windows whose starting time falls in the range between 1 and \( s_i \). If \( k \leq s_i \leq n - k + 1 \), it appears in all windows that start in \( s_i - k \ldots s_i \). If \( s_i > n - k + 1 \), it appears in all windows that start in \( s_i - k \ldots n - k + 1 \). The cases for the global death count are identical except that the event time is \( e_i \) instead of \( s_i \).

We have just shown two nearly identical formulas to compute two seemingly different object counts: local live and global death. The reason is that both are counting “local” events in a window. In local liveness, it is an allocation happening inside a window. In global death, it is a death happening inside the window. Next we show the last metric, which depends on both ends of a lifetime.

### 2.4 Local Death

In a time window \( w \), the local death count is the number of objects allocated in \( w \) and then dead in \( w \). The average local death, \( wd_l(k) \), is the average of all \( n - k + 1 \) length \( k \) windows.

The equation to compute the average local death \( wd_l(k) \) is given in Eq. 5. The numerator counts the total local deaths in all windows. It uses a predicate function to select only the object lifetimes smaller than \( k \). For each lifetime, the contribution is computed as the minimum of four cases. The four cases are similar to those discussed before: \( s_i < k, s_i \geq k, e_i \leq n - k + 1, k - (e_i - s_i) \), and finally the total number of length \( k \) windows. For example, in the first case, the starting time \( s_i < k \), so it may be local only in windows starting in the range \( 1 \ldots s_i \), that is, at most \( s_i \) windows. Considering all cases, we have

\[
wd_l(k) = \sum_{i=1}^{m} I(v_i < k) \min(s_i, k - v_i, n - e_i + 1, n - k + 1)
\]

where \( I(v_i < k) \) is a predicate function, which equals to 1 if the condition is true and 0 otherwise, and \( v_i = e_i - s_i \).

### 2.5 Inclusion Lattice

The object counts presented in the preceding sections — global/local liveness/death — can be defined for either reachable or actual lifetimes, effectively doubling the number of metrics to 8. Two of them, local reachable liveness \( w_{l_{r,l}} \) and local actual liveness \( w_{l_{a,l}} \), are the same since both are concerned with only the allocation time. We denote it by \( w_l \) without the second subscript. In total, we have seven new metrics.

The seven metrics are related by inclusion. For example, all the objects counted in local live are also counted in global live. To understand these relations, we show a set of example lifetimes in Figure 2. The graph shows four example objects whose lifetime overlaps with a given execution window. The global liveness counts all four objects, while the local liveness counts only the two that are allocated inside the window. The global death counts the two objects whose death time is inside the window, while local death counts the two that are allocated inside the window. The inclusion extends to reachable and actual liveness. In particular, global actual liveness includes global reachable liveness.

The inclusion relations among the seven metrics are shown in Figure 3. Each inclusion relation is shown by a directed edge between the two metrics. If we view edges as a form of order, the relations form a partially ordered set. It has the greatest element, global actual liveness \( w_{r,l} \), and the smallest element, local actual death \( w_{d_l,a} \). Because of the maximum and minimum, the partially ordered set qualifies as a full lattice.

We call Figure 3 the inclusion relation lattice.

---

1 In the lattice theory, a full lattice is one where upper-bound and lower-bound elements always exist for any set of lattice elements.

---

![Figure 2: Illustration of the relations among four types of counts by showing how four example objects are counted differently in a given window.](image1)

![Figure 3: The inclusion relation lattice of the seven metrics. The object count at the source of an arrow includes the object count at the sink of the arrow.](image2)

**Non-negative Differential** We can compare among liveness metrics. Some are helpful to understand program behavior, for example, the difference between the liveness and death counts. Some are helpful to evaluate garbage collectors, for example, the different counts between the actual and the reachable objects in a heap.

The inclusion relation lattice shows which pairs of metrics have always non-negative differences. These include all directly or indirectly connected pairs in the lattice. The maximum is the difference between the top and bottom metrics of the lattice. It shows the upper bound of the possible difference between any two metrics. In the evaluation section, we will show the non-negative difference between global actual liveness and global actual death.

### 3. Deriving Optimal Heap Sizes

In this section, we first explain the advantages of average liveness and then use its metrics to compute the data growth in a heap and analyzes the performance of garbage collectors.

#### 3.1 Avoiding Measurement Bias

We define the GC rate as the GC frequency in a unit of time. The reciprocal is the GC interval or inter-GC time. The time here may be real or logical. A logical time may be the total number of allocated objects or the total amount of allocated memory.

Evaluating a garbage collector is tricky. After the first GC, the following GCs happen whenever the heap is full. However, the time of the first GC is not identical if we run the same program.
multiple times. It depends on initialization by the JVM, which varies from machine to machine and run to run. Mytkowicz et al. called such variation the measurement bias and showed that such bias is prevalent and significant in all systems [19]. For Java, the use of a just-in-time (JIT) compiler further complicates the matter since the compiler may allocate a significant number of objects before and during a program execution.

Blackburn et al. gave the widely adopted solution known as the second-run methodology. The method forces initialization and JIT compilation to happen in the first run and then runs the program again to measure performance [7]. In this way, the result does not vary and can be reproduced.

The second-run solution ensures reproducible behavior, but it is not always the behavior seen by a user. This weakness does not necessarily invalidate the empirical conclusions drawn from second-run experiments. To characterize program behavior under all executions, the second-run result is incomplete.

Average liveness measures the expected behavior. It is the average count of all windows and therefore all possible GC windows. Window-based liveness does not evaluate garbage collectors as the second-run methodology does, but it evaluates the memory demand of a program in ways independent of specific GC windows, heap sizes, and initial conditions. In addition, the new metrics can be computed in a single pass. We may call average liveness a single-run solution to the problem of measurement bias.

3.2 Full Heap GC Frequency

We first consider mark-sweep GC. Let \( y \) be the heap size, i.e. the maximal size of data that the heap can store. The average GC interval \( gci(y) \) can be computed using the global reachable liveness as follows:

\[
gci(y) = \frac{1}{g,a} w \mid_l \mid_r(y)
\]

(6)

The derivation is illustrated in Figure 4. The plot shows the average growth of live data in the heap. When it reaches the heap size \( y \), GC is triggered. The window length \( t \) is then the average GC interval.

![Figure 4: Computing the full-heap GC rate using the global reachable liveness](image)

The computed interval is an average. It assumes that the GC window could be any window of the length of the interval. The computed interval is for all heap size \( y \). The larger the heap size is, the longer the GC interval. If the heap size is infinite, the GC interval is also infinite.

**Heap Size Selection** The shape of the liveness curve shows the tradeoff between the heap size and the GC interval. Adding heap space reduces the GC frequency more effectively when the slope is steep than when it is flat. By examining the cost-effect, a user may choose the best heap size.

3.3 Minimal Mature Space Growth

Generational GC exploits the common case that most objects die young. In the basic form, generational GC divides the heap into two spaces: the nursery and the mature space. New objects are allocated in the nursery. When it is full, GC is incurred but only inside the nursery. A nursery GC is also called an incremental GC.

In this paper, we assume a simple design. Both the nursery and the mature space have a fixed size. After the nursery GC, the uncollected objects are moved (or “promoted”) into the mature space. When the mature space is full, a full-heap GC is performed.

To compute the frequency of the nursery GC, we extend the previous formulation and make mainly two changes. The analysis is illustrated in Figure 5.

![Figure 5: Computing the nursery and full-heap GC interval in generational GC. The local liveness is used to compute the nursery GC interval. The difference of local live and local reachable death gives the lower bound on the growth rate of the mature space.](image)

First, we use the local liveness to model the nursery GC. Since only the objects allocated after the last collection occupy the space in the nursery, the inter-GC time for the nursery is simply the size of the nursery divided by the local liveness. Upon a nursery collection, the dead objects allocated after the last collection are collected.

Figure 5 shows the local liveness of a hypothetical program. The allocation starts from \( y = 0 \), because the nursery is initially empty. When the allocation fills the nursery, a nursery collection is triggered.

The second difference in generational GC is that the nursery heap is emptied after each GC, as the remaining objects are moved to the mature space. The process repeats when the emptied nursery is re-occupied by new allocation. The inter-GC time is still the size of nursery divided by the local liveness.

The mature space grows steadily as the uncollected objects are copied from the nursery. Figure 5 illustrates the growth curve of the mature space. The nursery acts as a filter to remove the objects that die young. An important metric is the survival rate, which is the average fraction of the nursery that survives the nursery GC.

The survivors include locally live objects, which can be counted by taking the local liveness and removing the local reachable death. The result is the minimal mature space growth \( msg \):

\[
msg(k) = w_l(k) - w_d,l_r(k)
\]

(7)

where \( k \) is the window length when \( w_l(k) \) equals to the nursery size, \( w_l \) is local liveness and \( w_d,l_r \) local reachable death.

Not all survivors are live. In particular, they may be dead objects referenced by other dead objects in the mature space. Eq. 7 counts only the truly live survivors, so it gives the lower bound of the mature space growth.

Premature promotion is the chief weakness of the original generational GC. It is part of the fundamental tradeoff: the generational
GC saves the collection time by granting immunity to the mature space, but it fails to collect dead mature objects or dead nursery objects pointed to by dead mature objects. The problem has been a focus of much of the past research, for example, the Beltway collector [6].

Without average liveness, it would be costly to measure the amount of premature promotion. We need a full-heap GC performed with every nursery GC. Even then, the result is specific to the specific execution, and the result changes when the initial GC time shifts (i.e. the measurement bias) and when the nursery size or the mature space size changes.

In evaluation, we will compute the lower bound growth for all heap sizes using Eq. 7. The difference between the actual survival and the minimal survival will show the average amount of premature promotion in an execution. It shows the efficiency of the generational GC in different heap configurations.

In future work, we plan to investigate a more direct method, which is to compute the average number of objects that are allocated and dead in a window but are pointed to by an object allocated outside the window. The extension would add a new type of average liveness in addition to global and local liveness.

4. Evaluation
In this section, we describe the experimental setup, the liveness result of DaCapo benchmarks, and the use of results in choosing the best heap size.

4.1 Experimental setup
Implementation using Merlin and Elephant Tracks We have implemented the new metrics using two existing systems, Merlin [13] by Hertz et al. and Elephant Tracks [20] by Ricci et al. Merlin [13] is the most precise technique for object lifetime profiling (OLP). It uses a forward pass to mark allocation times and potential death times for all objects and then a backward pass to (transitively) compute the exact death times. Using the precise object lifetimes measured by Merlin, we count reachable/unreachable objects.

Elephant Tracks [20] is a publicly available heap profiling tool. It captures a precise and complete record of object allocation, death and pointer updates. It monitors method entries and exits and uses the frequency as logical time. Elephant Tracks uses the standard JVMTI tool interface [18], so it can profile any JVM that has the interface, including Oracle HotSpot and IBM J9.

In more detail, the JVMTI interface [18] specifies a number of hooks for programmers to monitor JVM internal events. Elephant Tracks registers the hooks including VMInit, VMDeath, GarbageCollectionStart, GarbageCollectionEnd, ObjAllocation, ObjDeath and PointerUpdates. The Merlin reachability analysis is the backward pass in Merlin using the VMDeath hook inserted by Elephant Tracks. The liveness analysis primarily uses VMInit, VMDeath, ObjAllocation and GarbageCollectionEnd hooks. At a VMInit hook, we initialize the necessary data structure. At an ObjAllocation hook, we record the allocation time in a histogram. Similarly, we use another event to maintain a histogram of object death times. At a GarbageCollectionEnd hook, the GC time is recorded. All the object actual death times are estimated with the closest GC time after they become unreachable. Finally, at a VMDeath hook, the reachability analysis finishes, and the liveness analysis follows. Special operations such as exception handling and weak references are handled separately, in ways consistent with the solution in Elephant Tracks [20]. All our metrics are computed in a single pass in linear time.

Platform All experiments were conducted on an eight-core machine shipped with two Intel Xeon E5520 3.2GHz processors and running 2.6.34 Linux kernel. We tested two JVMs. On Oracle HotSpot OpenJDK7, we tested two garbage collectors: ParallelGC and ConMarkSweepGC. The initial heap size and maximal heap size were set to be same, 1000MB. For experiments that require explicit setting of the nursery size, we set the initial nursery size and the maximal nursery size the same. On IBM J9 1.6.0, we tested two garbage collectors: a full-heap GC called optthruput and a generational GC called gencon. For optthruput, the initial and maximum heap sizes were 4MB and 512MB. For gencon, the initial and maximum heap sizes were both set to 1000MB.

Cost of analysis On our machine platform, Elephant Tracks incurs over 500 times slowdown on average. Our liveness analysis is in linear time and the cost is negligible compared to the cost of instrumentation. Most of the space cost is due to object allocation time histograms and object death time histograms. Our liveness analysis is an online algorithm and does not store traces.

Test Programs For testing, we used the DaCapo benchmarks [7]. On Oracle HotSpot, we tested five DaCapo programs with two inputs, default and small. The programs are luindex, pmd, sunflow, fop and avrora. The five programs show a wide range of characteristics as we will see. The other seven programs are tested on IBM J9 using the default input and the small input except for the program h2, which uses only the small input.²

4.2 Liveness Trends
Figures 6 and 7 show the average liveness for the twelve DaCapo programs measured on HotSpot and J9. The x-axis is the window size from 1 to n, the length of the execution. Each window size has seven object counts. For easy viewing, we connect the points of the same metric and refer to a metric by its curve as we will see. The other seven programs are tested on IBM J9 using the default input and the small input except for the program h2, which uses only the small input.²

² We did not finish testing the default input before the submission of the final paper.
Figure 6: Average liveness metrics measured on HotSpot for luindex, pmd, sunflow, fop, avrora and on J9 for batik. "wL_{g,a}" in the figures denotes wL_{g,a}.
Figure 7: Average liveness measured on J9 for eclipse, jython, lusearch, tomcat, xalan, h2. "wL.g.a" in the figures denotes wL,g,a.
As the window size increases, both sides of the shaded area grow in parallel at the rate of new object allocation. The growth rate changes at a larger window size when GC frees data that are live at a smaller window size. This is shown when the shaded area narrows.

At the window length \( k = n \), there is no penalty. The reason is more subtle, because the liveness count at larger \( k \) includes both live and dead objects. At \( k = n \), it counts all objects, which are both live and dead in the window. The overlapping counting is also the reason that a liveness curve always keeps growing with the window size even though the size of live data is usually bounded.

In the experiments on HotSpot, we have tested two input sizes and each on two garbage collectors, for a total of four tests per program. In the figures, we choose to show one result for each program: the one that has striking shaded area. In the experiments on IBM J9, we found little or no variation between actual and reachable liveness. The reason is that J9 performs many garbage collections during an execution. The actual death time is computed from the closest GC time, which is very close to the reachable death time. Hence, the shaded area is small. The cause for this effect is still being investigated.

### 4.3 Finding Best Heap Size

First we consider full-heap GC. At each collection, a collector has to traverse all objects, so we define the effort of a collection by the amount of the data in the heap at the time of the collection. The GC effort is the sum of the effort of all collections.

We use the GC effort to select the best heap size. In a small heap, GC is efficient but more frequent. It may take multiple GC passes for an object to become dead and collected. In a larger heap, each GC takes longer, but the collection of the same object may take just one pass. It depends on the frequency of the GC, which we can compute using average liveness, i.e., by Eq. 6 in Section 3.2. Then the GC effort is the heap size times the GC frequency.

As discussed in Section 3.1, if we naively test and find the best heap size, the result may be wrong because of measurement bias. Average liveness gives the best heap size on average, so the result is not biased.

Figure 8 shows the GC effort for all test programs on the default input (a,c) and the small input (b,d), except for h2 in (d) which uses the small input. In these graphs, lower means less GC effort for the same mutator computation. The programs shown in (a,b) allocate a small amount of data. The total allocation is less than 70MB in all but two cases. When the heap size exceeds the total allocation, the effort is zero since no GC is needed.

The programs in (c,d) allocate more data. The GC effort is as high as 1.6GB. The effort decreases as the heap size increases. The change is not proportional. All programs have an inflection point, beyond which increasing the heap size no longer reduces the GC effort as much. In fact, in the majority of the cases, there is no benefit going beyond the inflection point. Before the inflection point, a larger heap means less GC effort. After the inflection point, a larger heap means the same GC effort.

To choose the best heap size, we need to compute the inflection point, which we can use average liveness. Figure 8 shows that the inflection point differs for different programs and inputs. However, at least two programs, lusearch, eclipse, have similar inflection points in their two inputs.

For these tests, the inflection point happens fairly early. Still, there are benefits in increasing the heap size beyond the inflection point. For sunflow, eclipse, the GC effort continues to drop for all heap sizes tested (up to 200MB).

### 4.4 Finding Best Nursery Size

We evaluate the (in)efficiency of a nursery size by the amount of premature promotion. Using average liveness (Section 3.3), we compute the minimal mature space growth. This is the ideal survival rate. There is no premature promotion. This is done through one-pass profiling. Then we test using different nursery sizes to measure the actual survival rate.\(^3\) For lack of a better name, we call the difference the zombie rate. The zombie rate is the premature promotion, which we want to minimize.

Figure 9 shows the three results for all test programs. The default input is used for all except for h2. The programs are divided into two groups. The first group were tested on HotSpot, and the second on J9. The graphs show the nursery size on the x-axis. It increases from 4MB to 50MB for the programs running on HotSpot, and 3MB to 100MB for the programs running on IBM J9. The total heap size is 1000MB, partitioned between the nursery and the mature space.

When looking at the survival rate, there are no consistent trends for the five programs running on HotSpot. The rate ranges between 20% and 80%. As the nursery size increases, the rate may decrease or increase. The variation is large. The smallest is 20% (of the nursery size) in fop. The other four are over 35%. The relation between the survival rate and the nursery size defies analysis since the variation is large and unpredictable. However, there is consistent trends for most programs running on IBM J9, except batik.

When we divide the survival rate into two sub-components, we observe three fairly consistent trends. First, the minimal survival rate decreases as the nursery size increases, except for luindex. Second, the variation of the minimal survival rate is small, less than 5% for most programs except luindex, batik, xalan and less than 2% for sunflow, avorara. Finally, in most programs, zombies contribute to the majority of the survived data. In sunflow and avorara, the vast majority are zombies. The minimal survival rate is 1% or less, while the zombie survival rate is over 70% when the nursery size is between 15MB and 40MB. In batik, python, eclipse, h2, tomcat, as nursery size increases, zombie rate goes bigger. For best performance, we may choose the nursery size to be as small as possible while minimizing the zombie rate.

5. Related Work

In this section, we first discuss solutions to a similar problem in locality analysis and then review existing techniques for heap analysis.

### All Window Analysis

In locality analysis, a recent concept is the footprint, which is the average amount of data accessed in a time window. Like average liveness, footprint analysis needs to measure for all \( O(n^2) \) execution windows. Xiang et al. gave two solutions. The first algorithm counts all windows efficiently and gives the distribution, including maximum and minimum, of footprints for each window length. It takes \( O(n \log m) \) time, where \( n \) is the length of a trace and \( m \) the size of all data \([10, 26]\). The second solution takes linear time and computes the average footprint in all windows without having to enumerate the footprint in all windows \([27]\). A recent extension is an algorithm that measures the shared footprint in all execution windows \([17]\). These algorithms are very different from average liveness. For footprint, we count the number of distinct data accessed in a window. For liveness, we count the number of live data.

The footprint theory is for cache management \([28]\). Average liveness is for heap management. The two theories differ. In the former, there is a higher order relation between cache performance.

\(^3\) The actual survival rate result is susceptible to the problem of measurement bias.
metrics. In this paper, there is the inclusion relation lattice. As a function, the footprint is not just monotone but concave. Average liveness is monotone.

**Object Lifetime Profiling (OLP)** Object lifetime is key to understand heap dynamics and evaluate heap efficiency. An early model by Röjemo and Runciman divides the lifetime and identifies the unnecessary parts as lag and drag [21]. Lifetime is defined by access. For garbage collected systems, lifetime is usually defined by reachability. Several techniques measure the lifetime using reference counting (RC) [4, 5]. While RC cannot detect death for objects involved in cycles, several variants [3, 24] use heuristics to detect dead cycles often by introducing periodic tracing collection or trial deletion. The solutions, however, do not guarantee precision. Merlin [13] uses a backward pass to compute transitive closure to find object death accurately. Although Merlin computes object lifetimes in linear time, it still incurs significant time and space overhead. Resurrector is a recent technique [29] to improve OLP performance. It identifies dead objects by timing method invocations on objects in conjunction with the use of reference counting. It does not detect dead cycles. Neither does it transitively discover the death times for objects that are pointed to by earlier dead objects. Other techniques such as [22, 25] proposed to use object lifetime profiles to improve GC performance. This paper uses Merlin and extends the use of OLP to include the seven new metrics of average liveness.

**Heap Performance Characterization** It is desirable to have machine and VM independent metrics to characterize the memory demand of a program. To examine the efficiency of generational GC, Ungar evaluated the relation between the survival rate and nursery size [23]. The evaluation is specific to generational GC. Dufour et al. measured the memory usage by the amount of allocation per unit of time [11]. The allocation intensity is not a complete measure of memory usage since it does not consider memory reuse due to reclamation. Dieckmann and Holzle measured the size of live data by forcing a full-heap GC every 50KB of allocation [9]. They showed the result in a time series and found important facts...
Figure 9: The efficiency of generational GC of different nursery sizes (in a 1000MB heap), measured by the actual survival rate (a), the minimal survival rate (b), and the difference, i.e. the zombie survival rate. The zombie is the portion of the dead objects survived due to premature promotion.
about the SPEC JVM programs. For example, the maximal live data size is reachable relatively early in execution. Blackburn et al. used a similar strategy to evaluate DaCapo benchmarks and compare them with SPEC JVM [7]. In addition to time series, these and other studies measure the histogram of lifetimes, known as object demographics [7, 14]. Others measured the connectivity in object graphs [12].

Kim and Hsu [15] studied the memory reference behavior of SPEC JVM applications by an exception-based tool. They observed that short-lived objects are more numerous in Java programs, small initial heap sizes are inefficient, and there exists an optimal heap size based on testing. In this paper we present the metrics for choosing the best heap size statistically. A set of papers [1, 2, 8] predicted heap memory requirements for garbage-collected languages. Braberman et al. [8] presents estimates based on program regions in particular methods. Their technique use static factors such as inputs and the program structure. In comparison, our metrics estimate the memory requirement in time windows. Albert et al. [1] describes an analysis to infer an accurate upper-bound on the peak heap usage. A later study [2] takes object lifetimes as parameters to give a conservative estimate of the memory requirement. If the estimated demand is met, a program should execute correctly.

6. Summary
In this paper, we have defined seven metrics: global actual live, global reachable live, local live, global reachable death, global actual death, local reachable death, and local actual death. They give the average count of live and dead objects in all execution windows in a program. They can be measured in linear time. They are monotone and have inclusion relations that form a complete lattice. The new metrics can be used to compute the average GC intervals for heaps of all sizes and the minimal mature space growth for nursery of all sizes, without having to run a program multiple times. The result can be used to find the best heap size in mark-sweep GC and the best nursery size in generational GC. The evaluation shows these metrics and their uses using the DaCapo benchmarks.

Acknowledgments
We thank Matthew Hertz, Eliot Moss and Rathan Ricci for their help with the use of Merlin and Elephant Tracks. We also thank them and the reviewers of MSPC 2013 and ISMM 2014 for the review and critiques especially for improving the presentation of the theory and the evaluation.

References
A. Monotonicity Proof

Theorem A.1. Monotonicity of Global Liveness.
The global liveness $wl_g(k)$ is monotonically non-decreasing.

Proof. We show $wl_g(k+1) - wl_g(k) \geq 0$. For the left-hand side to be minimal, we should have $wl_g(k+1) - wl_g(k)$, $n - k + 1 \leq e_i$ and $k \geq s_i$. Under these conditions, we have:

$$wl_g(k+1) - wl_g(k)$$

$$= \sum_{i=1}^{m} \min(n-k, e_i) - \sum_{i=1}^{m} \max(k+1, s_i) + mk + m$$

$$\geq \sum_{i=1}^{m} ((n-k) - (k+1)) + mk + m$$

$$= m(n-k) - m(n-k+1)$$

$$= m - m$$

$$= 0$$

Theorem A.2. Monotonicity of Local Liveness.
The local liveness $wl_l(i)$ is monotonically non-decreasing.

Proof. We show $wl_l(k+1) - wl_l(k) \geq 0$. For $wl_l(k+1) - wl_l(k)$ to be minimal, we have $n - k + 1 \leq s_i$, $n - s_i + 1, k$ and the following inequality:

$$wl_l(k+1) - wl_l(k)$$

$$= \sum_{i=1}^{m} \min(s_i, k+1, n-s_i+1, n-k)$$

$$\geq \sum_{i=1}^{m} (n-k)$$

$$= m(n-k)$$

$$= m - m$$

$$= 0$$

The global death $wd_g(k)$ is monotonically non-decreasing.

Proof. We show $wd_g(k+1) - wd_g(k) \geq 0$. For $wd_g(k+1) - wd_g(k)$ to be minimal, we have $n - k + 1 \leq e_i$, $n - e_i + 1, k$ and the following:

$$wd_g(k+1) - wd_g(k)$$

$$= \sum_{i=1}^{m} \min(e_i, k+1, n-e_i+1, n-k)$$

$$\geq \sum_{i=1}^{m} (n-k)$$

$$= m(n-k)$$

$$= 0$$

Theorem A.4. Monotonicity of Local Death.
The local death $wd_l(i)$ is monotonically non-decreasing.

Proof. We show $wd_l(k+1) - wd_l(k) \geq 0$. In the following derivation, the first "$\geq"$ is because $wd_l(k+1)$ contains more objects than $wd_l(k)$. For $wd_l(k+1) - wd_l(k)$ to be minimal, we have $n - k + 1 \leq s_i$, $n - e_i + 1, k - v_i$ and the second "$\geq"$ relation shown below:

$$wd_l(k+1) - wd_l(k)$$

$$= \sum_{i=1}^{m} I(v_i < k+1) \min(s_i, k+1 - v_i, n-e_i+1, n-k)$$

$$\geq \sum_{i=1}^{m} (n-k)$$

$$= m(n-k)$$

$$= 0$$