Efficient Nursery Sizing for Managed Languages on Multi-core Processors with Shared Caches

Mohamed Ismail

G. Edward Suh

Cornell University

Ithaca, NY, USA

{mii5,gs272}@cornell.edu

Abstract

In modern programming languages, automatic memory management has become a standard feature for allocating and freeing memory. In this paper, we show that the performance of today's managed languages can degrade significantly due to cache contention among multiple concurrent applications that share a cache. To address this problem, we propose to change the programs' memory access patterns by adjusting the nursery size. We propose *Dynamic Nursery Allocator (DNA)*, an online dynamic scheme that automatically adjusts the nursery sizes of multiple managed-language programs running concurrently without any prior knowledge or offline profiling. The experimental results on a native Intel machine show that DNA can significantly improve the system throughput by 16.3% on average and as much as 73% over today's nursery sizing scheme when four applications run concurrently sharing the last-level cache.

CCS Concepts • **Software and its engineering** → **Garbage collection**; Scripting languages; • **Computer systems organization** → *Multicore architectures*.

Keywords automatic memory management, shared caches, managed languages, Python

ACM Reference Format:

Mohamed Ismail and G. Edward Suh. 2020. Efficient Nursery Sizing for Managed Languages on Multi-core Processors with Shared Caches. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization (CGO '20)*, February 22–26, 2020, San Diego, CA, USA. ACM, New York, NY, USA, 15 pages. <https://doi.org/10.1145/3368826.3377908>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org. *CGO '20, February 22–26, 2020, San Diego, CA, USA*

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-7047-9/20/02...\$15.00

<https://doi.org/10.1145/3368826.3377908>

1 Introduction

Automatic memory management is used in many modern programming languages as a way to allocate and free objects without explicit programmer involvement. The programmer can focus on writing functional aspects of code instead of having to worry about correctly allocating objects and debugging memory bugs related to improper allocation or freeing of objects. In most cases, a language runtime will allocate objects on demand as needed by the program and use periodic *garbage collection* to free dead objects. In order to amortize its cost, garbage collection must be run at infrequent intervals. In some cases, the cost of automatic memory management can be less than the cost of stack allocation [3].

In this paper, we show that the performance of automatic memory management can be significantly affected by cache sharing among multiple concurrent applications, and propose *Dynamic Nursery Allocator (DNA)*, a new dynamic memory management scheme that automatically adjusts managed memory at run-time considering the cache interference. Unfortunately, handling cache interference in the context of multiple applications in managed languages has not been adequately explored previously. We found that cache thrashing can happen even with as few as two to four concurrently running applications. This problem is particularly important for modern multi-core processors with shared caches, whether on a personal device running multiple instances of JavaScript or on a Platform-as-a-Service (PaaS) cloud service where many application instances are sharing underlying hardware. We believe that the problem will be increasingly important as the number of cores per chip increases.

While cache interference in multi-core processors has been heavily studied, managed languages enable a new approach to handle cache interference, which is not possible in traditional multi-programmed workloads. In particular, automatic memory management provides parameters that allow us to actively change an application's memory access behavior and more effectively manage shared caches instead of considering the application's memory access patterns as fixed. DNA adjusts the nursery size in a cache-aware fashion to intelligently balance cache interference and garbage collection overhead. To reduce cache interference and reduce an application's memory footprint, a smaller nursery size can be used. However, smaller nursery sizes lead to more frequent

garbage collection (GC) and increased GC overhead. As a result, a good nursery size of an application depends on both the application’s own characteristics and other applications that share the cache.

DNA is implemented and evaluated on an Intel i7-based platform running a broad range of Python benchmarks. The experimental results show that today’s static nursery sizing heuristic, which does not consider cache interference and contention, leads to significantly lower performance compared to the case with cache-aware nursery sizes. The performance gap becomes even more significant as we increase the number of concurrent applications. The results also show that DNA is quite effective in determining good nursery sizes and can significantly increase performance over today’s baseline. When four programs run concurrently, the proposed scheme improves the system throughput by 16.3% on average and up to 73%. The performance improvements for individual programs can be as high as 3.1x. On average, this throughput is within 9.3% of the throughput achievable using the best combination of nursery sizes for each workload determined through offline profiling.

The following summarizes the main contributions:

- To the best of our knowledge, this paper represents the first to identify the interaction between cache sharing and managed language performance in the case of multiple concurrent programs. The paper also shows that changing memory access behavior by adjusting parameters in automatic memory management can be an effective tool for reducing cache contention.
- The paper develops an analytical model to better understand the interactions between the execution time of a managed language program and the cache and nursery sizes.
- The paper presents *Dynamic Nursery Allocator (DNA)*, which can automatically adjust the nursery size at run-time without offline profiling.
- The paper implements and evaluates the proposed scheme on a real-world system and demonstrates significant performance benefits.

The rest of the paper is organized as follows. Section 2 provides background materials on automatic memory management, and discusses the trade-off between garbage collection and cache performance for a single program. Section 3 introduces the cache-sharing problem that we identify as a new challenge. Section 4 presents an analytical model that captures the main interactions between an effective cache size and the best nursery size and describes DNA based on the intuitions from the model. Section 5 evaluates the effectiveness of our approach. Section 6 discusses related work, and Section 7 concludes the paper.

2 Automatic Memory Management

In managed languages, the language runtime allocates objects on demand as needed by the program and uses periodic

garbage collection to free dead objects. This section describes how garbage collection is typically implemented in modern languages and discusses how the garbage collection interacts with caches in modern microprocessors.

2.1 Generational Garbage Collection

Garbage collection can be performed based on reference counts or tracing of live objects. Trace-based garbage collection is used in most modern programming language implementations due to the high computational overhead and other limitations (e.g. difficulty in dealing with cycles) of reference counting. The tracing garbage collection works by following a root set of pointers to objects and from pointers in those objects to other objects. All objects that can be reached through tracing are considered live objects while those that have no pointers to them are considered dead objects.

As the number of allocated objects increases, tracing all objects in the full memory space becomes expensive. Generational garbage collection [22] is an optimized form of tracing garbage collection where the memory space is broken down into segments, with objects in a particular segment being of a particular “age”. Garbage collection is performed on the different segments at different intervals. It is used in high performance implementations of modern programming languages, such as Java [24], Python [26], and C# [38].

Generational garbage collection in its most basic form segments the memory into a space for young objects, sometimes called *nursery*, and another space for old objects called *old space*. The nursery is small compared to the old space so that *minor* garbage collection happens frequently in the nursery, while *major* garbage collection happens infrequently in the old space in order to amortize the larger overhead. Most objects die during the minor garbage collection cycle and only a small percentage of objects in the nursery are moved to the old space if they survive one or more minor garbage collection cycles.

2.2 Trade-off between Garbage Collection and Cache Performance

The interval at which minor garbage collection is run is directly related to the nursery size. Objects are allocated in the nursery in a sequential fashion. Once the nursery fills up, the minor garbage collection will run. In order to amortize the overhead of garbage collection, it can be run at infrequent intervals by increasing the nursery size. With less frequent garbage collection, the overall garbage collection overhead becomes lower and more objects die in the nursery. Unfortunately, simply increasing the nursery size can hurt the cache performance and can negatively impact the overall execution time.

To study this further, we run PyPy [6], a popular Python runtime, on ZSim [30], an x86 simulator, with the parameters shown in Table 1. The native hardware that is used in the

Figure 1. LLC miss rate as a function of nursery size.

Figure 2. Execution time breakdown as a function of nursery size.

evaluation section does not support cache partitioning, so the simulator is used to experiment with changes in LLC size. In addition, all results in Sections 2 and 3, even those that do not require changing LLC sizes, use the simulator to make them consistent and comparable.

Figure 1 shows the last-level cache (LLC) miss-rate averaged across 50 benchmarks (see Table 4) running individually without interference for a processor with an 8MB LLC. When the nursery size exceeds the cache size of 8MB, the miss-rate sharply increases because initializations to newly-allocated objects will result in off-chip memory accesses to bring the corresponding cache lines into the on-chip cache [17, 31]. On the other hand, when the nursery does fit into the cache, nursery accesses to new objects after a garbage collection cycle reuses memory locations already in the cache.

Figure 2 shows the corresponding execution times for ZSim with an 8MB LLC normalized to a 4MB nursery (i.e. nursery is half the cache size) and averaged across the 50 benchmarks. As the nursery size increases, the garbage collection (GC) overhead decreases. However, the execution time of the program increases due to the poor cache performance. On the other hand, as nursery size decreases, the garbage collection overhead increases and the execution time of the program also increases.

Figure 3 shows how the trade-off varies for different benchmarks. In some cases, such as for *float*, the reduction in garbage collection overhead for larger nursery sizes is greater than the additional overhead due to poor cache performance. In other cases, such as for *xml_etree_parse*, the additional overhead due to poor cache performance is greater than the reduction in overhead from less frequent garbage collection,

Figure 3. Execution time for individual benchmarks as a function of nursery size.

Table 1. ZSim configuration.

Core	Simple, 3.4GHz
	Single in-order unpipelined
L1I	32KB, 8-way, 1-cycle
L1D	32KB, 8-way, 1-cycle
L2	256KB, 8-way, 12-cycle
LLC	0.5-8MB, 16-way, 42-cycle
Memory	Simple, 173-cycle

and a smaller nursery size leads to better performance. These results suggest that sizing the nursery based on application characteristics as well as the cache size has a potential to yield better performance than today's static sizing scheme that sets the nursery size to be the half of the last-level cache (LLC) size. The results also show that a large nursery size such as 32MB can lead to significant performance overhead for some programs, which explains why today's heuristic sets the nursery size to be smaller than the LLC size.

For the remainder of the paper, we use 32MB as the maximum nursery size for all benchmarks as it is large enough to make the GC overhead negligible and is much larger than the LLC size that we evaluate. We assume that this is a platform-specific value set considering the cache size and memory capacity. While the maximum nursery size can be set for each application individually, the performance difference will be small. We use the same value for simplicity.

3 Cache-Sharing Effect

The previous studies on the interactions between the on-chip cache size and the nursery size primarily focused on the trade-off for a single program with a fixed cache size. In this section, we show that the trade-off also depends significantly on the interference among multiple concurrent programs when the cache is shared by multiple cores. The experiments in this section use also use ZSim (see Table 1) to study the impact of cache sharing for varying LLC sizes.

3.1 Impact on Optimal Nursery Size

When multiple applications are running on the same multi-core processor, the LLC will be shared among them. As a

Figure 4. The best nursery size for individual benchmarks under different cache sizes.

result, the effective cache size for each program will be less than the total physical cache size. As an application’s miss-rate often increases significantly when its nursery does not fit into the cache, a program needs to use a smaller nursery to achieve good cache performance when its effective cache size is reduced due to sharing. At the same time, the garbage collection overhead at the smaller nursery size will be higher. If the performance improvement from the reduced cache pressure is higher than the increased garbage collection overhead, the smaller nursery size will lead to better overall performance. On the other hand, if the garbage collection overhead dominates, then a large nursery will be better; the large nursery decreases the garbage collection overhead at the expense of more cache misses.

Figure 4 shows how the optimal nursery size for different benchmarks changes as the LLC size is varied. The benchmarks are running in isolation without interference, but the change in LLC size serves as a proxy for modeling the changing effective cache size. The optimal nursery size is determined empirically by sweeping nursery sizes 0.5MB, 1MB–8MB at 1MB increments, 32MB and choosing the size that results in the best performance. For all benchmarks except for *float*, the optimal nursery size for an 8MB cache is between 2MB and 6MB. As the cache size decreases, the optimal nursery size decreases until the cache size reaches a certain point. At that point, the garbage collection overhead becomes greater than the cache performance overhead, and the maximum nursery size (e.g. 32MB) becomes the best choice. This threshold point, which we call the *saturation threshold*, depends on the program characteristics and is specific to each program. Some benchmarks such as *rietveld* and *krakatau* reach this threshold when the cache size is large, while other benchmarks such as *xml_etree_** reach this threshold when the cache size is small. This variation can be explained by looking at Figure 3. Since *xml_etree_** have a large performance penalty when the nursery size exceeds the cache size, it is important for the nursery to fit into the LLC for these benchmarks. Their optimal nursery size decreases as LLC size decreases. In that sense, programs with a smaller threshold will be affected more by cache sharing.

Figure 5. The performance penalty of fixed nursery sizing over the optimal nursery sizing at varying cache sizes.

Figure 5 shows the problem of using a fixed nursery size without considering cache sharing, which reduces the effective cache size. The graph shows the performance penalty when the cache size is reduced to less than 8MB. The results show the averages over 50 Python benchmarks. The first scheme sets the nursery to be half the original cache size (4MB nursery for 8MB cache). The second scheme sets the nursery to be the maximum nursery size (32MB). The third scheme sets the nursery size to be half the actual cache size ($\text{Cache Size} / 2$). The final scheme sets the nursery size to be the optimal size when the application has 8MB cache to itself (Best @ 8MB LLC).

All these schemes have significantly lower performance compared to the optimal nursery sizing. In other words, the nursery sizing schemes that do not consider cache sharing lead to significant performance penalty. Even in the final scheme, the optimal nursery size when the application has 8MB cache to itself becomes sub-optimal for smaller cache sizes. The average performance penalty reaches close to 25% when the optimal nursery size for an 8MB cache is used for a 2MB cache. For sizes less than 2MB, the average penalty decreases because more applications reach the saturation threshold and the maximum nursery size becomes optimal. Also, note that the penalty for individual programs can be much higher than the average shown in the figure. The results suggest that choosing the optimal nursery size at a single cache size is not sufficient to achieve good performance when a cache is shared among multiple programs. There is an opportunity to significantly improve overall performance if we can carefully choose the nursery size for each program considering the effective cache capacity that is smaller than the physical cache size due to sharing.

4 Cache-Aware Nursery Sizing

4.1 Overview

Our cache-aware nursery sizing scheme aims to improve performance of multiple managed-language programs running concurrently with a shared cache by automatically adjusting their nursery sizes at run-time without prior knowledge or offline profiling. There are multiple technical challenges that

need to be addressed in order to achieve this goal. In order to determine a good nursery size for a shared cache, we need a better understanding of how the optimal nursery size changes when the effective cache size changes due to sharing. We also need run-time techniques to obtain individual program characteristics and measure cache contention among multiple programs without relying on offline profiling. Then, we need an efficient algorithm to determine a good nursery size for each program. Finally, the run-time operations need to be simple and low-cost in order to provide speed-ups; the overhead of monitoring programs and determining nursery sizes should not outweigh the performance improvements from better nursery size choices.

This section presents our dynamic nursery sizing scheme, named DNA, and discusses how we address the above technical challenges. We first introduce an analytical model that provides us an intuition for how to adjust the nursery size as the effective cache size changes. The model shows that the nursery size for each program can be chosen from two trade-off points, either the largest nursery size where the program working set still fits into the shared cache or the maximum nursery size. We propose a low-cost cache probing scheme to efficiently determine the nursery size that fits into the cache considering cache contention and program characteristics at run-time, and we introduce performance estimation methods to choose good nursery sizes.

4.2 Analytical Model

Cache sharing impacts the nursery sizes primarily by changing the effective cache size that each program gets at run-time. In order to understand how to determine a good nursery size to use, we consider the impact of changing the cache size on an execution time profile of a program, which shows the execution time at various nursery sizes. We start with a profile $T(S_N, S_C)$ at a single cache size S_C and various nursery sizes S_N :

$$T(S_N, S_C) = T_{GC}(S_N, S_C) + T_{NGC}(S_N, S_C) \quad (1)$$

where $T_{GC}(S_N, S_C)$ is the time of garbage collection and $T_{NGC}(S_N, S_C)$ is the execution time of an application (i.e. non-GC time). Our goal is to predict the profile $T(S_N, S'_C)$ for a different cache size S'_C . We do this by considering the two terms of the equation separately.

4.2.1 GC Execution Time

The nursery size changes the number of dynamic instructions that are executed for garbage collection and directly affects the GC execution time. It also determines how often GC runs and how many objects survive after each GC period. For example, with a smaller nursery, garbage collection is run more often and more objects survive than with a larger nursery. In the case of GC execution time, caching only has a second-order effect. Therefore, to simplify our model, we

Figure 6. Execution time of GC normalized to GC with 8MB cache and 4MB nursery. The x-axis shows the nursery sizes and the lines represent different cache (LLC) sizes.

make the following approximation:

$$T_{GC}(S_N, S'_C) \approx T_{GC}(S_N, S_C) \quad (2)$$

The effect of the nursery fitting and not fitting in the cache during garbage collection can be seen from the shift in the curves in Figure 6. Although, the execution time curves for garbage collection at different cache sizes diverge for small nursery and cache sizes, we found that we could still choose good nursery sizes with this approximation.

4.2.2 Non-GC Execution Time

Unlike the GC case, the nursery size does not affect the number of instructions for the non-GC execution. It primarily affects the non-GC cycle count through its interaction with the caches. We break this interaction into two components. First, there is the effect of cache contention among accesses within the nursery itself. If the nursery is too large to fit in the cache, bringing new nursery blocks into the cache may evict other nursery blocks. Second, there is the effect of cache contention between the non-nursery working set and the nursery.

In the case where the program only uses the nursery, the execution time will be small and there will be no cache miss as long as the nursery fits in the cache. Once the nursery size reaches the cache size, the cache miss-rate will increase until it saturates. While the saturation point varies based on each program's access patterns, we can bound it analytically. Consider a cache with the following parameters: size (capacity) S_C , block size b , N_s sets, and N_w ways. In the worst case, the saturation point will be when the nursery size (S_N) is equal to $S_C + S_C/N_w$ which represents the nursery being larger by one cache block for every cache way. As shown in Figure 7(a), this case happens when the nursery is sequentially accessed with no reuse of earlier parts of the nursery. The LRU policy implies that every access will miss. In the best case, the saturation point will be when $S_N = 2 * S_C$. As shown in Figure 7(b) and Figure 7(c), this case happens if the nursery access pattern has temporal reuses that match the LRU policy. For example, the repeated nursery access

Figure 7. Graphical depictions of a stream of cache accesses on a single cache set.

Figure 8. Miss-rate curves for two benchmarks. The breakdown shows miss rates if the accesses were performed in isolation.

pattern of [1, 2, 3, 4, 5, 4, 3, 2, 1], where the nursery is accessed again in a reverse order before a garbage collection, results in 6 hits out of 9 accesses as shown in Figure 7(b). In essence, reuse of earlier nursery blocks can have cache hits even if the nursery is larger than the cache. In our model, we conservatively assume that the saturation point occurs at $2 * S_C$.

Figure 8 shows the results of a cache simulation of two different benchmarks with an 8MB cache. The overall miss-rate curve includes accesses to both the nursery and the non-nursery. The curves of nursery-only and non-nursery only show the miss rates when those accesses are done in isolation on an 8MB cache. Figure 8(a) shows close to the worst case access pattern for nursery accesses. Figure 8(b) shows close to the best case access pattern for nursery accesses. For both benchmarks, the nursery-only miss-rates are relevant only when the nursery exceeds the cache size.

Now, let us consider the effect of non-nursery accesses on the cache performance. Although the non-nursery cache accesses generally have low miss-rates, they interfere with nursery accesses. As shown in Figure 8, there is little effect of the nursery size on the miss-rate of non-nursery accesses if performed in isolation. However, the overall miss-rate curve shows a significant increase in cache misses before the nursery size reaches the cache size.

We define the *effective cache size for nursery* as the nursery size where the miss rate starts to increase and can compute it as $S_{N,W} = S_C - S_W$ where S_W is the non-nursery working set size of the program. If the non-nursery working set is small, then the nursery size can be larger before it begins to interfere. If the non-nursery working set is large, the miss-rate begins to increase at a smaller nursery size. For

example, S_W is 0.2MB and $S_{N,W}$ is 7.8MB for Figure 8(a). S_W is 2MB and $S_{N,W}$ is 6MB for Figure 8(b). In our model, we assume that the non-nursery working set size is roughly fixed regardless of cache and nursery sizes, and predict the non-GC execution-time curve for a different cache size by shifting a known curve while keeping S_W as a constant. If the cache size were 4MB for example, then $S_{N,W}$ would be 3.8MB for Figure 8(a) and 2MB for Figure 8(b).

We can combine the effect of both components to compute T_{NGC} at a different cache size S'_C as follows:

$$T'_{NGC}(S_N, S'_C) = \begin{cases} T_{NGC}(S_C - (S'_C - S_N), S_C), & S_N \leq S'_C \\ T_{NGC}(S_N * S_C / S'_C, S_C), & S_N > S'_C \end{cases} \quad (3)$$

where the transformation for $S_N \leq S'_C$ represents a shift of the profile to keep a constant S_W and the transformation for $S_N > S'_C$ represents a scale of the profile between S_C and $2 * S_C$.

4.3 Dynamic Nursery Allocator (DNA)

In this subsection, we describe how DNA can adjust the nursery size during run-time based on program phase behavior and cache contention from other programs. This approach requires no knowledge of the programs that are running and no input from the programmer. We use cache miss-rate and execution time measurements and insights from our analytical model to predict and set the nursery size at run-time. While the insights and observations from the analytical model are used, the DNA algorithm does not directly evaluate the analytical model. As a result, DNA can be run with low overhead.

Figure 9. Overview of parameters estimated by DNA.

In Equation 1, we expect T_{GC} to be a monotonically decreasing function that has its minimal value at the maximum nursery size, which we call $S_{N,MAX}$. On the other hand, T_{NGC} is minimal at any point where $S_N < S_{N,W}$. Therefore, the overall equation T has two local minima: one near $S_{N,W}$ (the *effective cache size for nursery*) where garbage collection overhead is moderate and cache performance overhead is low and one at $S_{N,MAX}$ where garbage collection overhead is low and cache performance overhead is high. DNA first estimates the value of $S_{N,W}$. It then tries to predict the trade-off between running at $S_{N,W}$ and $S_{N,MAX}$ by evaluating whether $T_{GC}(S_{N,MAX}, S_C) + T_{NGC}(S_{N,MAX}, S_C) > T_{GC}(S_{N,W}, S_C) + T_{NGC}(S_{N,W}, S_C)$ or, equivalently, $\Delta T_{GC} + \Delta T_{NGC} > 0$. The parameters are depicted in Figure 9.

4.3.1 Cache Probing: Estimating $S_{N,W}$

In order to compare the two local minimum points, we first need to determine the *effective cache size for nursery*. This point can change at run-time based on program characteristics and other programs that contend for the cache. To determine this size, we note that objects are sequentially allocated in the nursery and temporal reuses over a long time period are rare (i.e. most objects die young). Therefore, we can consider the nursery as a large array that is being sequentially accessed. Due to the LRU cache replacement policy, the beginning of the nursery is most likely to be evicted before the later parts.

To find $S_{N,W}$, we need to find the nursery size at which the beginning of the nursery gets evicted by other memory accesses. We use cache probing at fixed nursery allocation intervals (such as after every allocation of 256KB) to access a unique set of memory addresses at the beginning of the nursery. Accessing unique memory addresses eliminates the possibility of measuring hits due to a corrupted LRU order in the cache from a previous probe. When a small portion of the nursery that fits into the cache has been allocated, the cache probing will lead to small access times, indicating that the beginning of the nursery is still in the cache. As we allocate more of the nursery, there will be a point when the access times increase indicating that the nursery is no longer fitting in the cache. We use statistical outlier detection and maintain a history of previous probe values to determine whether the access time increase is significant. If it is, then we set $S_{N,W}$ to be the allocation size at which this happens.

4.3.2 Estimating ΔT_{GC}

In order to estimate ΔT_{GC} , we need to estimate T_{GC} at the nursery sizes $S_{N,W}$ and $S_{N,MAX}$. Since GC time is amortized by running garbage collection less frequently, we can use the observed T_{GC} from a previous garbage collection at a certain nursery size S_N to make a first order estimate of GC at a different point S'_N :

$$T_{GC}(S'_N, S_C) \approx T_{GC}(S_N, S_C) * S_N / S'_N \quad (4)$$

At program initialization, when there is no historical value, we approximate the first GC time as being a fixed percentage of the measured non-GC time. We use 25% based on empirical studies.

The time of each garbage collection can vary over the course of program execution due to the types of data structures used by the program and the object survival rates. As a result, there is an accuracy trade-off between using T_{GC} from a *recent* garbage collection for a significantly different nursery size ($S_{N,RCNT}$) and T_{GC} from a less-recent garbage collection for the nursery size that is *close* to $S_{N,W}$ or $S_{N,MAX}$ ($S_{N,CLSE}$); using $S_{N,RCNT}$ reflects the current program phase, while $S_{N,CLSE}$ requires less extrapolation. We found that Equation 4 tends to over-estimate the GC time at smaller nursery sizes, and using $\min(T_{GC}(S_{N,RCNT}), T_{GC}(S_{N,CLSE}))$ works well for the purpose of choosing good nursery sizes.

4.3.3 Estimating ΔT_{NGC}

ΔT_{NGC} is the difference in non-GC execution time when using the smaller nursery size of $S_{N,W}$ that fits in the cache and the larger nursery size of $S_{N,MAX}$ that does not. The difference primarily comes from the cache miss penalty (in cycles) of missing in the nursery for object initializations. Since the maximum nursery size is already known, we can estimate this difference if we have the cache miss penalty. Initially, when we do not have any measurements of program execution time, we start with a rough estimate of the additional cycle penalty per byte allocated by measuring time to access a small (512KB) array twice. The first access to the array results in all misses, while the second access to the array results in all hits.

During the program execution, we can improve the cache penalty estimates using the actual non-GC execution time (T_{NGC}) at $S_{N,W}$ and $S_{N,MAX}$, and take the difference. However, the program phase may change between the two garbage collection periods, and comparing the non-GC time between two different program phases may not provide accurate enough estimates. Also, trying both nursery sizes frequently can incur non-trivial overhead of using sub-optimal nursery sizes. To improve accuracy with low overhead, we exploit the observation that, when we switch the nursery size from $S_{N,W}$ to $S_{N,MAX}$, the nursery allocations until we reach $S_{N,W}$ still hit in the cache after garbage collection. After the allocation passes $S_{N,W}$, nursery allocations start to miss in

Figure 10. Control flow diagram of the DNA algorithm. The algorithm is called after each chunk (256KB) of the nursery is allocated with objects.

the cache. Therefore, we can do a partial measurement of T_{NGC} for $S_N < S_{N,W}$ (cache hits) and another measurement for $S_N > S_{N,W}$ (cache misses). We scale the results to represent the same number of bytes allocated and subtract the times to get the difference. To reduce noise, we maintain a running average of our cache performance penalty from multiple measurements.

4.3.4 Implementation Details

DNA is implemented as a nursery allocator that allocates at the chunk granularity. We use 256KB for the chunks to balance between too many allocator calls and fine-grain decision-making. The allocator is called after every 256KB of a nursery is initialized with objects. Based on the estimates of $S_{N,W}$, ΔT_{NGC} , and ΔT_{GC} , the allocator can decide whether to allocate another 256KB chunk for the program to continue execution or perform garbage collection (GC). By choosing to perform GC instead of allocating another chunk, the allocator can control the nursery size dynamically at run-time.

Figure 10 depicts the control flow diagram of the core DNA algorithm. The allocator starts by recording the non-GC time that is measured between allocator calls. If less

Table 2. Microarchitectural details of native machine.

Core	i7-6700, 3.4GHz
	4 physical, 8 virtual
L1I	32KB, 8-way
L1D	32KB, 8-way
L2	256KB, 8-way
LLC	8MB, 16-way
Memory	16GB, DDR3-1600

Table 3. Nursery sizing schemes used in the evaluation

Scheme	Description
Base	Set the nursery size (S_N) to 4MB.
DNA	Use our Dynamic Nursery Allocator.
Best	Use the best static nursery size combinations from experiments.

than the maximum nursery size has been allocated, then it will perform $S_{N,W}$ estimation to update $S_{N,W}$ using cache probing. When the nursery allocation reaches $S_{N,W}$, the expression $\Delta T_{GC} + \Delta T_{NGC}$ is evaluated to determine whether $S_{N,W}$ or $S_{N,MAX}$ should be used as the nursery size. In order to keep the estimates up-to-date, the algorithm forces the nursery size to switch if the same nursery size is used for 15 consecutive GC cycles. When the decision is made to perform garbage collection, ΔT_{NGC} is updated based on the recorded non-GC time. After the garbage collection, the GC time is recorded so that it can be later used to estimate ΔT_{GC} . Once GC is run, the algorithm restarts allocation from an empty nursery to determine the next nursery size.

5 Evaluation

5.1 Experimental Setup

Our experiments are run on an Intel i7-based system as shown in Table 2. We run our experiments using PyPy [6], which is a high-performance implementation of Python with generational garbage collection and just-in-time compilation. While we do not experiment with other languages, we believe that the proposed technique is general and can be applied to other managed languages with generational garbage collection where the same trade-off between garbage collection and cache performance exists.

For the baseline, we use PyPy’s default static nursery sizing where the nursery size is set to be one half of the machine’s last-level cache size (e.g. 4MB in our case). In order to compare to the performance of ideal static nursery sizes, we try all possible combinations of nursery sizes for each group of benchmarks. Since the nursery size is a continuous variable in PyPy (in some runtimes, it has to be a power of 2), we choose 5 discrete nursery size points to enumerate possible nursery-size combinations: 1MB, 2MB, 4MB, 8MB, and 32MB. For groups of 2 benchmarks, we have 25 combinations. For groups of 3 benchmarks, we have 125 combinations. For groups of 4 benchmarks, we have 625 combinations. In our

Table 4. List of benchmarks and acronyms.

chameleon	CHM	pyflate	PYF
chaos	CHS	pyxl_bench	PYX
crypto_pyaes	CRY	raytrace	RYT
deltablue	DLT	regex_compile	RGC
dulwich_log	DLW	regex_dna	RGD
eparse	EPR	regex_effbot	RGF
fannkuch	FNN	regex_v8	RGV
float	FLT	richards	RCH
genshi_text	GNT	rietveld	RTV
genshi_xml	GNX	scimark_fft	SCF
go	GO	scimark_monte	SCM
hexiom	HXM	spectral_norm	SPN
html5lib	HTM	spitfire	SPT
json_dumps	JSD	spitfire_cstringio	SPC
json_loads	JSL	sym_expand	SYX
krakatau	KRK	sym_integrate	SYN
mako	MAK	sym_str	SYS
mdp	MDP	sym_sum	SYM
meteor_contest	MTR	telco	TLC
nbody	NBD	unpickle	UNP
nqueens	NQN	unpickle_list	UNL
pickle	PCK	xml_etree_generate	XMG
pickle_dict	PCD	xml_etree_iterparse	XMT
pickle_list	PCL	xml_etree_parse	XMP
pidigits	PDG	xml_etree_process	XMR

dynamic nursery sizing scheme, we can choose any nursery size at 256KB granularity. The schemes evaluated are summarized in Table 3.

For benchmarks, we use a wide array of applications to get a representative sample of real-world applications. We use a combination of benchmarks from the official Python benchmark suite [27] and benchmarks from the PyPy benchmark suite. In total, we evaluate our designs with 50 benchmarks. The benchmarks are listed in Table 4 along with the acronyms used in the figures.

To study cache contention among applications, we need to choose groups of applications that will run concurrently. Instead of manually selecting applications based on known characteristics, we randomly select groups of applications. In some cases multiple instances of the same application can also run together. We study running 2, 3, and 4 applications concurrently. For the groups of 2 or 3 benchmarks, the figures show the results for all groups that we used in the experiments. For the groups of 4 benchmarks, our averages include all 68 groups but we only show 34 in our graphs for readability. They are selected by sampling every other group after sorting based on performance improvement. More groups are evaluated for the 4-application case as it is a more interesting case for cache contention. All results are shown (i.e. no cherry picking) and no group is repeated.

We run each benchmark group until the longest running benchmark completes at least one execution. We repeatedly run shorter running applications so they continue to interfere with the longer running applications. We use the `rdtsc` instruction to obtain the total execution time and the garbage collection time of the first run for each benchmark.

Figure 11. Detailed throughput results when running two applications concurrently. **Figure 12.** Detailed throughput results when running three applications concurrently.

As a performance metric, we compute the system throughput (STP) using the following equation:

$$Throughput(STP) = \sum_i \frac{T_{i,alone,S_N=4MB}}{T_{i,shared}} \quad (5)$$

where the performance of each application running in a group is normalized to the performance when running alone with the nursery size of 4MB.

We use GNU Parallel [35] to schedule the applications to run concurrently on the processor cores in a way that reduces the possibility of multiple applications running on the same physical core. Our processor has 4 physical cores and 8 virtual cores with hyper-threading. As our study focuses on cache contention, we do not use hyper-threading and each application gets a dedicated physical core.

5.2 Performance Results

Here, we evaluate how well our nursery sizing scheme performs when running applications in groups by looking at

Figure 13. Detailed throughput results when running four applications concurrently.

the system throughput. Figures 11, 12, and 13 show the performance results for individual application groups. For each graph, the groups are ordered by the improvement of our nursery sizing scheme over the baseline. The results show that DNA consistently outperforms the baseline nursery sizing scheme even without prior profiling. The system throughput is improved by 10.4%, 13.9%, and 16.3% on average compared to the baseline. The average improvements are within 8.1%, 7.5%, 9.3% of the best for the groups of 2, 3, and 4 applications, respectively.

DNA improves the system throughput by over 15% for 5/23 groups of 2 applications, by over 20% for 6/19 groups of 3 applications, and by over 30% for 13/68 groups of 4 applications. In addition, the maximum improvement is 107%, 36%, 73% for groups of 2, 3, and 4 applications respectively. Individual applications can have performance improvements as high as 3.1x. This happens for a workload consisting of XMR, FLT, PYF, and CHS (not shown in Figure 13 due to the limited space) with individual benchmarks improving by 1.12x, 3.1x, 1.28x, and 1.26x, respectively, and the resulting overall improvement being 1.73x (or 73%). There are a few cases where DNA can do better than even the best static nursery sizing scheme, demonstrating one of the benefits of adapting the nursery size at run-time based on the program phase behavior.

DNA is a run-time scheme without offline profiling, so it needs time to monitor program behaviors to make good predictions. DNA also assumes that the past program characteristics are good indicators of future behaviors. DNA performs slightly worse than the baseline for a small number of cases when these assumptions are not met. In the first case, some benchmarks only run for a short period and do not perform enough garbage collection cycles. For example, UNL does not perform any garbage collection, and CRY only has a small number of garbage collection cycles. In the second case,

Figure 14. Performance improvement for various benchmarks that are dynamically scheduled to run together.

DNA does not adapt quickly enough when program characteristics change rapidly. For example, FNN and XMG need to switch from a large to small nursery sizes every three or four garbage collection cycles. DNA can be improved by incorporating offline profiling to quickly predict good nursery sizes for the short-running benchmarks, or by more explicitly detecting and remembering program phases to more quickly adapt to changes.

5.3 Dynamic Scheduling

Here, we consider the case where many applications are scheduled to run together on the machine. We run 16 applications in an alphabetical order with a maximum of 4 running concurrently on the cores of the machine. The cache contention changes over time as applications starts and ends. In Figure 14, DNA is compared to the baseline scheme of setting the nursery to be 4MB. We are able to improve the execution time of the applications by an average of 8.2% and by more than 10% for 7 of the 16 benchmarks. This demonstrates the benefits of having a dynamic scheme over a static scheme.

5.4 Overhead of Dynamic Allocation Decisions

Since DNA is invoked after every 256KB of nursery allocated, it introduces additional overhead over the baseline design.

By measuring the execution time spent in the DNA code, we find that the runtime overhead is trivial. Across the 50 benchmarks, DNA overhead is on average less than 1% of the overall execution time. In all cases, it is less than 4% of the overall execution time with more than half the benchmarks having less than 0.5% overhead. The results suggest that the benefits of using DNA far outweigh its costs.

6 Related Work

To the best of our knowledge, this paper represents the first study on the impact of *cache sharing between multiple managed-language applications*. Eizenberg et al. [10] introduce Remix to automatically detecting false cache sharing within a single Java program and repairing it by padding class fields to eliminate memory accesses to the same cache line between cores. Their approach to addressing the cache sharing issue in the context of a single program is complementary to the approach presented in this paper.

Zhao et al. [43] identify a strong correlation between the object allocation rate and memory bus write traffic in Java programs. They conclude that scalability and performance are limited by the object allocation rate, resulting in an “allocation wall”. Our work confirms their initial experiments and shows that the effect is amplified when multiple programs run concurrently. Yang et al. [41] show that the impact of initialization is non-trivial. Sartor et al. [31] and Ismail and Suh [17] show that the “allocation wall” can be overcome by directly initializing cache lines without reading from memory using *cache installation*. However, cache installation will not eliminate cache thrashing between applications, and our nursery sizing technique would still be useful in such cases.

Garbage collection in the context of multiple concurrent applications has been studied, but the work focuses on improving the virtual memory performance by reducing paging. Alonso and Appel [1] discuss a holistic approach where managed runtimes adjust their working set size according to system utilization. Other work has looked at reducing paging using equation models [36], control theory [39], and forcing garbage collection to limit unnecessary memory usage [14]. In addition, Hertz et al. [13] describe a garbage collection algorithm that works with the virtual memory manager to guide page eviction decisions and reduce overall paging. This paper studies a new problem, which is cache contention among multiple programs.

Garbage collection has been explored in multi-core contexts, but most of the work focuses on single multi-threaded applications. For example, Ogasawara [25] studied the scalability problems of a Java-based web server running on a chip multi-processor. Using object pooling for long-lived classes, they can greatly improve the scalability and throughput of an application. There has been much work on implementations of garbage collection on multi-threaded applications [2, 9, 23, 28, 42, 44]. The designs follow Doligez and

Leroy’s work [9] where a small thread-private local heap is used for fast allocations and efficient garbage collection and a larger shared heap is used only when needed. Raghunathan et al. [28] extend this design to setup the heap hierarchy for further optimization in functional languages. More recent work focuses on NUMA-aware data placement in the context of automatic memory management [4, 11, 12, 44]. Gidra et al. [12] propose NumaGiC, which attempts to minimize remote accesses to memory during garbage collection through a distributed algorithm.

Previous studies have proposed to improve the locality by shaping memory accesses with garbage collection [8, 15, 16, 20]. For example, Huang et al. [16] use online profiling to determine which objects have frequently-accessed fields and use a copying garbage collector to reorder objects in a way that improves locality. The improved locality does not significantly change the cache contention among multiple programs, which is the problem that we address.

Some work considers automatically tuning garbage collection and runtime parameters [7, 19, 21, 32–34, 37]. Jayasena et al. [19] use an autotuning algorithm to tune up 600 different JVM parameters relating to garbage collection, just-in-time compilation, and class loading. Cameron et al. [7] apply economic theory to perform holistic tuning of heap sizes of multiple applications. Our work is the first to consider nursery size in the context multiple applications and show that we can tune nursery size for significant performance improvement.

Previous work discusses how to set the best nursery size for individual applications [5, 18, 29, 40]. This work further extends the nursery sizing for multiple concurrent applications, and shows that proper nursery sizing is essential for good performance when multiple applications share a cache.

Cache partitioning is a well known technique for handling interference among applications. Our work is complementary to the cache partitioning and future work could consider partitioning along with nursery sizing. The cache partition sizes can be used as an input to our scheme to choose good nursery sizes for a given partitioning configuration.

7 Conclusion

In this paper, we identify cache contention as an important factor in determining good nursery sizes for applications running in managed languages. To address the problem, we develop a model to understand the impact of cache size on nursery-sizing, and propose a new dynamic nursery sizing scheme, named DNA, that can find good nursery sizes with low overhead. Experimental results show that DNA can significantly improve the system throughput and that changing memory access behavior can be an effective tool in reducing cache contention and improving performance of concurrent programs.

A Artifact Appendix

A.1 Abstract

Our artifact provides a full framework to evaluate the performance of our *Dynamic Nursery Allocator*. The framework is provided as a Docker image to ensure repeatability and to make setup and running it easy on any platform.

Our framework consists of:

1. The compiled PyPy binary that supports DNA along with the PyPy source code and the DNA source code.
2. The Python and PyPy benchmark suite programs used in the evaluation.
3. Scripts to setup the experiments and automate running the benchmarks.
4. An analysis script to summarize the results as CSV files.
5. Graphing scripts to generate graphs from the CSV files.

We additionally provide reference CSV files with the results presented in the paper for inspection and comparison.

A.2 Artifact Check-list (Meta-information)

- **Algorithm:** Dynamic Nursery Allocator, an online dynamic scheme that automatically adjusts the nursery sizes of multiple managed-language programs running concurrently.
- **Program:** A modified version of PyPy that supports DNA and benchmarks from the Python and PyPy benchmark suites are included.
- **Compilation:** Supports GCC 4.9+.
- **Binary:** A modified version of PyPy is pre-compiled.
- **Run-time environment:** An Ubuntu-based docker container has been preconfigured with all necessary software dependencies.
- **Hardware:** Any x86 machine should work.
- **Run-time state:** For best evaluation results, limit background processes while experiments are running.
- **Metrics:** Per-application throughput as well as total throughput for each benchmark group.
- **Output:** A CSV file containing throughput numbers that can be further analyzed in Excel or using a scripting language.
- **Experiments:** Scripts are provided to configure the experiments and run the benchmarks.
- **How much disk space required (approximately)?:** At least 2GB to support the Docker container.
- **How much time is needed to prepare workflow (approximately)?:** Once Docker is setup, a few minutes to import the image.
- **How much time is needed to complete experiments (approximately)?:** Tens of minutes to a few hours depending on how many experiments are run.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** Both PyPy and the Python Benchmark suite are distributed under the MIT

license. The full container is licensed under the GNU GPL license.

A.3 Description

A.3.1 How Delivered

The framework is provided as a Docker image and is available for download on ACM DL.

A.3.2 Hardware Dependencies

Our framework will work on any x86_64 machine that supports virtualization. To get comparable results with those provided in the paper, a machine with a configuration similar to Table 2 should be used.

A.3.3 Software Dependencies

All software dependencies have been pre-installed in the provided Docker image. The framework can be run on any machine with Docker installed. However, we recommend using Docker on a Linux operating system for best efficiency and most comparable results. Verify in the Docker settings that the number of CPUs available for a container is not limited and that there is sufficient memory available (i.e. at least 2GB) for the container.

A.4 Installation

After downloading the image, import it in docker and run an interactive shell.

```
host> docker import pydna.tar pydna
host> docker run -it pydna bash
docker>
```

Since the container is already setup, the system should be ready once it enters the docker bash prompt.

A.5 Experiment Workflow

Once in the docker bash prompt, navigate to the runs directory and start the run.

```
docker> cd python_optimize/benchmarks/pnative/runs
docker> ../local_run
```

The runs directory is setup to run the same benchmark groups as in the paper. The customization section explains how to modify the benchmark groups.

For best results, run the experiments with minimal CPU utilization.

A.6 Evaluation and Expected Result

After the experiments finish running, each of the leaf subdirectories has an `accumulate_stats.out` file which has raw statistics collected from the run including program execution time. Run the `compute_throughputs.pl` script in the results folder to compare the throughput between runs.

```
docker> cd ../results
docker> ./compute_throughputs.pl -n 2
```



```
docker> ./compute_throughputs.pl -n 3
docker> ./compute_throughputs.pl -n 4
```

These commands will generate `throughput_*.csv` which give the total throughput for each benchmark group when running PyPy with default parameters (i.e. base) and when running it with DNA. An optional `-d` flag can be used to generate detailed results which have the throughput broken down per application in the benchmark group.

Run the graphing scripts in the `results` folder to generate graphs from the CSV files.

```
docker> ./throughput_graph.py <throughput_x.csv>
docker> ./improvement_graph.py <throughput_x.csv>
```

For the above commands, replace `throughput_x.csv` with each of the CSV file names. The `throughput_graph.py` script will generate the throughput comparison graphs similar to Figures 11–13 in the paper. The `improvement_graph.py` script will graph the improvements of the benchmark groups instead of raw throughput numbers. It will also print statistics to the terminal. If it is called with a detailed CSV file (which is generated by calling `compute_throughputs.pl` with a `-d` flag), then it will also print the maximum improvement of an individual application.

The CSV files can be copied using `docker cp` to the local machine and then further analyzed using Excel or a scripting language. For comparison, the numbers presented in the paper are also included in the `results` folder as `*paper.csv`.

A.7 Experiment Customization

The framework is highly customizable. The following are some ways in which the framework can be customized (file-names given relative to the `pnative` folder):

1. `results/compute_throughputs.pl` can use a different run directory for analysis and can use a different output file name. Use the `-h` option to view the help.
2. `create_groups.pl` can randomly generate new benchmark groups. In the file, modify the `group_quantities` variable, which maps number of benchmarks per group to the number of unique groups to generate. When the script runs, it overwrites the `bmgroup.txt` file with the new benchmark groups. For comparison, `bmgroup_paper.txt` includes all the benchmark groups evaluated in the paper.
3. `setup.pl` creates a new run directory based on the benchmark groups in `bmgroup.txt`. It is generally executed after running `create_groups.pl`. It contains variables that determine how many times to run the same benchmark group (`max_runs`) and the command line arguments for running the different PyPy configurations (`py_configs`).
4. `./run_int.sh` contains the commands used to run the benchmarks. New benchmarks can be added by adding the appropriate environment variables and commands to the file. Benchmarks need to be able to run in a looped fashion encapsulated by a function with the name `my_main`. The

PyPy code uses the `my_main` function to determine the start and end of one full run of the benchmark.

A.8 Notes

The commented DNA source code can be found at the following path:

```
/python_optimize/pypy-5.3.1-src/
translated-c-x64-trace3/testing_1/sharedcache.{c,h}
```

If any of the source files are changed, then PyPy can be recompiled by running `make` in the `testing_1` directory.

A.9 Methodology

Information on submission, reviewing, and badging methodology can be found at the following sites:

- <http://cTuning.org/ae/submission-20190109.html>
- <http://cTuning.org/ae/reviewing-20190109.html>
- <https://www.acm.org/publications/policies/artifact-review-badging>

Acknowledgments

This work was partially supported by the Office of Naval Research grant N00014-15-1-2175. We would also like to thank the anonymous reviewers for their constructive comments and suggestions.

References

- [1] Raphael Alonso and Andrew W. Appel. 1990. An Advisor for Flexible Working Sets. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. ACM, New York, NY, USA, 153–162. <https://doi.org/10.1145/98457.98753>
- [2] Todd A. Anderson. 2010. Optimizations in a Private Nursery-based Garbage Collector. In *Proceedings of the International Symposium on Memory Management (ISMM)*. ACM, New York, NY, USA, 21–30. <https://doi.org/10.1145/1806651.1806655>
- [3] Andrew W. Appel. 1987. Garbage Collection can be Faster than Stack Allocation. *Inform. Process. Lett.* 25, 4 (June 1987), 275–279. [https://doi.org/10.1016/0020-0190\(87\)90175-X](https://doi.org/10.1016/0020-0190(87)90175-X)
- [4] Sven Auhagen, Lars Bergstrom, Matthew Fluet, and John Reppy. 2011. Garbage Collection for Multicore NUMA Machines. In *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC)*. ACM, New York, NY, USA, 51–57. <https://doi.org/10.1145/1988915.1988929>
- [5] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. 2004. Myths and Realities: The Performance Impact of Garbage Collection. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/Performance)*. ACM, 25–36. <https://doi.org/10.1145/1005686.1005693>
- [6] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, and Armin Rigo. 2009. Tracing the Meta-level: PyPy’s Tracing JIT Compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS)*. ACM, New York, NY, USA, 18–25.
- [7] Callum Cameron, Jeremy Singer, and David Vengerov. 2015. The Judgment of Forseti: Economic Utility for Dynamic Heap Sizing of Multiple Runtimes. In *Proceedings of the International Symposium on Memory Management (ISMM)*. ACM, New York, NY, USA, 143–156. <https://doi.org/10.1145/2754169.2754180>

- [8] Trishul M Chilimbi and James R Larus. 2001. Data Structure Partitioning with Garbage Collection to Optimize Cache Utilization. US Patent 6,321,240.
- [9] Damien Doligez and Xavier Leroy. 1993. A Concurrent, Generational Garbage Collector for a Multithreaded Implementation of ML. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, New York, NY, USA, 113–123. <https://doi.org/10.1145/158511.158611>
- [10] Ariel Eizenberg, Shiliang Hu, Gilles Pokam, and Joseph Devietti. 2016. Remix: Online Detection and Repair of Cache Contention for the JVM. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, New York, NY, USA, 251–265. <https://doi.org/10.1145/2908080.2908090>
- [11] Lokesh Gidra, Gaël Thomas, Julien Sopena, and Marc Shapiro. 2013. A Study of the Scalability of Stop-the-world Garbage Collectors on Multicores. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, New York, NY, USA, 229–240. <https://doi.org/10.1145/2451116.2451142>
- [12] Lokesh Gidra, Gaël Thomas, Julien Sopena, Marc Shapiro, and Nhan Nguyen. 2015. NumaGiC: A Garbage Collector for Big Data on Big NUMA Machines. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, New York, NY, USA, 661–673. <https://doi.org/10.1145/2694344.2694361>
- [13] Matthew Hertz, Yi Feng, and Emery D. Berger. 2005. Garbage Collection without Paging. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, New York, NY, USA, 143–153. <https://doi.org/10.1145/1065010.1065028>
- [14] Matthew Hertz, Stephen Kane, Elizabeth Keudel, Tongxin Bai, Chen Ding, Xiaoming Gu, and Jonathan E. Bard. 2011. Waste Not, Want Not: Resource-based Garbage Collection in a Shared Environment. In *Proceedings of the International Symposium on Memory Management (ISMM)*. ACM, New York, NY, USA, 65–76. <https://doi.org/10.1145/1993478.1993487>
- [15] Martin Hirzel. 2007. Data Layouts for Object-oriented Programs. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. ACM, New York, NY, USA, 265–276. <https://doi.org/10.1145/1254882.1254915>
- [16] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J Eliot B. Moss, Zhenlin Wang, and Perry Cheng. 2004. The Garbage Collection Advantage: Improving Program Locality. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, New York, NY, USA, 69–80. <https://doi.org/10.1145/1028976.1028983>
- [17] Mohamed Ismail and G. Edward Suh. 2018. Hardware-software Co-optimization of Memory Management in Dynamic Languages. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management (ISMM)*. ACM, New York, NY, USA, 45–58. <https://doi.org/10.1145/3210563.3210566>
- [18] Mohamed Ismail and G Edward Suh. 2018. Quantitative Overhead Analysis for Python. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 36–47. <https://doi.org/10.1109/IISWC.2018.8573512>
- [19] S. Jayasena, M. Fernando, T. Rusira, C. Perera, and C. Philips. 2015. Auto-tuning the Java Virtual Machine. In *IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, 1261–1270. <https://doi.org/10.1109/IPDPSW.2015.84>
- [20] Thomas Kistler and Michael Franz. 2000. Automated Data-member Layout of Heap Objects to Improve Memory-Hierarchy Performance. *ACM Trans. Program. Lang. Syst.* 22, 3 (May 2000), 490–505. <https://doi.org/10.1145/353926.353937>
- [21] Philipp Lengauer and Hanspeter Mössenböck. 2014. The Taming of the Shrew: Increasing Performance by Automatic Parameter Tuning for Java Garbage Collectors. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE)*. ACM, New York, NY, USA, 111–122. <https://doi.org/10.1145/2568088.2568091>
- [22] Henry Lieberman and Carl Hewitt. 1983. A Real-time Garbage Collector Based on the Lifetimes of Objects. *Commun. ACM* 26, 6 (June 1983), 419–429.
- [23] Simon Marlow and Simon Peyton Jones. 2011. Multicore Garbage Collection with Local Heaps. In *Proceedings of the International Symposium on Memory Management (ISMM)*. ACM, New York, NY, USA, 21–32. <https://doi.org/10.1145/1993478.1993482>
- [24] Sun Microsystems. 2006. Memory Management in the Java HotSpot Virtual Machine. <http://www.oracle.com/technetwork/java/javase/tech/memorymanagement-whitepaper-1-150020.pdf>
- [25] Takeshi Ogasawara. 2010. Scalability Limitations when Running a Java Web Server on a Chip Multiprocessor. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference (SYSTOR)*. ACM, New York, NY, USA, Article 6, 10 pages. <https://doi.org/10.1145/1815695.1815703>
- [26] The PyPy Project. 2014. Garbage Collection in PyPy. https://pypy.readthedocs.io/en/release-2.4.x/garbage_collection.html
- [27] PyPerformance. 2017. Python Performance Benchmark Suite. <http://pyperformance.readthedocs.io/>
- [28] Ram Raghunathan, Stefan K. Muller, Umut A. Acar, and Guy Blelloch. 2016. Hierarchical Memory Management for Parallel Programs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, New York, NY, USA, 392–406. <https://doi.org/10.1145/2951913.2951935>
- [29] Vimal K Reddy, Richard K Sawyer, and Edward F Gehringer. 2006. A Cache-pinning Strategy for Improving Generational Garbage Collection. In *International Conference on High-Performance Computing (HiPC)*. Springer, 98–110.
- [30] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*. ACM, 475–486.
- [31] Jennifer B. Sartor, Wim Heirman, Stephen M. Blackburn, Lieven Eeckhout, and Kathryn S. McKinley. 2014. Cooperative Cache Scrubbing. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT)*. ACM, New York, NY, USA, 15–26.
- [32] Jose Simão and Luis Veiga. 2012. VM Economics for Java Cloud Computing: An Adaptive and Resource-aware Java Runtime with Quality-of-execution. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, Washington, DC, USA, 723–728. <https://doi.org/10.1109/CCGrid.2012.121>
- [33] Jeremy Singer, Richard E. Jones, Gavin Brown, and Mikel Luján. 2010. The Economics of Garbage Collection. In *Proceedings of the International Symposium on Memory Management (ISMM)*. ACM, New York, NY, USA, 103–112. <https://doi.org/10.1145/1806651.1806669>
- [34] Jeremy Singer, George Kovoor, Gavin Brown, and Mikel Luján. 2011. Garbage Collection Auto-tuning for Java Mapreduce on Multi-cores. In *Proceedings of the International Symposium on Memory Management (ISMM)*. ACM, New York, NY, USA, 109–118. <https://doi.org/10.1145/1993478.1993495>
- [35] Ole Tange. 2018. GNU Parallel 2018. <https://doi.org/10.5281/zenodo.1146014>
- [36] Y. C. Tay, Xuanran Zong, and Xi He. 2013. An Equation-based Heap Sizing Rule. *Perform. Eval.* 70, 11 (Nov. 2013), 948–964. <https://doi.org/10.1016/j.peva.2013.05.009>
- [37] David Vengerov. 2009. Modeling, Analysis and Throughput Optimization of a Generational Garbage Collector. In *Proceedings of the International Symposium on Memory Management (ISMM)*. ACM, New York, NY, USA, 1–9. <https://doi.org/10.1145/1542431.1542433>
- [38] Maira Wenzel, Alan Dawkins, Luke Latham, Tom Pratt, Mike Jones, Michal Ciechan, and Xaviex. 2017. Fundamentals of Garbage Collection. [https://msdn.microsoft.com/en-us/library/ee787088\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/ee787088(v=vs.110).aspx)

- [39] David R. White, Jeremy Singer, Jonathan M. Aitken, and Richard E. Jones. 2013. Control Theory for Principled Heap Sizing. In *Proceedings of the International Symposium on Memory Management (ISMM)*. ACM, New York, NY, USA, 27–38. <https://doi.org/10.1145/2491894.2466481>
- [40] Paul R Wilson, Michael S Lam, and Thomas G Moher. 1992. Caching Considerations for Generational Garbage Collection. In *ACM SIGPLAN Lisp Pointers*. ACM, New York, NY, USA, 32–42.
- [41] Xi Yang, Stephen M. Blackburn, Daniel Frampton, Jennifer B. Sartor, and Kathryn S. McKinley. 2011. Why Nothing Matters: The Impact of Zeroing. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. ACM, New York, NY, USA, 307–324. <https://doi.org/10.1145/2048066.2048092>
- [42] Tatu J Ylonen. 2010. Parallel Garbage Collection and Serialization Without Per-object Synchronization. US Patent App. 12/388,543.
- [43] Yi Zhao, Jin Shi, Kai Zheng, Haichuan Wang, Haibo Lin, and Ling Shao. 2009. Allocation Wall: A Limiting Factor of Java Applications on Emerging Multi-core Platforms. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. ACM, New York, NY, USA, 361–376. <https://doi.org/10.1145/1640089.1640116>
- [44] Jin Zhou and Brian Demsky. 2012. Memory Management for Many-core Processors with Software Configurable Locality Policies. In *Proceedings of the International Symposium on Memory Management (ISMM)*. ACM, New York, NY, USA, 3–14. <https://doi.org/10.1145/2258996.2259000>