HARDWARE-SOFTWARE CO-OPTIMIZATION FOR DYNAMIC LANGUAGES

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Mohamed Ibrahim Ismail

August 2019

HARDWARE-SOFTWARE CO-OPTIMIZATION FOR DYNAMIC LANGUAGES

Mohamed Ibrahim Ismail, Ph.D.

Cornell University 2019

As software becomes more complex and the costs of developing and maintaining code increase, dynamic programming languages such as Python are becoming more desirable alternatives to traditional static languages such as C. Programmers can express more functionality with fewer lines of code and can spend less time debugging low-level bugs such as buffer overflows and memory leaks. Unfortunately, programs written in a dynamic language often execute significantly slower than an equivalent program written in a static language, sometimes by orders of magnitude.

This dissertation investigates the following question: How can dynamic languages achieve high performance through HW/SW co-optimization? The first part of the dissertation studies inefficiencies in dynamic languages through a detailed quantitative analysis of the overhead in Python. The study identifies a new major source of overhead, C function calls, for the Python interpreter. Additionally, studying the interaction of the runtime with the underlying processor hardware shows that the performance of Python with JIT depends heavily on the cache hierarchy and memory system. Proper nursery sizing is necessary for each application to optimize the trade-off between cache performance and garbage collection overhead.

Based on insights from the study, the software and hardware are co-optimized to improve the memory management performance. In the second part of the dissertation, a cache-aware optimization for single-application memory management is presented. The performance and memory bandwidth usage is improved by co-optimizing garbage collection overhead and cache performance for newly-initialized and dead objects. Fur-

ther study shows that less frequent garbage collection results in a large number of cache misses for initial stores to new objects. The problem is solved by directly placing uninitialized objects into on-chip caches without off-chip memory accesses. Cache performance is further optimized by reducing unnecessary cache pollution and write-backs through a partial tracing algorithm that invalidates dead objects between full garbage collections.

The dissertation then focuses on the case of multiple applications running concurrently on a multi-core processor with shared caches. It is shown that the performance of dynamic languages can degrade significantly due to cache contention among multiple concurrent applications that share a cache. To address this problem, program memory access patterns are reshaped by adjusting the nursery size. Both a static and a dynamic scheme are presented that determine good nursery sizes for multiple programs running concurrently.

BIOGRAPHICAL SKETCH

Mohamed Ibrahim Ismail attended Cornell University starting in 2008, where he first received his Bachelor of Science degree (summa cum laude) in Electrical and Computer Engineering. He then continued his studies at Cornell University, where he pursued his Ph.D. degree in the School of Electrical and Computer Engineering. He worked with his advisor, Professor G. Edward Suh, on various topics in the field of computer architecture including runtime monitoring, real-time systems, and dynamic languages.

In the name of God, the most Compassionate the most Merciful.

ACKNOWLEDGEMENTS

Over the seven years of pursuing the Ph.D., there have been many people that have pushed me and helped me get to the point where, all praise to God, I have successfully completed my dissertation and earned the Ph.D. degree. Every small act of goodness had its part in my trajectory through graduate school and I would like to thank each and every person that helped along the way (even if I cannot mention everyone by name in these acknowledgements).

Particularly, I would like to thank my advisor, Professor G. Edward Suh, for his continuing support and feedback throughout my Ph.D. When I first started, I had vague notions of what research was and how to conduct it. Through regular feedback, I learned how to more effectively identify problems, propose approaches, analyze the potential benefits and costs, and compare to existing solutions. He gave me the freedom to choose my research direction and was ready to help reorient me when I got lost along the way. For every idea, paper, and presentation, he would also provide extensive feedback on how to improve my work and how to think more critically about the problem. Even when some ideas were not good or results were not too promising, he would try to point out the positives and encourage me to continue brainstorming and to rethink the problem.

I would like to thank my other committee members, Professor José Martínez and Professor Hakim Weatherspoon, for their input and feedback on my dissertation. José's questions would always get me thinking about different aspects of my research and evaluation that I may have not completely thought about. Hakim would always push me to think about the impact of my work in the context of the broader research in my area.

Members of the Suh research group and the Computer Systems Laboratory at Cornell provided a supportive environment where I could discuss research ideas as well as challenges I was facing along the way. I would especially like to thank Daniel Lo

and Tao Chen who I collaborated with in the early years of my Ph.D. Our interactions and work helped me get a better understanding of the process of going from an idea to a paper. Thanks to Berkin Ilbeyi for the discussions on ideas for optimizing dynamic languages. Thanks also to Professor Christopher Batten for his encouragement and help throughout my time at Cornell. Finally, I would also like to thank my other research group members Ruirui (Raymond) Huang, Wing-kei (KK) Yu, Yao Wang, Taejoon Song, Andrew Ferraiuolo, Benjamin Wu, Weizhe (Will) Hua, Mulong Luo, and Sungbo Park for their feedback on my research ideas during our group meetings and practice talks.

Outside of Cornell, I had support from many people who were interested in my progress and wanted to see me succeed. Their words of encouragement, concern over my progress, and push to keep me focused were essential to my successful completion of the Ph.D. I would like to thank Dr. Zaman Marwat for his mentorship during challenging times of my Ph.D. and his check-ins to make sure I was getting closer to graduation. Turki Baroud and Ali Al-Forqani, both contemporary Ph.D. students in different departments, helped me understand different perspectives on my experiences during the Ph.D.

Finally, I would like to thank my family members for their support. My wife, Hafiza, came to Ithaca during the last two years of my Ph.D. and became an essential pillar of support. Even though I spent large amounts of time finishing up research and writing papers, she was patient with me and continued to encourage me. My siblings Khadija, Lookmaan, Talha, Saadiya, and Aaqib have always been there when needed. Particularly, my older sister Khadija has always pushed me to do my best. My parents, Ibrahim and Fatema, have always supported me to pursue my interests without reservation. I am grateful for all their sacrifices in raising me and getting me to the point where I am today.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

As software becomes more complex and the costs of developing and maintaining code increase, dynamic programming languages are becoming more desirable alternatives to traditional static languages. Dynamic languages allow programmers to express more functionality with less code. In addition, runtime checks and memory management are built-in, limiting the possibility of low-level program bugs such as buffer overflow. Dynamic languages such as JavaScript, Python, PHP, and Ruby consistently rank in the top ten most popular languages across multiple metrics [16, 24, 103]. These dynamic languages are increasingly utilized in production environments as well in order to bring new features quickly.

Unfortunately, programs written in a dynamic language often execute significantly slower than an equivalent program written in a static language, sometimes by orders of magnitude. Therefore, the performance overhead represents a major cost of using dynamic languages in high-performance applications. Ideally, companies with enough time and resources may rewrite performance-critical portions of code in faster static languages when they are mature. For example, Twitter reported a 3x reduction in search latencies after porting their Ruby on Rails infrastructure to Java [65]. However, porting code is an expensive proposition and just-in-time (JIT) compilation is often used as a lower-cost alternative to improve the performance of dynamic language programs.

Even though dynamic languages and JIT are becoming increasingly important workloads, we currently do not have a good understanding of sources of overhead and how the underlying microarchitecture affects their performance. State-of-art high-performance superscalar processors with out-of-order speculation, increasingly complex branch prediction hardware, and multi-level cache hierarchies are still optimized

for single-threaded performance of programs written in low-level languages such as C and are not specialized to accelerate common features of dynamic languages such as automatic memory management or just-in-time compilation. Furthermore, implementing these features in dynamic languages without considering the underlying microarchitecture can result in poor utilization of processor resources and poor overall performance.

This dissertation studies inefficiencies in dynamic languages through quantitative analysis and co-optimizes the software runtime and hardware architecture to improve the overall performance based on the findings. Optimizations are targeted to be transparent to the programmer of the dynamic language and to be generally applicable to a range of dynamic languages and implementations. While there has been substantial work in improving software runtime of dynamic languages, many do not consider the role of the hardware architecture in improving the performance. Therefore, this dissertation seeks to address the following research question: *How can dynamic languages achieve high performance through HW/SW co-optimization?*

A quantitative analysis of the Python runtimes provides new insights into sources of overhead that may lead to inefficient program execution. For example, C function calls are identified as a significant source of overhead in dynamic languages. In addition, it is found that careful tuning of automatic memory management parameters is essential for good performance. Optimizations are then proposed to improve the performance of automatic memory management in dynamic languages. First, hardware support for improving the cache performance of individual programs while lowering their garbage collection overhead is discussed. Then, an optimization for efficient nursery sizing in the context of multiple applications running on a multi-core processor with shared caches is discussed. This dissertation demonstrates that careful HW/SW co-optimization can significantly improve the performance of dynamic languages.

Table 1.1: Comparison of the top 10 most popular programming languages in November 2008, 2013, and 2018. Popularity was measured as the percent of questions tagged with a particular language in Stack Overflow (shown in parenthesis). Dynamic languages are shown with emphasis. Data compiled by Global App Testing [115].

Rank	2008	2013	2018
1	C# (13.4%)	***JavaScript (10.6%)***	***Python (11.3%)***
2	Java (7.7%)	Java (10.2%)	***JavaScript (10.2%)***
3	C++ (5.6%)	***PHP (8.3%)***	Java (7.6%)
4	***JavaScript (4.8%)***	C# (8.2%)	C# (5.3%)
5	***PHP (4.1%)***	***Python (5.1%)***	***PHP (4.9%)***
6	***SQL (3.5%)***	C++ (4.3%)	C++ (2.7%)
7	***Python (3.5%)***	***SQL (3.7%)***	***R (2.6%)***
8	C (2.0%)	Objective-C (2.4%)	***SQL (2.4%)***
9	***Ruby (1.6%)***	C (2.3%)	Swift (1.9%)
10	***Perl (0.9%)***	***Ruby (1.5%)***	C (1.5%)

1.1 Dynamic Language Popularity and Performance

Over the past decade, dynamic languages have become increasingly popular and have become increasingly used in place of their static language counterparts. They are now being used in a variety of applications, including scientific computing and numerical applications, web applications, graphical user interfaces, cloud applications, scripting, and even embedded devices. These languages provide a rich set of features that programmers can use to write more complex code with fewer lines and provide more advanced debugging capabilities through interactive introspection.

Table 1.1 shows one of the many popularity rankings of programming languages published by various trade groups. Global App Testing [115] gathered data from Stack Overflow of the percent of questions tagged with a specific language for every month over a ten year period. The table shows the results for November of 2008, 2013, and 2018. In 2008, many dynamic languages were already in use. However, static languages such as C#, Java, and C++ were more widely used. By 2013, the landscape looked more

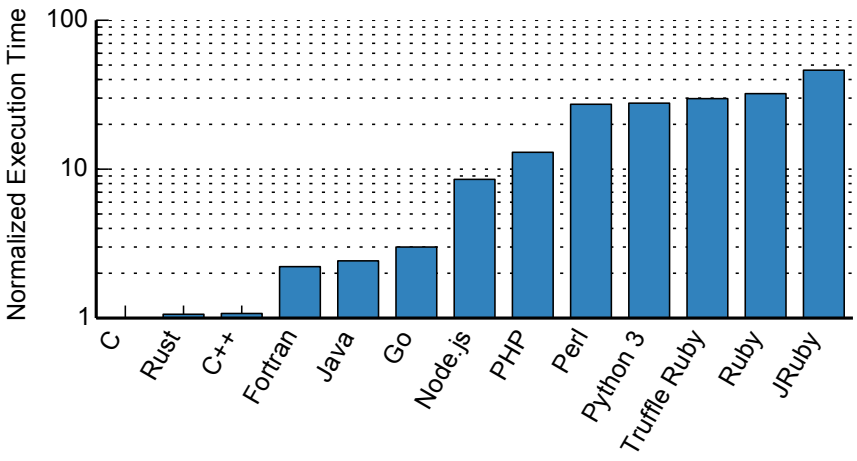


Figure 1.1: Average execution time of the same 10 benchmarks written in different languages normalized to the execution time of C. Data is taken from the Computer Language Benchmark Games [53].

even with approximately the same popularity between JavaScript and Java, PHP and C#, and Python and C++. As of 2018, Python and JavaScript have continued to see increased usage with the other languages seeing noticeable declines.

Unfortunately, the best implementations of dynamic languages as of 2019 still run an order of magnitude slower than static languages. One project, known as the Computer Language Benchmark Games, has measured the execution time of the same 10 benchmarks written for various languages [53]. Figure 1.1 shows their results. The average execution time of the benchmarks written in different languages is normalized to the average execution time of the benchmarks written in C, and the results are ordered by increasing average normalized execution time. Even with this ordering, all the languages to the left of and including `Go` are static languages and the languages to the right of `Go` starting at `Node.js` (an implementation of JavaScript) are dynamic languages. The results clearly show that there is an order of magnitude increase in normalized execution time for applications when they are written in dynamic languages.

It is also worth noting that `Truffle Ruby`, `Ruby`, and `JRuby` are all implementations

of Ruby. `Truffle Ruby` and `JRuby` use Java runtimes as their backend. The results show that merely compiling code from one language to another does not eliminate the performance inefficiencies of using dynamic languages.

The performance numbers already capture the fact that dynamic language runtimes have been improved and optimized for performance. For example, just-in-time compilers and more efficient garbage collection algorithms are already used to greatly reduce the computational cost of executing programs written in these languages. The data shows that there is more potential for improving performance of these languages.

One natural question that follows is why programmers would be willing to write in these dynamic languages if they are so slow. Some reasons are as follows:

1. Dynamic languages enable programmers to express more complex algorithms with less lines of code and provide better debugging capabilities with built-in error checking and interactive introspection. As a result, programmers can be more productive and companies can develop new features more quickly. For example, Twitter initially used Ruby on Rails for development of their service [65] and Dropbox used Python [81].
2. Some languages are part of standards with no other alternatives. For example, JavaScript is the only supported scripting language by browsers, so web applications have to be written in JavaScript.
3. Significant resources have been invested in developing libraries for some languages and rewriting code in other languages would require porting those libraries. For example, Facebook has invested resources in developing PyTorch, a Python library for machine learning [110].
4. Runtimes are constantly being improved for performance, so code that is currently

executing in a slow interpreter can potentially run faster with future advances in runtime performance. For example, just-in-time compilation has already improved performance in these languages without programmers having to rewrite code.

Therefore, dynamic languages will continue to increase in popularity as algorithms become more complex and they become more difficult to express in static languages. With enough investment and development in dynamic languages, it may be possible for the performance of dynamic languages to match the performance of some static languages in the future. This dissertation demonstrates some ways to achieve high performance in dynamic languages.

1.2 Thesis Goals

1.2.1 Identifying Sources of Overhead

Previous studies have attempted to identify the sources of inefficiencies in dynamic languages. However, they have generally been limited in scope. They either focus on a few sources of overhead or are only able to evaluate a few benchmarks. Some studies merely rely on a qualitative understanding of software inefficiencies in order to propose optimizations.

The first goal of this dissertation is to more comprehensively study the following research question: *What are the sources of inefficiencies in dynamic languages?* To better understand which overheads are worth optimizing, a quantitative analysis of a large number of overheads using a large number of benchmarks needs to be performed.

In addition, how microarchitectural and runtime parameters affect the performance of these languages needs to be studied. Finally, the findings should be generally applicable to various implementations and languages.

1.2.2 Single-Application Memory Management Performance

Once the various sources of overhead have been identified, the dissertation focuses on reducing the overheads and optimizing single-application memory management performance. The following research question is investigated: *Can the garbage collection overhead be reduced while optimizing the cache performance?* In the baseline design, tuning runtime parameters to lower garbage collection overhead hurts the cache performance of the program by increasing the last-level cache miss-rate. The overall program performance can be improved if the impacts on cache misses are reduced when garbage collection overhead is reduced.

1.2.3 Efficient Tuning on Multi-core Processors with Shared Caches

Multi-core processors improve resource utilization by sharing the memory hierarchy among multiple cores. In particular, the last-level cache (LLC) is shared by multiple applications that are running concurrently. In some cases, cache contention between the concurrent applications can significantly hurt performance. Cache partitioning is a traditional approach that limits the impact of cache contention for unmanaged static languages where memory access patterns are fixed.

In dynamic languages, automatic memory management parameters can be additionally adjusted to alter memory access patterns and improve performance. Therefore, the

following research question is investigated: *Can automatic memory management parameters be tuned to achieve good performance on multi-core processors with shared caches?* The impact of cache sharing on the optimal memory management parameters needs to be modeled and schemes for adjusting the parameters need to be developed.

1.3 Contributions

This dissertation proposes to reduce the performance gap between dynamic and static languages by co-optimizing the software runtime and hardware architecture for dynamic languages. The optimizations at these layers are transparent to the programmer and require no change to the language definitions or the programs. The results show that there are opportunities for significantly improving application execution time by targeting inefficiencies at these layers.

This dissertation quantifies the sources of overhead in dynamic languages and proposes optimizations related to memory management performance. Optimizations for both single-application performance as well as the performance of multiple concurrent applications running on a multi-core processor are considered. In particular, the dissertation makes the following three contributions.

1. The dissertation presents a comprehensive breakdown study of Python for a large number of benchmarks [70]. The breakdowns of the Python interpreter execution time are presented. Sweeps of microarchitectural parameters provide better understanding of which aspects of hardware designs affect performance of Python runtimes. The trade-off of cache performance and garbage collection time is analyzed. Finally, it is shown that key findings can apply to other dynamic languages

and runtime implementations. The study provides new insights regarding the opportunities to improve the performance of Python and other dynamic languages. C function calls represent a major source of overhead not previously identified. The microarchitectural study shows that dynamic languages exhibit low instruction-level parallelism and that the presence of JIT lowers sensitivity to branch predictor accuracy and increases sensitivity to memory system performance. Finally, nursery sizing is shown to have a large impact on dynamic language performance and needs to be done in an application-specific manner, considering the trade-off between cache performance and garbage collection overhead, for the best result.

2. The dissertation proposes a new invalid memory region tracking mechanism that improves single-application performance through cache installation of newly-allocated objects, write-back reduction, and pollution control [69]. The proposed solution works with small objects commonly used in dynamic languages. In addition, a partial tracing algorithm, which can be run to identify invalid cache lines with lower overhead compared to full garbage collection, is described. Experimental results show that this co-optimization of garbage collection and cache performance can achieve significant improvements in execution time; 22% improvement on average and up to 68% improvement for PyPy [15], a popular implementation for Python, and 17% improvement on average and up to 63% improvement for V8 [52] running JavaScript.

3. The dissertation identifies that cache sharing degrades dynamic language performance and proposes using automatic memory management to reshape memory accesses to reduce cache contention among concurrent programs. An analytical model is developed to better understand the interactions between the execution time of a program and the cache and nursery sizes. A static nursery sizing scheme based on offline profiling is presented as well as a dynamic nursery sizing scheme

that can automatically adjust the nursery size at runtime without offline profiling. The proposed schemes are implemented and evaluated on a real-world system. The results show that the proposed nursery sizing schemes are quite effective in determining good nursery sizes and can significantly increase performance over the baseline. When four programs run concurrently, the static scheme based on offline profiling improves the system throughput by 18.5% on average and up to 92%. This performance improvement is within 7.5% of the best nursery sizing on average. The performance improvements for individual programs can be as high as 3.28x. The dynamic scheme also provides performance improvements comparable to the static scheme on average.

1.4 Organization

The rest of the dissertation is organized as follows. Chapter 2 describes key features of dynamic languages and details of their implementation. Chapter 3 discusses the study on the sources of overhead for Python. Chapter 4 discusses the hardware support and software optimizations that reduce modern memory management overhead for a single application in dynamic languages. Chapter 5 discusses offline and online schemes to adjust memory management considering the cache sharing between multiple applications running on multi-core processors. Related work is surveyed in Chapter 6 and Chapter 7 discusses future work. Finally, Chapter 8 concludes the dissertation.

CHAPTER 2

BACKGROUND

2.1 Features of Dynamic Languages

Dynamic languages parse and execute code at runtime. Since code is generally not pre-compiled, they can support a range of features not found in static languages. For example, they support dynamic typing where variables types do not need to be explicitly written in code. In addition, code and classes are first-class types and can be modified at runtime. As a result of the flexibility provided by dynamic languages, programmers can express more complex algorithms with less code leading to more productivity. Unfortunately, supporting these features requires more computational overhead than writing explicit programs with well defined types and data structures. This section briefly describes common features of dynamic languages and why they add computational overhead. In the following sections, some details about implementations of dynamic language runtimes are discussed.

All variables in dynamic languages are usually dynamically typed and can be redefined at runtime. The programmer does not need to specify the type of a variable and can use the same variable to hold objects of multiple types. One advantage of dynamic typing is that functions can be reused for multiple object types as long as the input variables support all the operations performed in the function. Classes can also be generated and modified at runtime based on inputs to the program. For example, a class can be constructed from a `JSON` file. The performance impact of supporting dynamic types is that memory for all objects must be dynamically allocated, since the size of the object cannot be known until runtime. In addition, functions cannot be optimized using traditional compiler techniques such as unboxing of primitives where `int` values are directly

stored in hardware registers or common subexpression elimination where expressions are simplified by the compiler to reduce computation.

Dynamic languages have runtime checks and automatic memory management built-in. One advantage of these features is that debugging and testing is simplified; the program will return an exception if a runtime check fails. Furthermore, these features eliminate the possibility of memory leaks and low-level memory bugs such as buffer overflow. The performance impact of supporting these features is that they are performed even in cases where it is clear to the programmer that there would not be any errors. For example, accessing the first element of a fixed-size array will still perform a bounds check operation. In addition, the programmer no longer has direct control over placement of objects in memory, so optimizing accesses for cache performance may no longer be possible.

Dynamic languages are interpreted and programs do not need to be compiled ahead of time. One advantage of this is that code can be modified with minimal compilation cost and can even be changed at runtime. Furthermore, input-specific code can be generated at runtime using `eval` statements. The performance impact of supporting these features is that interpretation is slow and optimizing code at runtime requires complex just-in-time (JIT) compilers that will add overhead to the program execution when they are run.

2.2 Interpreter Design

In order to run a program, it is parsed and translated to a series of bytecode instructions. The language runtime contains an interpreter that reads the bytecode instructions and executes the necessary actions. The interpreter is often implemented as a virtual machine

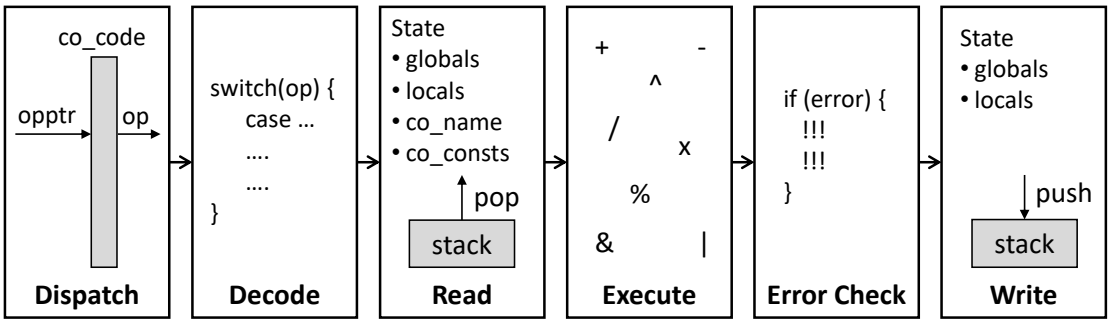


Figure 2.1: Overview of CPython virtual machine architecture.

(VM) with local storage and a dispatch loop. During each loop iteration, a bytecode instruction is fetched from an array and is executed. Compared to a traditional compiler, the interpreter is usually simpler to design and can more easily implement complex language constructs.

For example, Figure 2.1 shows the stages of interpreter execution in CPython [31], the official runtime for Python. It is implemented in C as a stack-based VM with some enhancements to support language-specific constructs. The VM uses `opptr` to index into the bytecode array (`co_code`) and retrieve the appropriate bytecode. The bytecode is then decoded using a *switch-case* construct. Data is read from the stack or other storage variables. The operation specified by the bytecode will be executed using the read data as operands. Some error-checking code will ensure that the execution completed successfully. Finally, data will be written back to the stack or other storage variables. `opptr` will be updated and the process will repeat until the program completes.

2.3 Just-in-Time Compilation

Interpreters are generally slow because they perform no optimizations across bytecode instructions. Just-in-time compilation can optimize runtime performance by converting

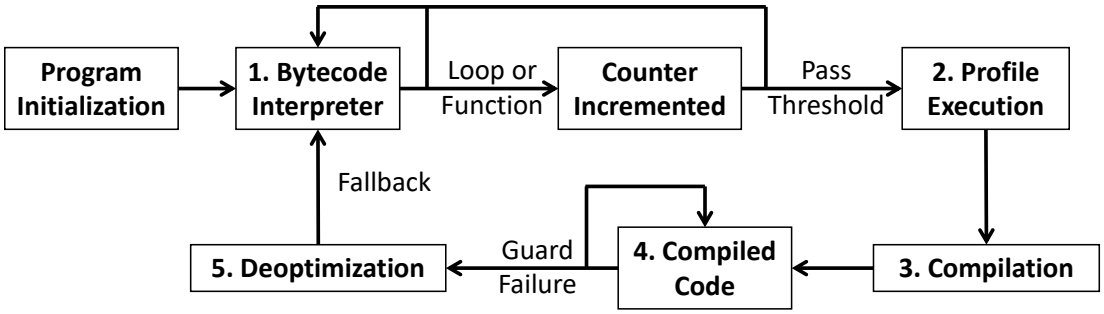


Figure 2.2: Steps in just-in-time compilation.

interpreted bytecode to machine code. In addition, runtime information about object types and values can be used to perform additional optimizations that cannot be done ahead-of-time. Running the just-in-time compiler during runtime is relatively expensive and the cost of compilation must be amortized by the performance improvement in the compiled code. For this reason, JIT compilers focus on frequently executed code, such as frequently executed loops or functions.

As shown in Figure 2.2, counters are used to track the number of times that loops or functions execute. Once the counter reaches a threshold, the loop or function is considered a good candidate for compilation. An additional profiling stage collects information for the compiler optimizations. The code is then compiled and the machine code is executed in place of the interpreted bytecode.

To generate optimized code, the compiler makes assumptions about variable types and values, and it inserts guards to check whether those assumptions are valid during the execution. If there is a failed guard, the compiled state is rolled back to a valid interpreted state and the bytecode interpreter continues execution. This is called *deoptimization* and is a relatively expensive operation that could affect overall performance if it occurs too frequently. Additional steps can be added to the JIT process to better handle repeated guard failures and optimize a portion of a function or loop.

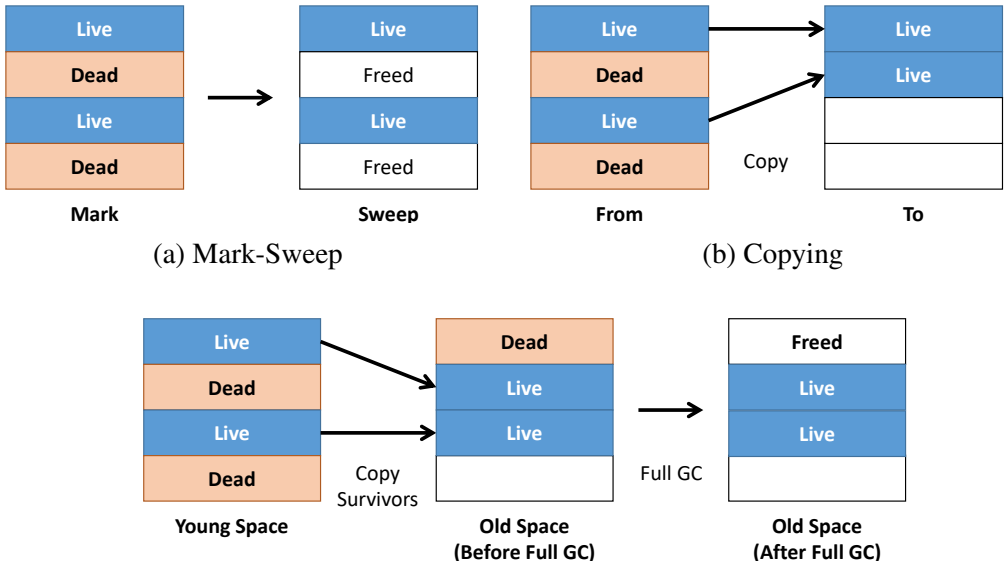


Figure 2.3: Graphical depictions of how the different classes of garbage collection algorithms work.

2.4 Automatic Memory Management

Automatic memory management is a key feature of dynamic languages that can 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.

2.4.1 Garbage Collection

In a language with automatic memory management, garbage collection is used to free memory from objects that are no longer in use. The process of determining which objects are live and which are not incurs non-trivial performance overhead. 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 [6]. Garbage collectors are either *reference-based* or *trace-based*. *Reference-based* garbage collectors track the number of references to an object and free the object once the reference count reaches zero. While simpler to implement, they generally are less efficient and cannot deal with cyclic data structures without additional complexity.

A *trace-based* garbage collector will run at intervals and traverse object pointers to determine which objects are live. It starts from a set of root pointers and follows them. It will then follow additional pointers that it encounters in the process. Objects with pointers pointing to them are live, while objects without any valid pointers pointing to them are dead and can be collected. In a managed language, the runtime has knowledge of all pointers in the system unlike low-level languages such as C or C++, where any variable can be dynamically cast to represent a pointer [14]. Once the live and dead objects are differentiated, the garbage collector will free memory corresponding to dead objects. Mark-sweep and copying are two common types of *trace-based* garbage collector algorithms.

Figure 2.3(a) shows how the mark-sweep garbage collector works [90]. It marks live objects during the mark phase and frees unmarked objects during the sweep phase. The advantage of this algorithm is that it can run with no additional memory overhead. All objects are freed in place and live objects are kept in the same location. It can easily be performed incrementally allowing for better responsiveness in the program. The main disadvantage is that the memory is left fragmented. A compact phase [28] can be added to perform defragmentation when needed.

Figure 2.3(b) shows how the copying garbage collector works [49]. It works by splitting the memory space into two semi spaces, one called the *from* space and the

other called the *to* space. During garbage collection, all live objects are copied from the *from* space to the *to* space. During the next garbage collection cycle, the space names are switched and the same process occurs. The main advantage of this algorithm is that dead objects are automatically freed and there is no fragmentation following the garbage collection. The disadvantages are that half of the memory space must be reserved for garbage collection and that pointers for live objects need to be updated once the object is copied. This makes it difficult to perform incrementally and as a result most copying garbage collectors will pause the application until they finish completely.

Generational garbage collection [85] is an optimized form of garbage collection that is used in many high performance implementations of modern languages. The memory is separated into subspaces based on object age and different garbage collection algorithms can run on the different subspaces. Figure 2.3(c) shows the simplest implementation of generational garbage collection. There is one subspace for young objects, sometimes called a *nursery*, and another subspace for old objects. Objects are allocated in the nursery and are moved to the old space if they survive long enough.

Efficient generational garbage collection relies on the assumption that most objects in a program die young. Therefore, a copying garbage collector can efficiently move a small number of surviving objects from the nursery to the old space. Once the object is in the old space, a slower garbage collector, such as a mark-sweep collector, can run less frequently. This can be extended to any number of subspaces based on age.

As shown in Table 2.1, real implementations of generational garbage collection add variations to this general scheme. For example, the PyPy collector runs the mark-sweep collector incrementally in the old space [108]. V8 adds an additional semi-space in the nursery. During garbage collection, young objects are copied from one semi-space to the other and only move to the old generation if they have already been copied once [104].

Table 2.1: Summary of popular language implementations that use generational garbage collection.

Language	Implementation	Garbage Collector Description
JavaScript	V8[104]	Two generation collector with two young semi-spaces and and an old space. Young objects are copied from one young semi-space to the other and then to the old space if they survive.
Python	PyPy[108]	Two generation collector with a nursery and an old region. Young objects surviving in the nursery are copied and moved to the old region, where incremental mark-sweep garbage collection is used.
Ruby	Rubinius[119]	Concurrent generational collection.
Java	Hotspot[94]	Three generation collector with a young generation with three subspaces, an old space, and a permanent space. An object starts in the eden subspace of the young generation and is copied to one of two survivor spaces. If it survives, it is copied to the old space.
C#	.NET CLR[142]	Three generation collector with a young generation for short-lived objects, a buffer generation for semi-short-lived objects, and an long-lived generation.

2.4.2 Sequential Allocation

In order to make allocation fast in generational garbage collection, a sequential allocator is typically used. The *nursery* is always guaranteed to be empty following a garbage collection. The sequential allocator simply uses a pointer to maintain the invariant that anything before the pointer is allocated and anything after it is unallocated. On allocation, the pointer is incremented to maintain the invariant. A check is also performed to ensure the allocation does not exceed the nursery bounds. If it does, garbage collection will be run to empty the nursery and reset the pointer to the beginning of the nursery.

CHAPTER 3

STUDY: PYTHON SOURCES OF OVERHEAD

3.1 Overview

This chapter describes a quantitative study on the sources of performance overhead in Python, a popular dynamic language. The overhead of a dynamic language can come from multiple aspects of the language design space. This study explores three different aspects of the overhead to provide a more comprehensive view. First, at the language level, some features of the dynamic language may lead to inherent inefficiency compared to static languages. Second, a language runtime also adds overhead to dynamic languages compared to statically compiled code. This study breaks down Python execution time into language and runtime components as well as core computations to understand overhead sources. Finally, at the hardware level, the impact of dynamic language features on microarchitecture-level performance is studied by looking at instruction-level parallelism, branch prediction, and memory access characteristics. CPython [31], an interpreter-only design, is compared with PyPy [15], a JIT-based design, to understand the microarchitecture-level differences between the runtime implementations.

The study is broken into two main parts. The first part of the study looks at the language and runtime features of Python to understand which aspects of the language and runtime add additional overhead compared to C, the baseline static language. By annotating instructions at the interpreter-level, breakdowns for a large number of benchmarks can be generated. In addition to the sources of overhead already identified by previous work, C function calls are found to represent a major source of overhead that has not been previously identified.

The second part of the study looks at the interaction of the runtime with the underlying processor microarchitecture. Both CPython and PyPy are found to exhibit low instruction-level parallelism. Using PyPy with JIT helps decrease sensitivity to branch predictor accuracy, but increases sensitivity to cache and memory configurations. In particular, the generational garbage collection used in PyPy introduces an inherent trade-off between cache performance and garbage collection overhead. Frequent allocation of objects in dynamic languages increases a pressure on the memory hierarchy. However, increasing the garbage collection frequency to improve cache performance can lead to high garbage collection overhead. The study shows that the optimal nursery size depends upon application characteristics as well as runtime and cache configurations. If the nursery is sized considering the cache performance and garbage collection overhead trade-off, then there can be significant improvements in program performance.

While the study focuses primarily on Python, the main results are applicable to other dynamic languages as well. Some evaluation is performed with V8 [52], a high-performance runtime for JavaScript, to show that the main lessons still apply. Finally, a study on JIT thresholds is discussed to show additional opportunities for improving dynamic language performance.

The following summarizes the main contributions in this chapter:

1. A comprehensive breakdown study of the CPython interpreter execution time for a large number of benchmarks.
2. Microarchitectural parameter sweeps to better understand which aspects of hardware designs affect performance of both the interpreter-only CPython and PyPy with and without JIT.
3. A detailed analysis of the trade-off of cache performance and garbage collection time for PyPy.

The following new insights are identified regarding opportunities to improve the performance of Python and other dynamic languages:

1. C function calls represent a major source of overhead not previously identified.
2. The microarchitectural study shows that dynamic languages exhibit low instruction-level parallelism and that presence of JIT lowers sensitivity to branch predictor accuracy and increases sensitivity to memory system performance.
3. Nursery sizing has a large impact on dynamic language performance and needs to be done in an application-specific manner, considering the trade-off between cache performance and garbage collection overhead, for the best result.

The rest of this chapter is organized as follows. Section 3.2 explains the experimental setup. Section 3.3 discusses the study on the sources of overhead for Python and Section 3.4 analyzes the interaction between the runtime and the underlying hardware. Section 3.5 discusses the study on JIT thresholds.

3.2 Experimental Setup

Experiments were run on an infrastructure based on Pin [86]. The Pin framework enabled instrumentation of the runtimes at both the instruction-level and function-level without having to modify the source code and without affecting the instructions executed by the program. Pin tools were developed to capture dynamic instruction counts and other statistics needed for analysis.

To get cycle count estimates for a variety of memory hierarchies and core configurations, the Pin tools were interfaced with ZSim [121], a fast x86-64 simulator built on

Table 3.1: ZSim configuration.

Core	4-way OOO, 16B Fetch, 3.40GHz
	2-level 2-bit BP with 2048x18b L1, 16384x2b L2
	224 ROB, 72 Load-Q, 56 Store-Q
L1I	64 kB, 8-way, 4-cycle latency
L1D	64kB, 8-way, 4-cycle latency
L2	256kB, 4-way, 12-cycle latency
L3	2MB, 16-way, 42-cycle latency
Memory	16GB DDR4-2400, 173-cycle latency

Pin. ZSim was run with a configuration that was similar to an Intel Skylake processor. The details of the configuration are shown in Table 3.1. An out-of-order core model (OOO) was used for most of the experiments. For the sources of overhead experiments, the simple core model was used to be able to accurately map individual instructions to their cycle contributions. It was assumed that each of the four physical cores had one-quarter of the 8MB shared L3 cache available for use, so the L3 cache size available to a core was 2MB. DRAMSim2 [117] was integrated with ZSim to model DDR4-2400 memory.

For runtimes, CPython [31] 2.7.10 with the standard compiler optimization flags (`-O3`) was used as the Python interpreter and PyPy [15] 5.3.1 was used as the JIT-based runtime for Python. Experiments were run using 48 benchmarks gathered from the official Python performance benchmark suite [109] and from the PyPy benchmark suite. The designers of the official Python performance benchmark suite have mentioned that their suite focuses on real-world benchmarks, using whole applications when possible, rather than synthetic benchmarks. Each benchmark was warmed up by running it 2 times followed by running it 3 times for evaluation.

Some experiments were performed using Google V8 [52] 4.2.0, a popular high-performance JIT-based runtime for JavaScript. The experiments were run using 37

benchmarks from the JetStream [17] benchmark suite, which combines benchmarks from other suites including SunSpider, Octane, and LLVM. Benchmarks were run 3 times for evaluation.

3.3 Sources of Overhead

This section presents a quantitative study on the sources overhead of Python compared to static languages such as C. Various sources of overhead are identified and categorized, a methodology is presented to break down the Python execution time by overhead category, and the main findings are discussed. The results in this section are reported for CPython. Evaluation is also presented to show that some findings are applicable to both PyPy and V8.

3.3.1 Overhead Breakdown

Table 3.2 shows the overhead sources that were identified and evaluated in this study through careful review of language features as well as CPython source code. The overhead categories can be placed into three groups. The language features of Python may incur overhead because they either do not exist in a static language or require additional dynamic operations. The interpreter itself also adds additional performance overhead that compiled code would not have. A majority of the features have been previously identified and evaluated either directly or indirectly (e.g. through an optimization). References are provided in the table to the previous work which evaluates those features. In addition, this study identifies three new overhead categories not evaluated in previous work. They are also indicated in the table. The different components and overhead categories are described further in the following subsections.

Table 3.2: Sources of performance overhead for Python.

Group	Overhead category	Description	Studied by
Additional Language Features	Error check	Check for out-of-bounds, overflow, and other errors	*NEW*
	Garbage collection	Automatically freeing unused memory	[11, 67]
	Rich control flow	Support for more condition cases and control structures	[26, 67]
Dynamic Language Features	Type check	Checking variable type to determine operation	[11, 18]
	Boxing/unboxing	Wrapping or unwrapping integer or float types	[11, 18]
	Name resolution	Looking up variable in a map	[26]
	Function resolution	Dereferencing function pointers to perform an operation	[26]
	Function setup/cleanup	Setting up for a function call and cleaning up when finished	[26, 11, 67]
Interpreter Operations	Dispatch	Reading and decoding bytecode instruction	[22, 26]
	Stack	Reading, writing, and managing VM stack	[26, 18]
	Const load	Reading constants	[26]
	Object allocation	Inefficient deallocation followed by allocation of objects	[67]
	Reg transfer	Calculating address of VM storage	*NEW*
	C function call	Setting up and cleaning up from calling helper functions in the interpreter	*NEW*

Additional Language Features

This category consists of language features that do not exist in static languages such as C. The `errorcheck` overhead comes from runtime checks that Python performs to guarantee safety and robustness. After an operation, Python performs checks such as an overflow check on the `int` types and bound checks on the `list` types. The `garbage collection` overhead comes from operations for runtime garbage collection such as

maintaining reference counters and freeing memory. The `rich control flow` overhead results from checking various conditions in the case of richer evaluation of condition or for managing the block stack in the case of support for more control structures.

Dynamic Language Features

This category captures language features that exist in C but requires additional runtime operations in Python. A majority of these features are managed statically in C at compile time. Setting up a function call and cleaning up on a return is done dynamically in C through the calling convention, but it requires significantly more computation in Python. The overheads in this group would still be present even if Python programs were compiled ahead-of-time because the compiler lacks necessary runtime information. Python uses dynamic typing, so types of the variables and where they are allocated are not known until runtime. Python cannot resolve types of variables statically because they are not explicitly given in the program. In addition, the variables with unknown types cannot be allocated statically so the locations of variables are only known dynamically.

The `typecheck` overhead relates to all checks the interpreter must perform to determine the type of the variable. In Python there is usually a check for variable type before an operation is performed on the object. The `boxing and unboxing` overhead relates to reading integer and float primitives values from the object and writing back these primitive values to the object. These primitives would normally be stored in machine registers for a C program, but are represented as objects with type information in Python. For example, in an add operation, the values of the two variables to be added will be read from the corresponding object. The sum will be computed and will be written back to another object representing the sum.

The `name resolution` overhead relates to looking up the variable pointer in a map by using the variable name as the key. Types of global variables are not known and they can be created and destroyed dynamically, so Python uses maps to store pointers to the variables. The `function resolution` overhead relates to dereferencing of function pointers. Functions in Python are first class objects that can be created and destroyed, so Python stores function pointers for common operations related to an object.

The `function setup/cleanup` overhead relates to setting up a call to a function and cleaning up on a return. In order to setup up a call, Python needs to determine the function type (both Python and C functions are supported). If it is a C function, then the inputs passed in through the C extension interface and the output needs to be returned. If it is a Python function, an execution frame for the function needs to be allocated. Functions that require variable arguments require special attention. Once the function returns, Python needs to deallocate the frame and pass the return value to the caller.

Interpreter Operations

In addition to categories relating to language features, there are categories related to the overhead of running the interpreter. These relate to the cost of emulating a virtual machine on a physical machine. The `dispatch` overhead relates to reading the bytecode and decoding it to perform the correct operation. This includes the execution of the dispatch loop and a switch statement for decoding.

CPython is a stack-based virtual machine. The `stack` overhead relates to operations for managing the stack. Operations read from the stack and write to the stack. The stack is local storage for the VM similar to the register file for the CPU. The stack is not meant to store program state, but act as local storage for intermediate values. There are some

bytecodes for explicitly managing the stack, such as `DUP_TOP` which duplicates the top entry. In addition to the stack, there are data structures which store constant values. The `const load` overhead is the overhead of loading constants to the stack. Constants are stored in the `co_const` array. The values first need to be loaded to the stack before they are used by other bytecodes.

In the interpreter implementation, there are certain objects that could be reused but are instead deallocated and reallocated. The `object allocation` overhead captures the case an object is deallocated then reallocated. For example, most method frames are allocated during execution of the method and deallocated when it finishes. In addition, arithmetic operations take operands from the stack and generate a new value. When the operation completes, the original operands are deallocated and a new object is allocated for the value.

Since CPython is written in C, there may be additional inefficiencies introduced by how the interpreter is written. The `C function call` overhead captures the additional cost of setting and cleaning up C functions in the interpreter. This includes the cost of creating and destroying stack frames and performing the call. The use of a C function to write good refactored code results in many function calls per bytecode instruction. These calls cannot be inlined in most cases because function pointers are used.

When reading a VM data structure, such as the stack, the CPU will first load the address of the data structure first to the machine registers. Then it will compute the effective address of the Python variable (e.g. top of stack). Finally, it will load the Python variable into the machine register. This additional step of finding the data structure of the VM and loading it to machine registers is categorized as `reg transfer`.

3.3.2 Analysis Approach

Based on the described overhead categories, the goal was to develop an analysis tool that could return the contribution of each category to the overall execution time. Since the Python program is running on a statically-compiled interpreter, each instruction of the interpreter can be annotated with a category label. When running the program with the annotated interpreter, a breakdown of the time spent in each category can be generated for any Python program with no additional effort.

The annotations of the execution must relate to the execution of the whole Python program and not just the sources of overhead. Some instructions can be directly annotated with the overhead categories summarized in Table 3.2. Other instructions are needed to execute the program and cannot be annotated with an overhead category. For example, a Python `BINARY_ADD` bytecode has overheads associated with type checking, unboxing, error checking, etc., but also performs an ALU add operation between the two variables. Instructions needed to execute the program are annotated with an `execute` label. The analysis breakdown includes the contribution of the `execute` category in addition to the overhead categories.

Annotating each static instruction alone cannot provide an accurate breakdown. There are cases where a function's annotation depends on the calling function. For example, CPython uses the same dictionary lookup function for both looking up a variable in a global map and for performing a lookup operation on a map data structure used in the Python program. In the case of looking up in the global map, the function should be annotated with the `name resolution` overhead category. In the other case, the function should be annotated with the `execute` category. The call sites of these functions can be used to support different labels.

An alternative analysis approach to quantifying the different overhead sources would be to start with C code of a program and transform it to a Python program while iteratively introducing the necessary language features and implementation details. At each step, the slowdown of introducing the additional feature could be measured. Based on this, the performance gap between Python and C for a given program could be understood. This process is very tedious and would be difficult to apply to many programs. Similarly, starting with a Python program and introduce more static features into the language to eliminate the dynamic runtime overhead would also be tedious and hard to apply to many programs.

Gathering Statistics with Pin

In order to implement the analysis method, Pin [86] was used to instrument the CPython interpreter. To make the analysis more flexible, a Pin tool was written to export essential runtime statistics and a post-processing step was then performed to generate the breakdown. The Pin tool exported statistics for some of the functions in the interpreter at the instruction granularity. For these functions, the total execution time of the static instruction at the given PC value was exported.

If a function could be labeled by a single category, then statistics were exported at the function granularity to limit the size of the statistics files and to make annotations more feasible. In addition to the function name and its execution time, the origin PC was also exported. The origin PC is the most recent PC in the call trace that belongs to a function that is being annotated at an instruction granularity.

Some categories were automatically annotated by the Pin tool by detecting instruction sequences in the assembly code. For example, the Pin tool could identify and cate-

gorize the assembly instructions relating to the `C function call` and `reg transfer` overheads. The time spent running instructions for each of these categories was exported.

Cycle Count Estimates Using ZSim

While instruction count may be a good first-order estimate, it does not capture micro-architectural aspects such as memory latency and branches. The Pin tool was interfaced with the ZSim [121] simulator (also a Pin tool) to estimate cycle counts in addition to the instruction counts. ZSim has an out-of-order core model that can model an out-of-order pipeline as well as branch mispredicts and cache misses. However, attributing cycle counts to a single instruction becomes challenging for an out-of-order core because the latency of an instruction in the pipeline can be affected by other instructions also in the pipeline.

Instead, the simple core model was used to measure the number of cycles each instruction took to execute. In the simple core model, instruction latency is only affected by misses in the instruction and data caches. Otherwise, an instruction takes a single cycle. Including memory latency in the execution time gave a better first-order estimate of the sources of overhead than just dynamic instruction counts.

Post-Processing

During post-processing, annotated source line numbers in the interpreter were mapped to PC values in the exported statistics file. For functions that were annotated at the instruction granularity, each line of the CPython source code was annotated with a category. Functions that were annotated at the function granularity were either anno-

tated with a single category based on the name of the function or by multiple categories based on the name of the function and the origin PC if the category was caller-specific. CPython was compiled with the (`-g`) flag to be able to match source lines with PC values. By running the post-processing, a breakdown of the execution time (in CPU cycles) for each of the categories was generated.

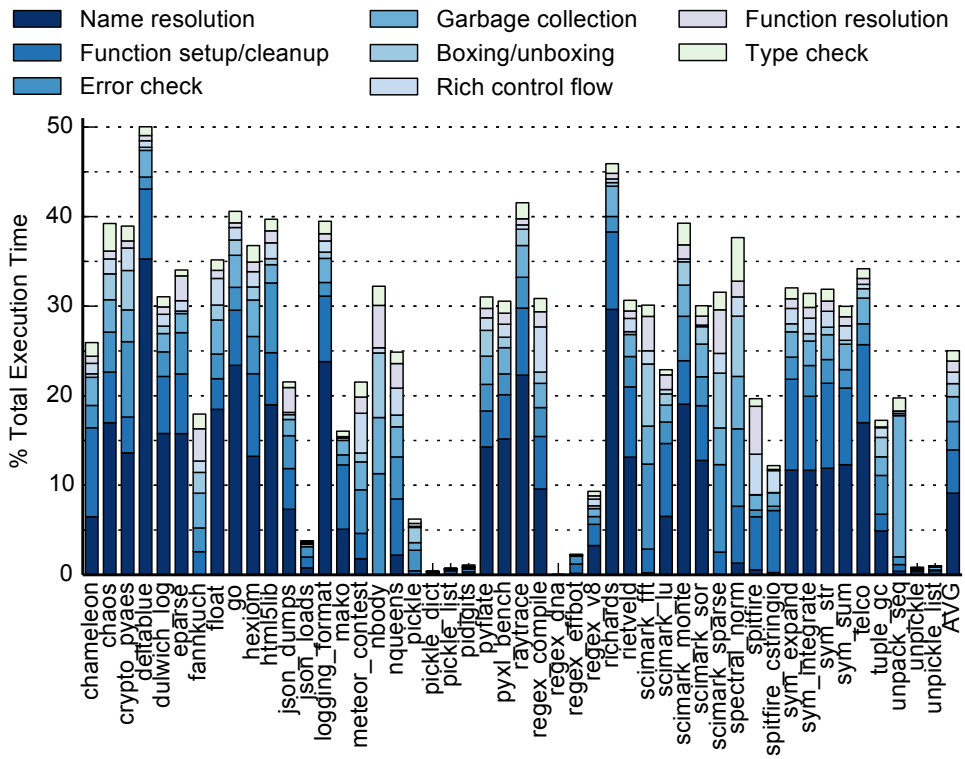
The CPython interpreter only needed to be annotated once and not for each Python program. Since all Python programs ran on the same interpreter, the PC values and source line number mappings for the interpreter remained the same and the annotations could be reused. As a result, a large number of Python programs were analyzed and compared with the same set of annotations.

3.3.3 Experimental Results

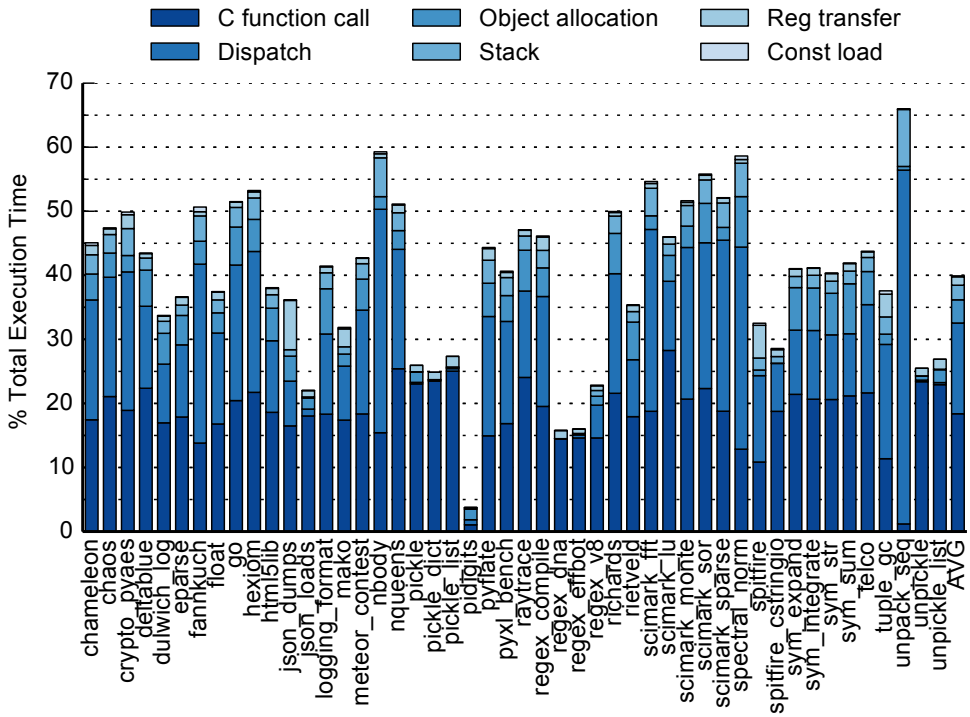
Execution Time Breakdown

Figure 3.1(a) shows the contributions of language features (both additional and dynamic) come from many categories, all adding up to a significant portion of the total execution time. Among these categories, `name resolution` and `function setup/cleanup` overheads dominate with 9.1% and 4.8% average overhead respectively. To reduce the impact of `function setup/cleanup`, Python functions can be inlined. For `name resolution`, variable look-ups can be cached [26].

Figure 3.1(b) shows the contributions of interpreter operations to the execution time. `C function calls` and `dispatch` are major contributors overall with 18.4% and 14.2% average overhead. In previous work, `dispatch` has been repeatedly identified as a major source of overhead [77, 22]. However, `C function calls` have not been identified as a major source of overhead in the context of interpreters.



(a) Language features



(b) Interpreter operations

Figure 3.1: Overhead breakdown for CPython.

Some previous work has focused on optimizing BTB performance of indirect branches and indirect calls in interpreters [23, 47]. Additional analysis shows that indirect calls (but not indirect branches) account for an average of 11.9% and up to 19.0% of the C function call overhead, representing an average of 1.9% and up to 4.1% of the overall execution time. Therefore, there are other aspects of the C function call overhead that more significantly impact the execution time, such as setting up and destroying the stack frame. These aspects should also be studied and optimized.

On average, the identified overheads account for 64.9% of the overall execution time. The remaining 35.1% is used for the execution of the program. Therefore, there is at least 2.8x increase in execution time on average moving from a C-like program to a Python program running on CPython due to language and interpreter overheads. In reality, the program written in C can run one or two orders of magnitude faster than the program written in Python [67] because the C compiler can further optimize the program using static information about types and memory layout of objects.

During execution, the Python programs spend an average of 7.0% of their overall execution time calling library code written in C. Some benchmarks such as `pickle_dict`, `pickle_list`, `regex_dna`, `regex_effbot`, `regex_v8`, `unpickle`, and `unpickle_list` spend more than 64% of their time executing C library code. As a result, the overhead categories account for a smaller percentage of the execution time. However, `C function call` overhead still exists and is still significant even in the C library code.

Applicability to Other Runtimes and Languages

Since C function call overhead can be automatically annotated by the Pin tool by detecting the instructions related to the calling convention, the Pin tool was used to analyze

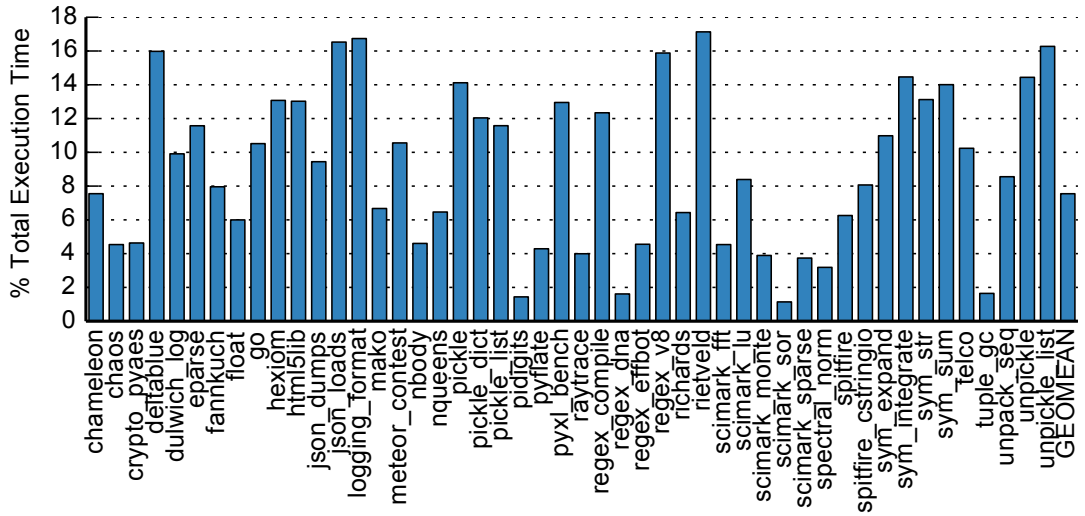


Figure 3.2: C function call overhead for PyPy.

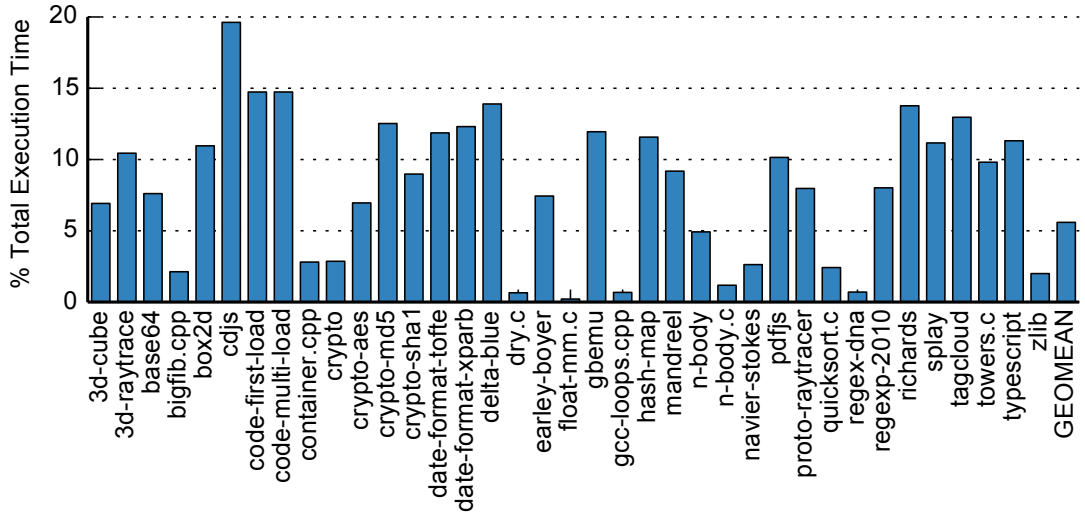


Figure 3.3: C function call overhead for V8.

the C function call overhead of PyPy with JIT and V8. Figure 3.2 and Figure 3.3 show that this overhead is significant in these runtimes as well with 7.5% and 5.6% average overhead for PyPy and V8, respectively. The JIT compilation reduces some of the overhead by inlining methods and generating traces. These results indicate that optimizing C function call overhead is important for achieving good performance in dynamic languages in general.

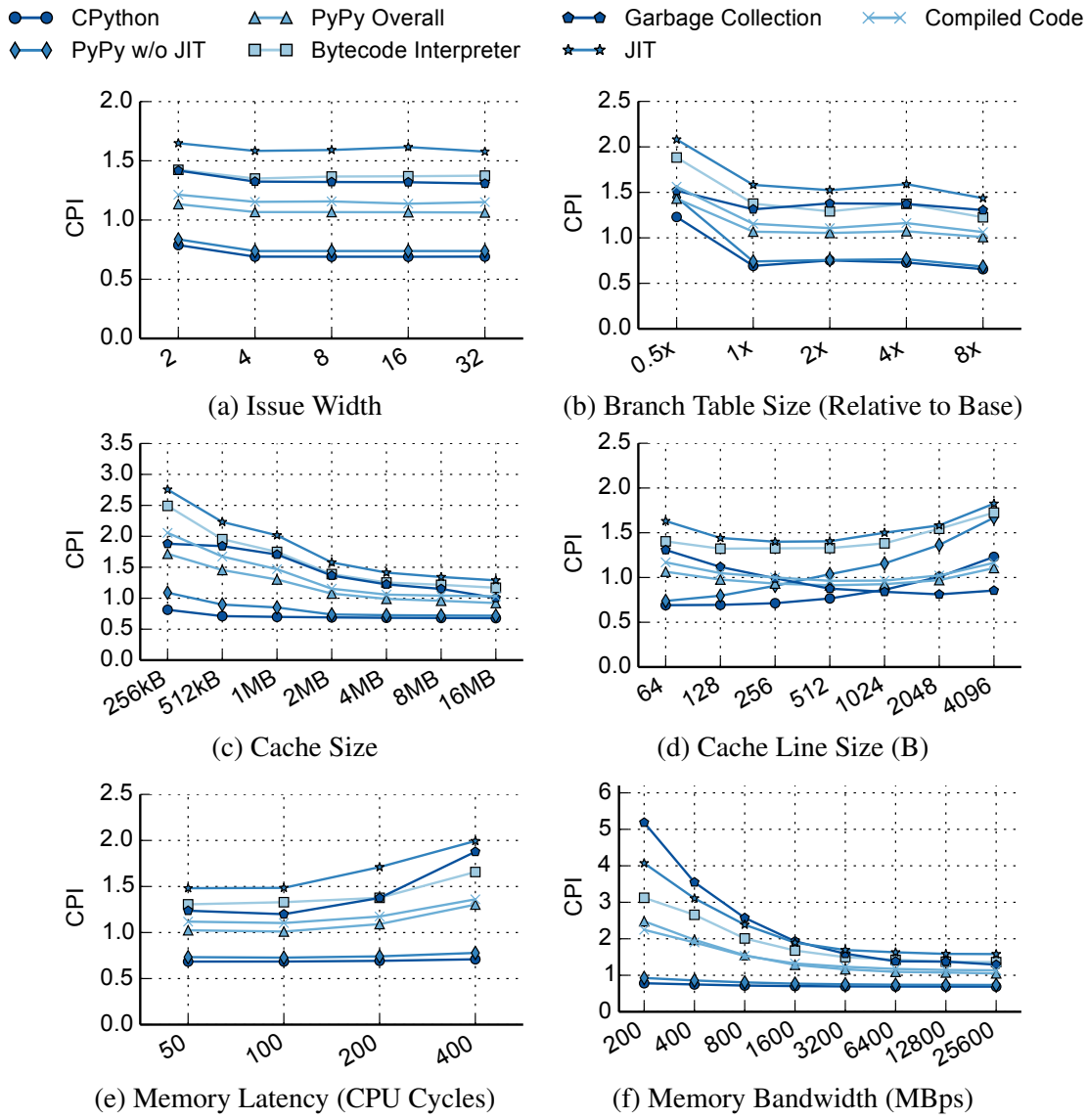
3.4 Interaction with Hardware

So far, the effects of underlying hardware on program behavior has not been considered. This section explores how the runtimes interact with the underlying hardware. First, the sensitivity of the runtime performance to various microarchitectural parameters is studied. PyPy performance is found to be sensitive to cache hierarchy and memory system parameters. Since memory management is a key contributor to cache performance, the interaction of memory management with the underlying hardware is then studied.

3.4.1 Microarchitecture Parameter Sweeps

This section explores the sensitivity of runtime performance to various microarchitectural parameters. Benchmarks were run on CPython and PyPy with and without JIT to see if there were differences in the sensitivity between an interpreter-based runtime and a runtime that additionally uses JIT compilation. Figure 3.4 shows how the CPI (cycle-per-instruction) changes as various microarchitecture parameters are swept. Here, the average CPI numbers across all benchmarks are shown. The PyPy with JIT execution is additionally broken down into different phases of execution by annotating PyPy at the function granularity using Pin. For the issue width sweep, the fetch width was set to be large to prevent it from becoming a bottleneck. The fetch width sweep results are not shown but show a similar trend as the result for issue width.

The results show that the performance of both CPython and PyPy are relatively insensitive to the processor fetch width and issue widths, suggesting that there is low instruction-level parallelism. The branch results indicate that merely increasing the branch prediction table size does not improve branch prediction accuracy enough to



(a) Issue Width

(b) Branch Table Size (Relative to Base)

(c) Cache Size

(d) Cache Line Size (B)

(e) Memory Latency (CPU Cycles)

(f) Memory Bandwidth (MBps)

Figure 3.4: CPI with microarchitecture parameter sweeps. A line is shown for each runtime as well as phases in PyPy execution.

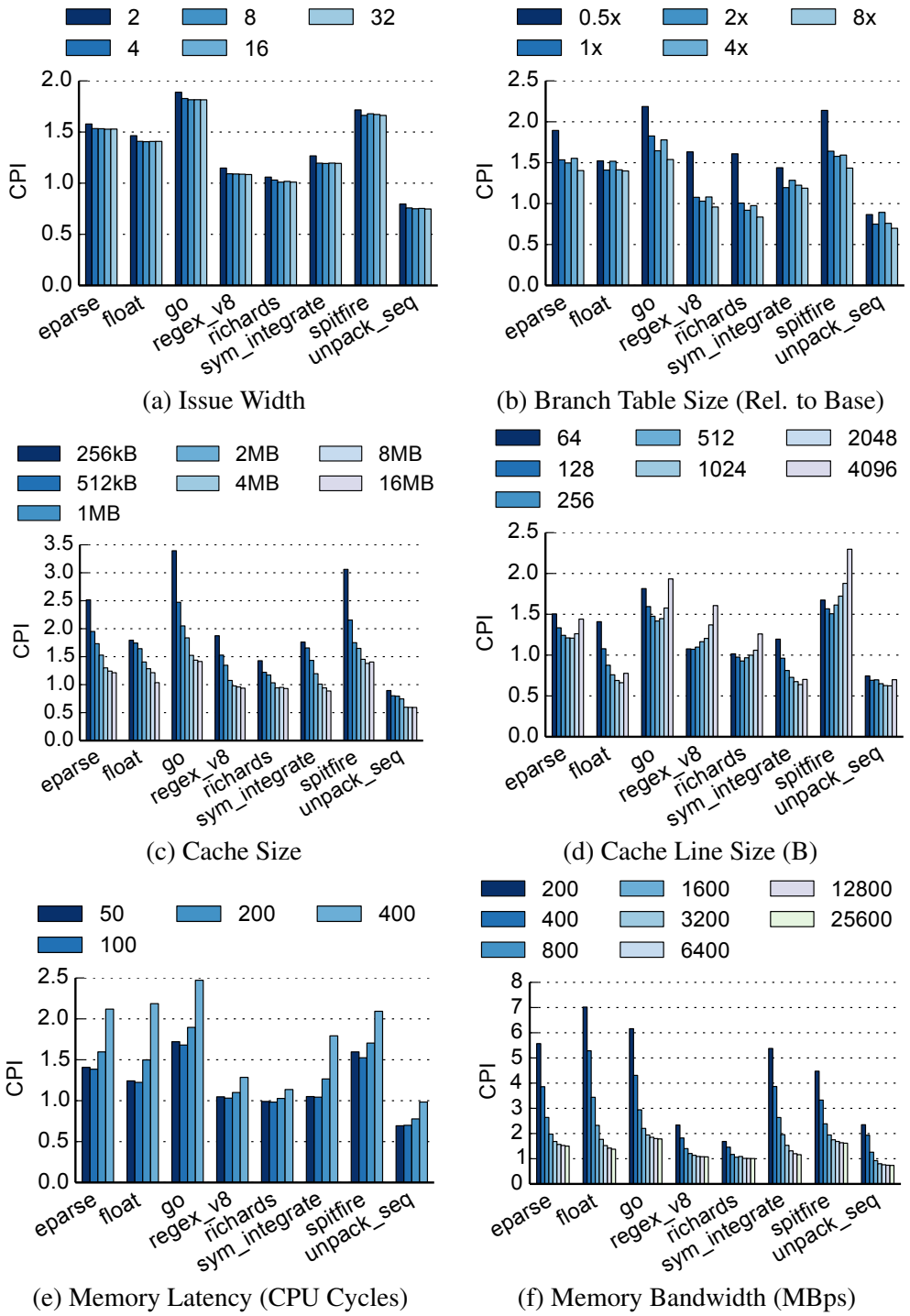
impact performance. However, when the table is too small and prediction accuracy suffers, the interpreter-based runtimes suffer more than a runtime with JIT. This indicates that JIT helps lower sensitivity to branch prediction accuracy.

On the other hand, cache and memory parameters have significant impacts on performance of PyPy with JIT. This indicates that the JIT significantly increases pressure on the memory hierarchy. In particular, the performance depends heavily on cache sizes.

Interestingly, the same programs running on the CPython interpreter and PyPy without JIT do not require a large cache. This indicates that the working set of an application itself is not large. Therefore, there is no fundamental reason why the bytecode interpreter and compiled code phases of PyPy with JIT require a large cache to run efficiently. In addition, the CPI for PyPy with JIT is greater than the CPI for CPython and PyPy without JIT. This indicates that while the JIT lowers the number of instructions executed, each instruction takes more cycles to execute due to longer average memory access latency. This is further shown by the sensitivity of the PyPy with JIT to memory latency and bandwidth.

The cache line size sweep shows that PyPy with JIT benefits from using larger cache line sizes, while the interpreter runtimes do not. After a closer study, the need for a large cache and cache line sizes appears to come from the interaction of the memory management system with the caches. This observation introduces an interesting opportunity for performance optimization and is discussed in more detail in the next subsection.

Figure 3.5 shows the results of microarchitecture parameter sweep when the overall CPI is shown for a few of the benchmarks. The general trend is the same as the previous figure. Yet, this figure shows that the performance impacts of microarchitecture parameter changes depend on individual application characteristics. Note that it is possible for benchmarks to perform better with a higher memory latency (e.g. 100 vs. 50 cycle latency) due to the unpredictable nature of out-of-order instruction scheduling. The sweeps for V8, another JIT-based runtime, are shown in Figure 3.6. They show trends similar to PyPy with JIT indicating that the memory management interaction is important for other JIT-based runtimes as well.



(a) Issue Width

(b) Branch Table Size (Rel. to Base)

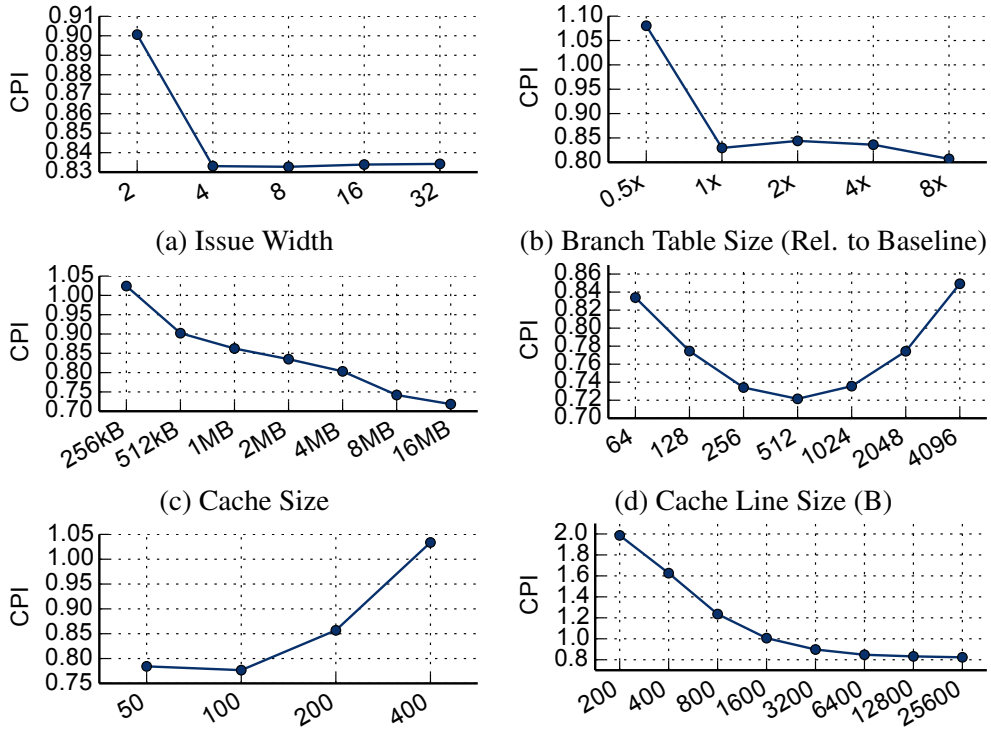
(c) Cache Size

(d) Cache Line Size (B)

(e) Memory Latency (CPU Cycles)

(f) Memory Bandwidth (MBps)

Figure 3.5: CPI with microarchitecture parameter sweeps. Each bar shows the overall CPI for one benchmark running on PyPy.



(a) Issue Width

(b) Branch Table Size (Rel. to Baseline)

(c) Cache Size

(d) Cache Line Size (B)

(e) Memory Latency (CPU Cylces)

(f) Memory Bandwidth (MBps)

Figure 3.6: CPI with microarchitecture parameter sweeps. The line shows the average CPI for V8.

3.4.2 Memory Management Interaction

This section shows that the memory management system contributes to the sensitivity in cache performance for PyPy with JIT. Proper sizing of the nursery is essential to achieving good cache performance. However, reducing the nursery size to improve cache performance may not lead to better overall program performance due to the increased overhead from garbage collection. The interaction of the memory management system with the cache hierarchy is explored in more detail.

Figure 3.7 shows the last-level cache (LLC) miss rates as a function of the nursery size. When the nursery is smaller than the cache size (i.e. 2MB), new objects can be allocated directly in the cache and miss rates are low. Once the nursery is too large to

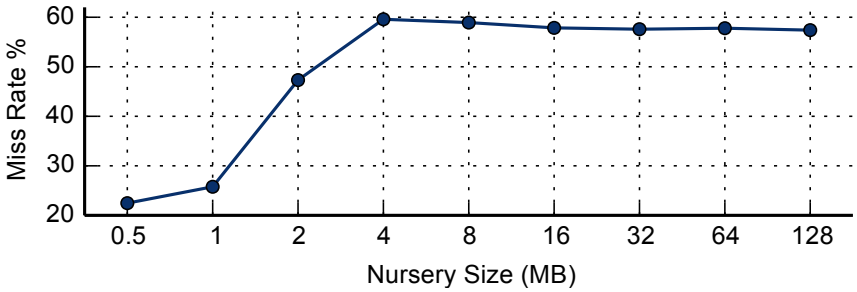


Figure 3.7: LLC miss rate as a function of nursery size.

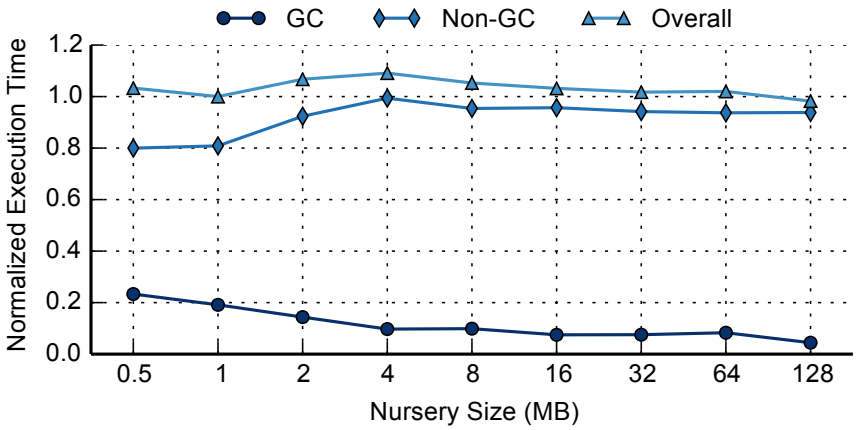


Figure 3.8: PyPy execution breakdown for different nursery sizes.

fit in the cache, cache thrashing occurs and most object initializations miss in the cache. The miss rate increases significantly by almost a factor of 2.4.

Figure 3.8 shows the breakdown of the execution time normalized to the overall execution time of running with a nursery that is half the cache size (i.e. 1MB nursery for 2MB cache) averaged across all of the benchmarks. It shows that on average the increase in cache miss rate hurts overall performance for nursery sizes slightly larger than the cache size. However, as discussed in Appendix A, garbage collection can be run less frequently by increasing the nursery size. This spreads some of the overhead over more of the program execution and reduces the number of live objects that will be traced and moved. With a much larger nursery, the lower garbage collection overhead offsets the increase in execution time for the rest of the application (i.e. Non-GC) due to

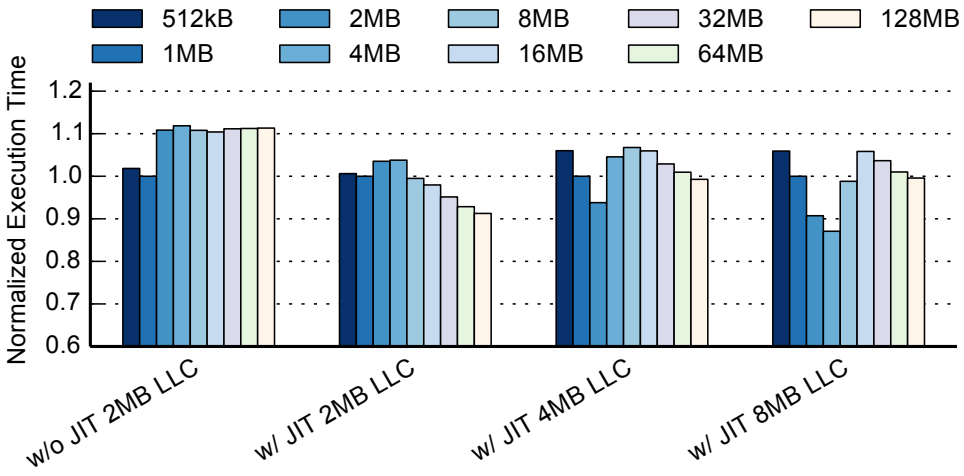


Figure 3.9: Nursery sweep for PyPy with different runtime configurations and last-level cache sizes.

the poor cache performance. These results suggest that nursery sizing purely for good cache performance may not always result in good overall performance.

The choice of runtime configuration and the amount of cache space also affect the performance trade-off. Figure 3.9 shows the average execution time of four configurations for the different nursery sizes normalized to the 1MB nursery case. The first two configurations use a LLC size of 2MB without and with JIT. The next two configurations use PyPy with JIT with different LLC sizes (8MB is the on-chip shared L3 size for Skylake processors).

For PyPy without JIT, the average trend suggests that sizing the nursery size for cache performance is beneficial for overall performance. As shown in Figure 3.10, this is due to the fact that the contribution of garbage collection to the overall execution time is small. By optimizing the program execution with JIT, the contribution of garbage collection increases by 4.6x from 3% to 14% on average. As a result, sizing the nursery only for cache performance can hurt the overall performance due to the larger relative garbage collection overhead. Note that although the relative overhead of garbage col-

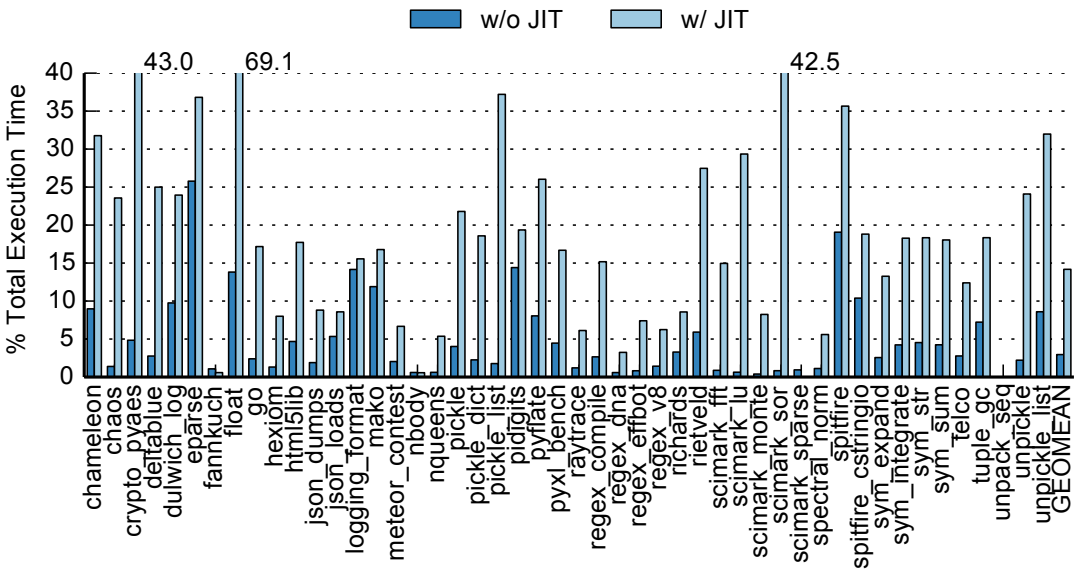


Figure 3.10: Garbage collection time as a percent of program execution time.

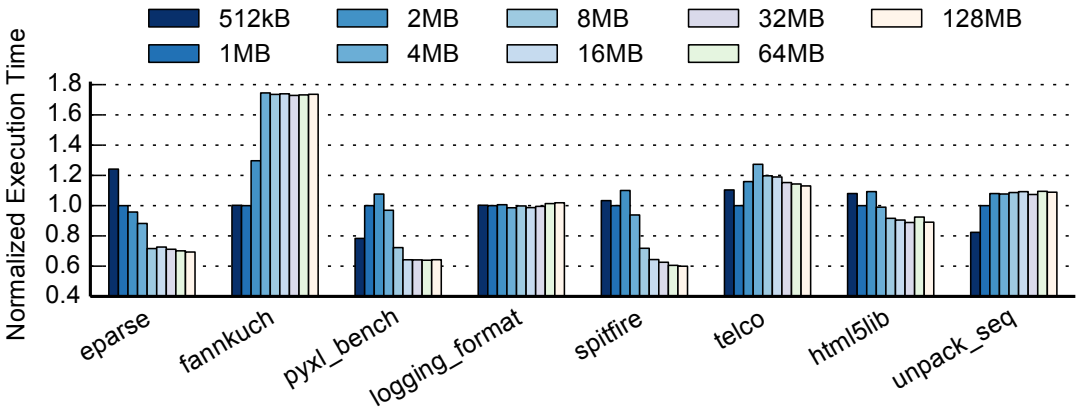


Figure 3.11: Nursery sweep for individual benchmarks running on PyPy with JIT.

lection increases significantly when using JIT, the absolute garbage collection time only increases by 5.4% on average.

Figure 3.11 and Figure 3.12 show the sweeps for individual benchmarks for PyPy with and without JIT respectively. The results suggest that one sizing policy is not good for all the benchmarks and the optimal nursery size also depends on the runtime configuration being used (i.e. with or without JIT). Some applications like `eparse` which have a large garbage collection contribution for both PyPy with and without JIT

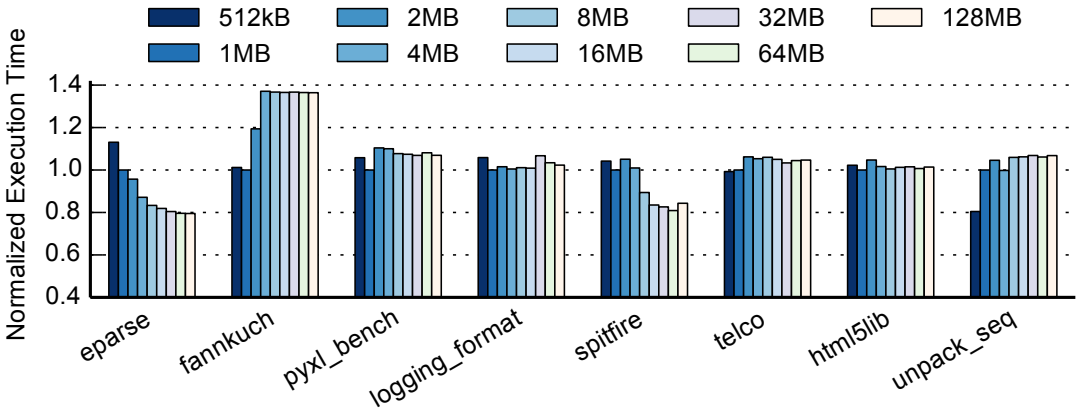


Figure 3.12: Nursery sweep for individual benchmarks running on PyPy without JIT.

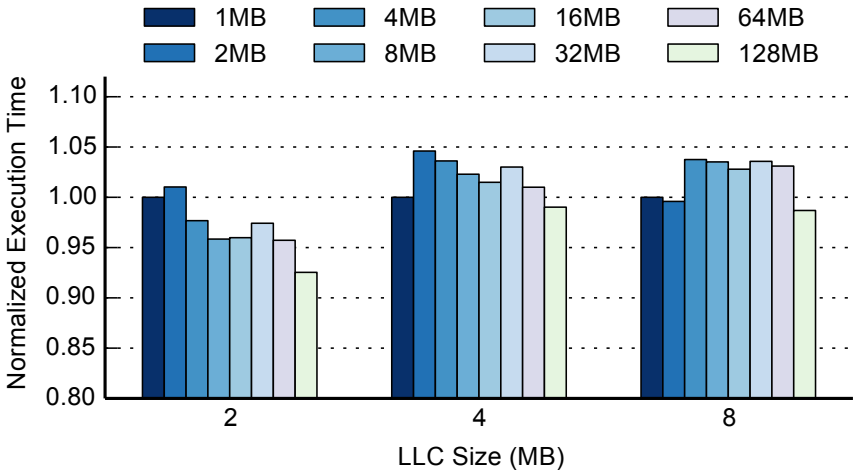


Figure 3.13: Nursery sweep for V8 with different last-level cache sizes.

will benefit from a large nursery. Other applications like `fannkuch` which have low garbage collection contribution for both PyPy with and without JIT will benefit from a nursery sized for good cache performance. There are also some applications like `pyxl_bench` which may benefit from a large nursery size for PyPy with JIT and a small nursery size for PyPy without JIT due to the large change in the garbage collection contribution as a result of running JIT.

Figure 3.9 also shows that the cache size affects the trade-off. With a larger cache, a larger nursery can fit in the cache and the better cache performance contributes to better overall performance. Figure 3.13 shows that this trend also exists for V8 suggesting that

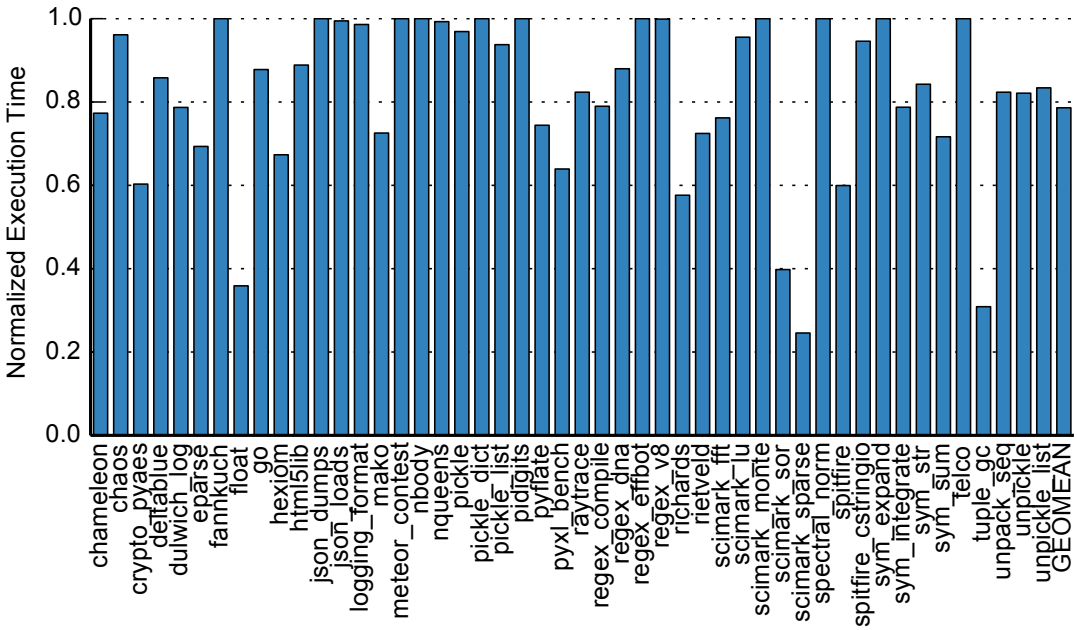


Figure 3.14: Normalized execution time for best nursery size per benchmark.

this trade-off will be important to explore for implementations beyond PyPy.

Figure 3.14 shows that by choosing the best nursery size for each application, the normalized execution time can drop by an average 21.4% over the baseline that sets the nursery size to be half of the cache size (i.e. 1MB nursery for 2MB cache). In comparison, simply increasing the nursery to the maximum size for all applications would only result in 9.8% average execution time reduction. These results further suggest that nursery sizing should be done considering cache performance, runtime configuration, and application characteristics.

3.5 Study on JIT Thresholds

JIT compilation can improve performance of an interpreted program by compiling bytecode to machine code. However, there are cases where using JIT is actually worse than using an interpreter. This is mostly due to the overhead of compilation and deoptimiza-

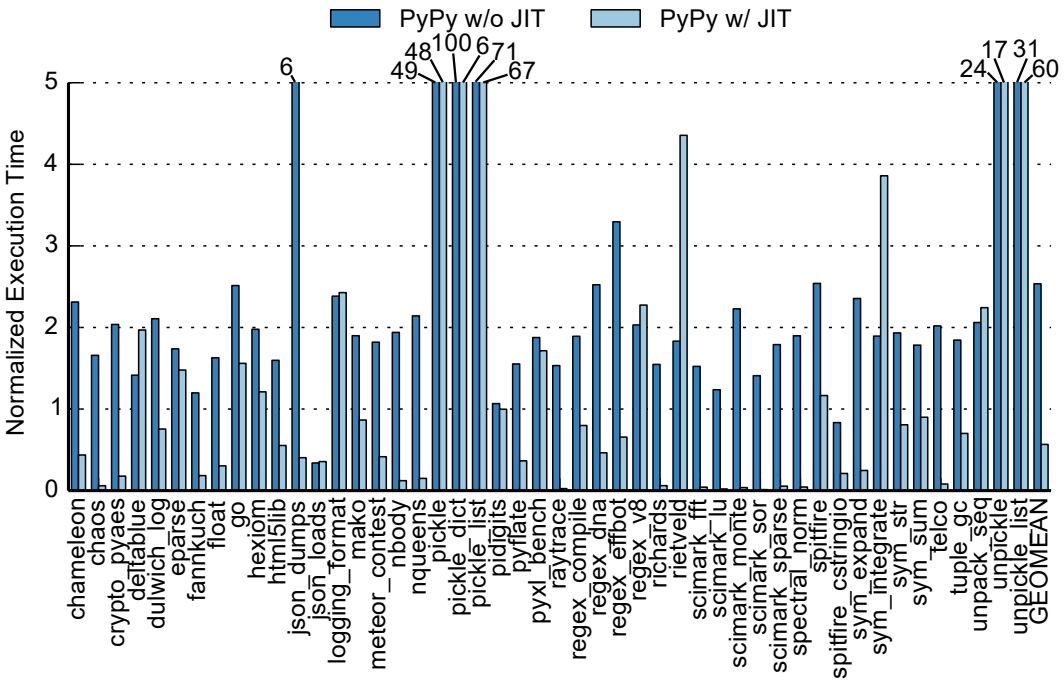


Figure 3.15: The execution time of PyPy normalized to CPython.

tion which needs to be amortized over time by the more efficient machine code. This section discusses a study which shows that one way to better amortize the JIT overhead is to more intelligently choose when to JIT.

Figure 3.15 compares the execution time of PyPy (with and without JIT) to CPython. The figure shows that JIT compilation in PyPy can often significantly speed up applications. On average, PyPy with JIT reduces program execution time by 43.3% and up to 98.8%. Yet, the speed-up heavily depends on application characteristics. There are some programs, such as `rietveld` that may execute faster without JIT. In addition, the PyPy bytecode interpreter runs on average 2.54x slower than the CPython interpreter, so the JIT optimizations must also improve performance enough to compensate for the slowdown.

Figure 3.16 shows the breakdown of a program execution time based on the JIT

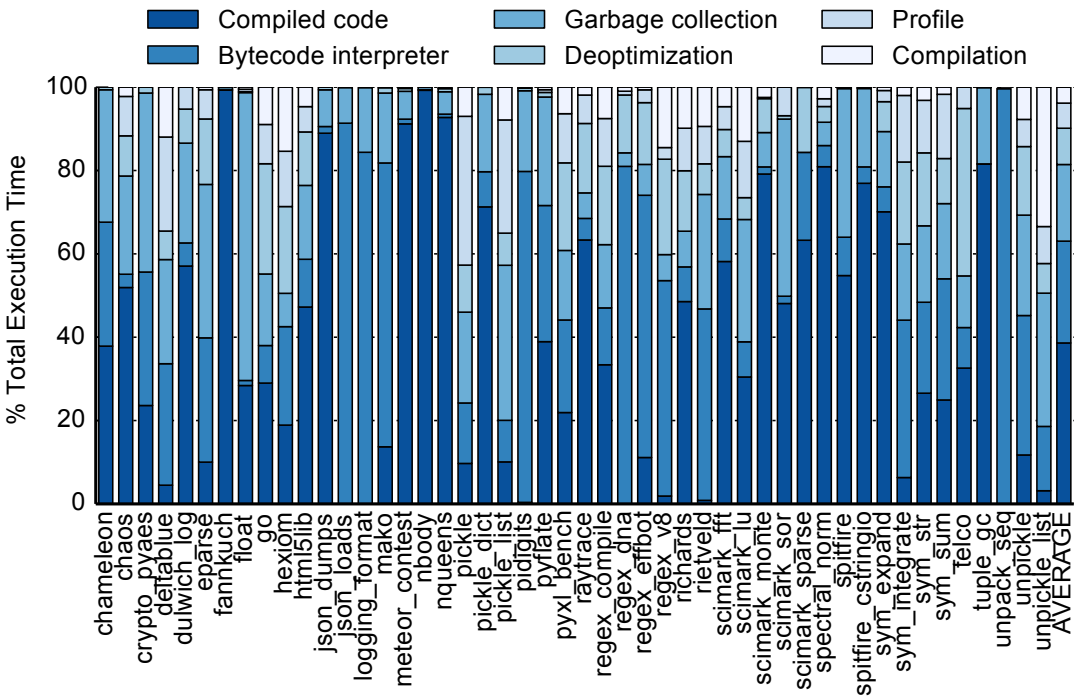


Figure 3.16: Breakdown of the execution time by JIT phases.

phases. A Pin tool was used with ZSim to identify functions related to the different phases and collect stats for each phase separately. For most applications, the compiled code execution and the bytecode interpreter account for most of execution time. Even though JIT compilation is slow, its contribution to the overall execution time is low because the cost is amortized over the entire program execution. The phases that are related to running JIT (i.e. deoptimization, profile, compilation) account for an average of 18.5% and up to 54.0% of the execution time. In addition, benchmarks still spend an average of 24.4% and up to all of their execution time in the bytecode interpreter. The breakdown suggests that there is opportunity to use JIT to optimize more code.

In order to test out the potential benefits of optimizing more code with JIT, a simple experiment was run to try out different JIT thresholds for each program and compare the improvement in the execution time. The JIT compiler optimizes code when a loop count

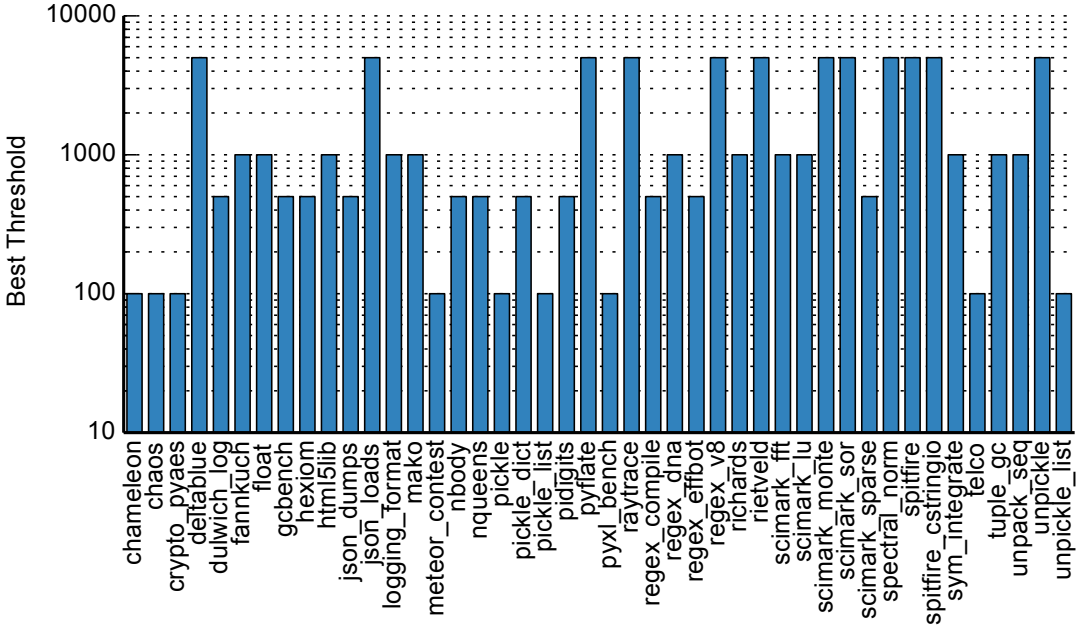


Figure 3.17: JIT threshold which results in the best performance.

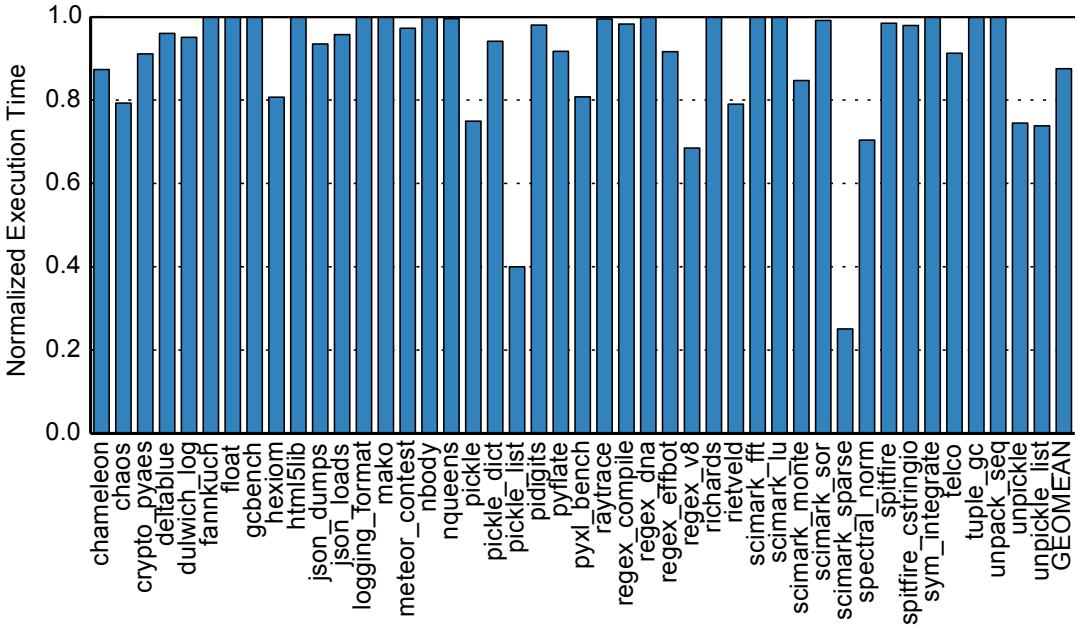


Figure 3.18: Execution time for best JIT threshold normalized to execution time when threshold is 1000.

reaches the JIT threshold. Thresholds of 100, 500, 1000, 5000 were used. With a threshold of 100, JIT compilation runs earlier in the execution and there is more opportunity to amortize the cost of running the compiler. In contrast, with a loop threshold of 5000, compilation is only done for the most frequently running loops. Figure 3.17 shows the threshold that resulted in the best execution time for each benchmark. As shown in the figure, there is no best threshold for all of the applications. Figure 3.18 shows the corresponding execution times at the best threshold for the various benchmarks normalized to the case of running with a threshold of 1000 (the default threshold for PyPy is 1039). If a good threshold is chosen, the results indicate that there would be an average 12% improvement in execution time and improvements can be as much as 75%. The results suggest that more intelligently choosing when to JIT based on the trade-offs can be one way to improve performance of programs with JIT.

It is worth noting that there would probably be more room for improvement if more loop thresholds were used in the experiments. In addition, the same loop threshold is applied to all of the loops in the program. Using a more fine grained approach and selecting loop thresholds on a per-loop basis may also result in more improvements. Determining how to best choose loop threshold remains an open question. There may be an opportunity to use hardware counters to inform these decisions.

CHAPTER 4

CACHE-AWARE OPTIMIZATIONS FOR SINGLE-APPLICATION MEMORY MANAGEMENT

4.1 Overview

As discussed in the previous chapter, there is a fundamental trade-off between garbage collection overhead and cache performance of dynamically allocated objects in dynamic languages. On one hand, frequent garbage collection operations lead to significant performance overhead. On the other hand, less frequent garbage collection requires more memory space to keep dynamically-allocated objects over a longer garbage collection period. Such memory allocation using a large memory region increases the working set size and can significantly degrade the cache performance through loading of newly allocated objects from memory and increased cache pollution and write-backs. The impact on cache performance is particularly significant if the memory space for frequent allocations does not fit into on-chip caches. The baseline heuristic for PyPy chooses the nursery size to be half of the last-level cache size for young objects in order to balance cache pressure and garbage collection overhead.

In this chapter, hardware support and software optimizations are proposed for reducing memory management overhead in dynamic languages. First, cache performance is optimized for newly-allocated objects by directly placing them in on-chip caches without reading the corresponding locations from off-chip memory. Because newly-allocated memory locations need to be initialized anyways, there is no need to read their previous values from memory. Next, cache pollution and additional write-backs caused by newly-allocated objects is reduced using a partial tracing strategy that determines dead cache lines that do not need to be kept or written back.

These optimizations remove the main obstacles in using a large memory region for new memory allocations and enable running garbage collection far less frequently than what is considered to be optimal. In this way, both overhead for both garbage collection and frequent memory allocations can be reduced. The experimental results show that this co-optimization of garbage collection and memory allocation can achieve significant improvements in execution time; 22% improvement on average and up to 68% improvement for PyPy [15], a popular implementation for Python, and 17% improvement on average and up to 63% improvement for V8 [52] running JavaScript.

The high-level idea of directly placing newly-allocated memory locations into on-chip caches without off-chip accesses is known as *cache installation* and has been studied previously in the context of C and C++ [83, 63, 120]. However, the cache installation itself only leads to small performance improvements for C and C++ because they only optimize relatively infrequent memory allocations. Optimizing initialization of newly-allocated locations is far more important for dynamic languages with frequent memory allocations and, more importantly, can be used to enable less frequent garbage collection to significantly reduce overhead of managed memory. Co-optimization of garbage collection and memory allocation is essential in obtaining the performance improvements that are reported.

Moreover, previous cache installation mechanisms for C and C++ are not well-suited for dynamic languages. The previous mechanisms are designed for memory allocations for objects larger than a cache line (64 bytes). Yet, dynamic languages often allocate objects smaller than a cache line. In addition, previous designs are not built to reduce unnecessary cache pollution and write-backs. The proposed invalid memory tracking mechanism is designed to enable cache installation even for small objects by leveraging the sequential memory allocators widely used with generational garbage collectors. The

same tracking mechanism can also be used to reduce unnecessary cache pollution and write-backs with software assistance.

The following summarizes the main contributions in this chapter:

1. A detailed study of the sources of cache misses in dynamic languages is provided as well as insight into the challenges in eliminating those misses.
2. A new invalid memory region tracking mechanism is presented that allows cache installation even for small objects commonly used in dynamic languages as well as write-back reduction and pollution control.
3. A partial tracing algorithm is described that can be run to identify invalid cache lines with lower overhead compared to full garbage collection overhead.
4. The results show that the proposed optimizations can lead to significant performance improvements in the state-of-the-art implementations of two widely used dynamic languages, Python and JavaScript, using a wide range of applications.

The rest of this chapter is organized as follows. Section 4.2 describes a study on cache performance in dynamic languages. Section 4.3 describes the invalid memory tracking mechanism and Section 4.4 describes the partial tracing algorithm. Section 4.5 evaluates the proposed memory management optimizations. Section 4.6 discusses some other approaches for improving performance and provides evaluation results for those approaches.

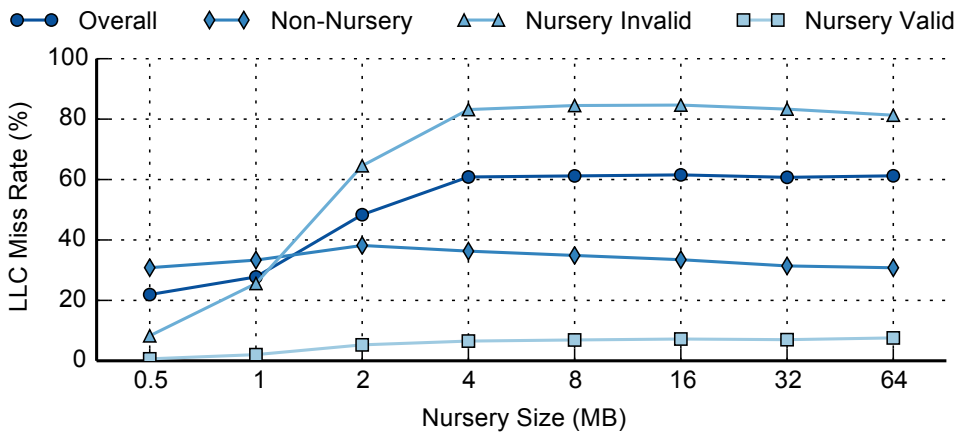


Figure 4.1: Breakdown of LLC miss rate as a function of nursery size.

4.2 Cache Performance Study

4.2.1 Miss-rate Breakdown

Figure 4.1, which shows the last-level cache (LLC) miss-rate breakdown as a function of the nursery size, gives insight into why there is poor cache performance for large nursery sizes. As the nursery size increases, the overall LLC miss-rate increases significantly, mainly because initial accesses to newly-allocated nursery locations miss in the cache (shown as Nursery Invalid). The results suggest that it is important to reduce cache misses for initial nursery accesses in order to enable using a larger nursery with low garbage collection overhead.

A larger nursery also puts more pressure on caches and increases cache misses for non-nursery accesses (Non-Nursery) or nursery accesses after the initialization (Nursery Valid). While not shown in the figure, the number of write-backs can also increase significantly for a large nursery.

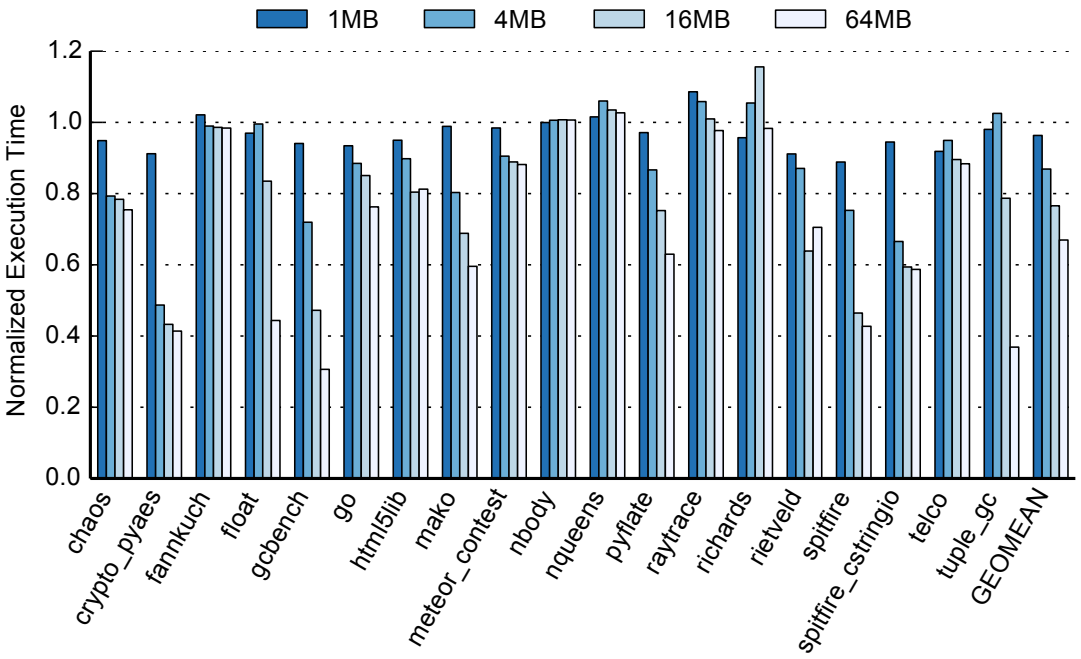


Figure 4.2: Execution time as a function of nursery size when new objects are directly installed in caches.

4.2.2 Cache Installation of Invalid Memory

The initial accesses to newly-allocated objects represent a series of stores to initialize the objects. These accesses retrieve a cache line from memory and simply overwrite it with a new value. Therefore, there is no need to read these invalid (uninitialized or unallocated) memory locations. Instead, a store miss to an invalid memory region can be serviced by directly placing an arbitrary value (such as zero) into the cache block without reading memory if all memory locations mapped to the cache block are invalid. This technique is often called *cache installation*.

The cache installation of invalid memory regions not only reduces unnecessary memory accesses, but also enables using large nursery sizes to reduce garbage collection overhead. Figure 4.2 shows the normalized execution time as a function of the nursery size when all initial accesses to invalid (uninitialized or unallocated) nursery locations

are somehow identified and directly placed into an on-chip cache without off-chip accesses. Unlike the baseline PyPy, large nursery sizes combined with cache installation can significantly improve the performance. On average, the 64MB nursery with cache installation outperforms the baseline (1MB nursery) by 28.7%. `Gcbench`, which has high garbage collection overhead runs 69.4% faster with a 64MB nursery. `Nqueens`, which originally shows a 2.7x slowdown for the 64MB nursery in the baseline, only shows a 2.7% slowdown with the ideal cache installation. The results suggest that the co-optimization of the nursery size and the initial cache misses for the nursery has a potential for significant performance improvements.

While cache installation helps reduce read memory traffic, it does nothing to reduce write-backs or cache pollution. The newly installed cache lines cause existing cache lines to be evicted. On average, using a 64MB nursery results in 3.74x more write-backs compared to a 1MB nursery in PyPy. Additional write-backs usually do not directly affect performance as they happen in the background. Yet, it can be a significant concern for bandwidth-limited systems.

4.2.3 Memory Allocation Size

In order to use the cache installation, a full cache line must be guaranteed to be uninitialized. In static languages such as C and C++, memory allocations are often larger than a cache line, and existing cache installation mechanisms either explicitly capture memory allocations larger than a cache line using a table [83] or capture a series of stores that overwrite an entire cache line over a short period [63, 120].

For dynamic languages, however, memory allocation sizes are often small. Figure 4.3 shows the cumulative distribution of the allocation size (in bytes) for PyPy and

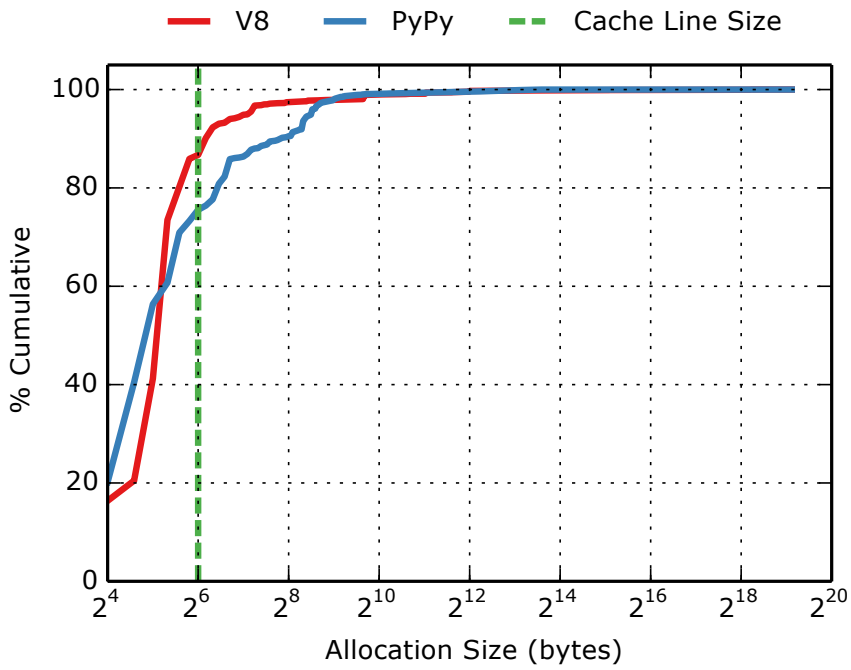


Figure 4.3: The cumulative distribution of the memory allocation size for PyPy and V8.

V8. The distribution is averaged across all benchmarks (49 for Python and 37 for V8). The vertical line represents the typical cache line size of 64 bytes. For PyPy, more than 73.2% of allocated objects are smaller than a cache line. For V8, 85.9% of allocations are smaller than a cache line. The results suggest that the cache installation for dynamic languages must be able to effectively handle small object allocations.

4.3 Invalid Memory Region Tracking (IMRT)

Invalid memory regions can be defined as memory locations that are either unallocated or uninitialized. When an object is created, memory is allocated but remains uninitialized. Upon calling the constructor, data is written to the object for initialization. The initialized object can thereafter be used.

To use cache installation and not load data from memory, a whole cache line needs to be guaranteed to be from an invalid memory region. Invalid memory regions do not contain useful data and can hold any value without affecting the functionality of the program. Most processor caches associate multiple words with a tag in a single cache line (e.g. 64-bytes). In a typical write-allocate cache design, cache lines which are written to must be first loaded from memory because neighboring words in the same cache line may be later read. If the whole cache line is from an invalid memory region, reading from memory unnecessarily uses memory bandwidth and increases latency.

Explicitly tracking memory allocations is not enough for cache installation in dynamic languages because most allocations are smaller than a cache line. Even if a memory allocation is identified, a cache line should still be read from off-chip memory because some words in the cache line could still be valid.

Instead, two features of the typical memory management with generational garbage collection can be used to enable allocating of full cache lines:

1. The nursery is fully unallocated after each garbage collection.
2. The allocation is done in a sequential fashion.

These features ensure that installation does not affect functional correctness by guaranteeing that memory locations above a newly-allocated object are always invalid. If an object is smaller than a cache line, a full cache line is installed into the cache. In the program, however, the object is still allocated in memory at a byte granularity.

To limit the cache pollution, objects should be installed in the cache when they are needed. Premature cache installation may unnecessarily evict useful cache lines. A small hardware table is proposed to track invalid memory regions in the nursery at the

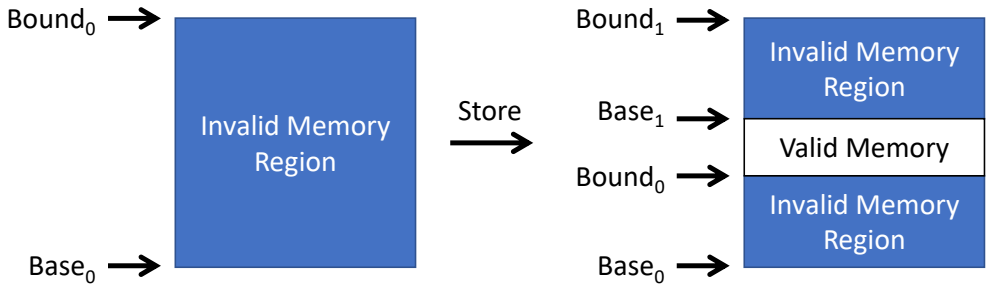


Figure 4.4: An example of splitting an invalid memory region into two after a memory allocation.

cache-line granularity (typically, 64 bytes). Then, invalid memory locations are directly installed in the cache on the first write to the corresponding cache line.

4.3.1 Tracking Table for Address Ranges

In order to track invalid regions, the software must first tell hardware where the initial invalid memory region is. For the nursery, this only needs to be done every garbage collection cycle and when it is first allocated. The software provides the base and bound as full addresses. Hardware keeps cache-line aligned base and bound addresses using a table, and only installs cache lines that are fully covered by the invalid memory region.

Each entry of the tracking table stores a memory range (cache-line aligned base and bound addresses) for one invalid memory region. When software initially provides the base and bound, an entry is added to the table and other entries that fall within the base and bound of the added entry are cleared. The hardware table monitors stores within the invalid region, and updates its entries to maintain the invariant that every range in the table is guaranteed to represent an invalid memory region. Figure 4.4 shows how the invariant is maintained by splitting one entry into two when there is a store in the middle of an invalid region.

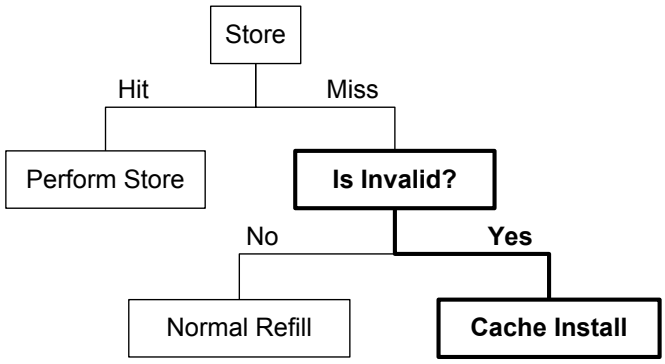


Figure 4.5: Cache installation decision on a store.

4.3.2 Handling Tracking Table Evictions

Because a hardware table has a limited size, it may eventually run out of space. In that case, one of the memory ranges need to be evicted. To limit loss of information, an eviction policy that evicts the smallest range should be used.

Even if an entry is evicted, the tracking will still be correct in the sense that there is no false detection of invalid regions. Some stores to invalid memory regions may not be detected and handled normally without cache installation, but the program execution will still be correct.

4.3.3 Cache Installation

Figure 4.5 shows how the cache installation using invalid memory region tracking can be done at the LLC (last-level cache) level. On a store miss, the IMRT will be checked. If the requested block is from an invalid memory region, then the cache line will be installed instead of being requested from memory. Placing the tracking table at the LLC removes the need for changing the cache coherence protocol and also enables using the table to identify invalid cache lines for reducing unnecessary cache evictions and write-backs.

Yet, placing the tracking at the LLC-level level introduces a challenge; tracking needs to be performed using physical addresses instead of virtual addresses. This requires OS support for setting the initial invalid region. When the user-level software performs a system call with an invalid memory region in virtual addresses, the OS translates them to physical address regions and sets the tracking table. As a contiguous virtual address range may not map to a contiguous physical address range, the OS may need to set multiple physical address ranges in the tracker.

Since the tracker is managed by the OS and uses physical addresses, multiple processes and threads can share the hardware by invoking the relevant system calls to set the entries. The same hardware can also be used by the OS when mapping and unmapping pages before, during, and after program execution, to eliminate the need to read in new pages to cache or write-back unused pages from cache.

4.3.4 Implementation Details

The tracking hardware needs interfaces for both an OS and a memory allocator. These interfaces can be implemented in many ways. Here, a design using memory-mapped interfaces is shown. For the OS interface, a unique hardwired physical address is assigned to the tracking table so that the OS can configure tracking hardware.

Table 4.1 summarizes the software interfaces necessary for the tracking hardware. When user-level software wants to set an invalid memory range, it makes a system call to set the invalid range. Since the tracking will be done automatically in the IMRT, the user-level software only needs to make system calls when setting ranges. The invalid memory range can be cleared by setting a valid range using another system call.

Table 4.1: Software interfaces for the tracking hardware.

Invoked By	**ISA Instruction**	**Operation**
OS	store <base> [<track_hw_addr> + set_inv_base_offset]	Sets the base address for an invalid region in a temporary register.
	store <bound> [<track_hw_addr> + set_inv_bound_offset]	Copies the base (held in a temporary register) and bound for an invalid region to the tracking table, evicting an entry if necessary.
	store <base> [<track_hw_addr> + set_val_base_offset]	Set the base address for a valid region in a temporary register.
	store <bound> [<track_hw_addr> + set_val_bound_offset]	Uses the base (held in a temporary register) and bound for a valid region to update the tracking table, splitting an entry if necessary.
Program	syscall <inv><ptr_to_array_base_bound>	The OS will iterate through the array, translate the virtual addresses to physical addresses, and will set the invalid regions in the track table.
	syscall <val><ptr_to_array_base_bound>	The OS will iterate through the array, translate the virtual addresses to physical addresses, and will validate the regions in the track table.

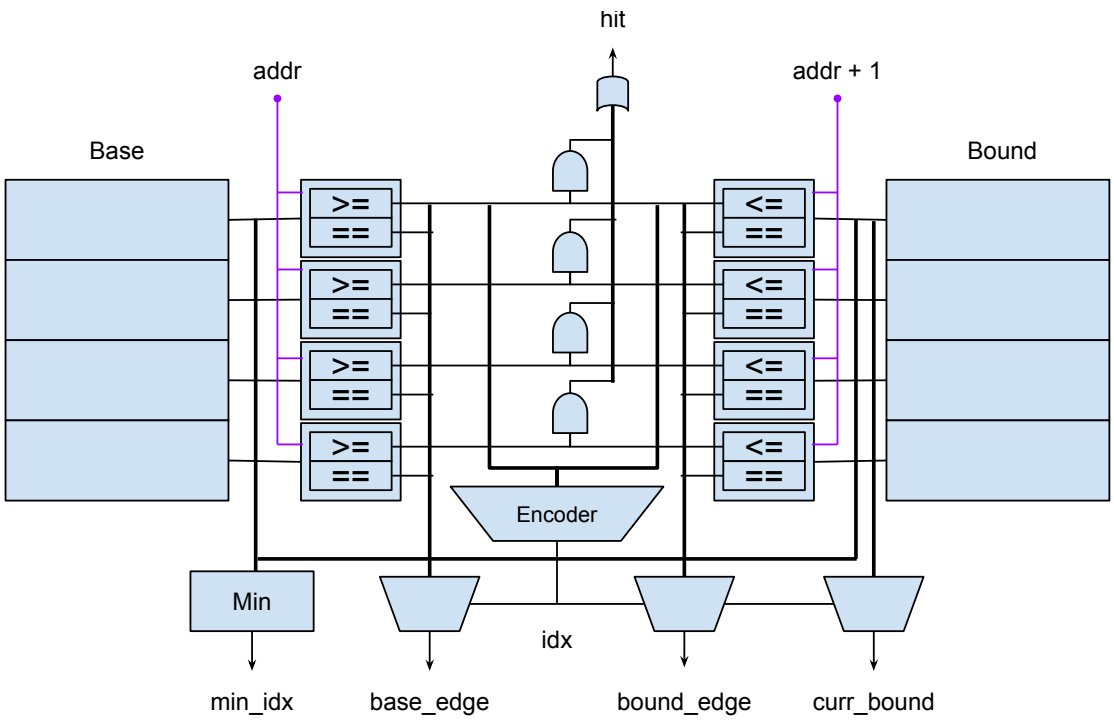


Figure 4.6: Tracking hardware data path.

Table 4.2: Pseudo-code for updating base and bound fields.

Base Update	**Bound Update**
`if (hit && base_edge):` `  base[idx] = addr + 1`	`if (hit && !base_edge):` `  bound[idx] = addr`

`if (hit && !(base_edge`
`  || bound_edge)):`
`  base[min_idx] = addr + 1`
	`if (hit && !(base_edge`
	`  || bound_edge)):`
	`  bound[min_idx] = curr_bound`

Figure 4.6 shows the data path of the tracking hardware with four entries. The input to the datapath are two addresses, `addr` and `addr+1`, at the cache-line granularity; for 64-byte cache lines, `addr` is obtained by removing the 6 LSBs of the memory address for a store. The output is a bit indicating a hit. Note that the bound value is exclusive, so the `addr` needs to be less than the bound for a hit.

The update hardware is not drawn, but follows the logic shown in Table 4.2. If the `addr` is at the `base_edge`, only the base at the current index needs to be updated by

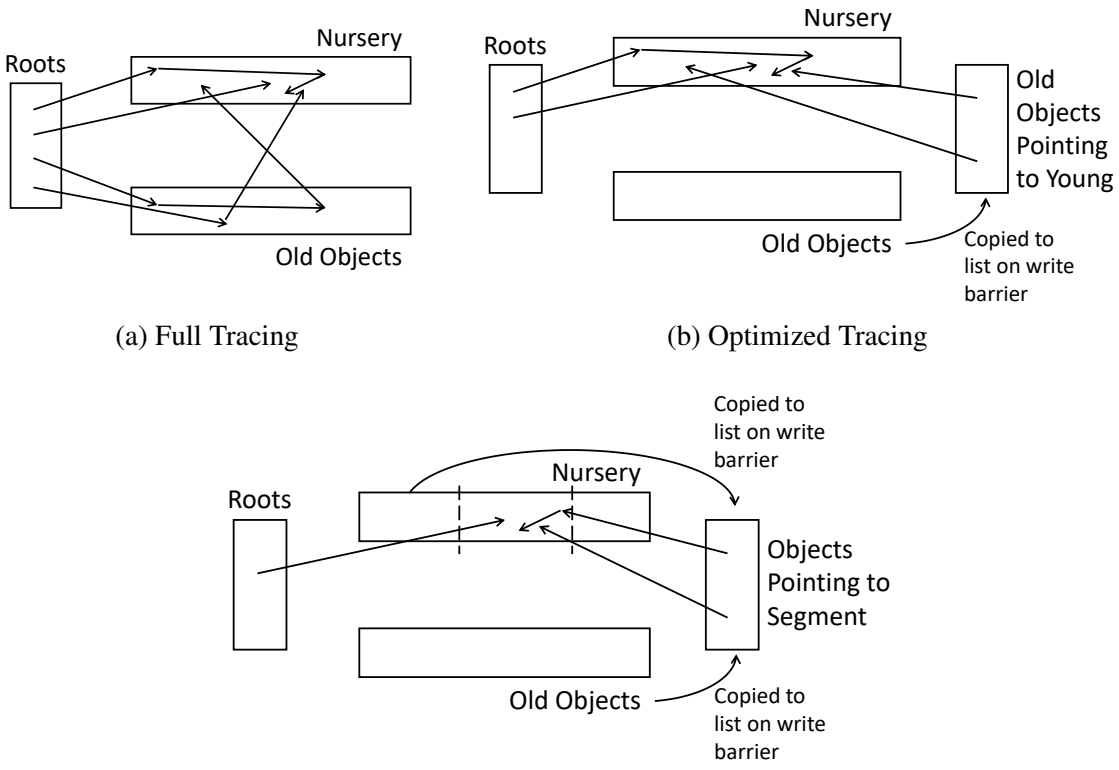
incrementing `addr` by 1. If the `addr` is at the `bound_edge`, only the bound needs to be set to be `addr`, since the bound is exclusive. Finally, if the `addr` is not at an edge, then the entry must be split by setting the existing entry bound to be the `addr` and the new entry to have the range of `addr+1` to the current bound. The hardware is more complex compared to traditional caches because each entry needs to handle an arbitrary memory range. Yet, the number of table entries is quite small compared to caches.

4.4 Partial Tracing

Although cache installation reduces memory reads, it does not address cache pollution and increased memory writes from a large nursery. Installed cache blocks will still evict other cache lines to make room and need to be written back to memory when evicted. The problem is particularly severe if the nursery is larger than the last-level cache when most cache lines in the nursery will be evicted and written-back to memory before they can be used again after garbage collection.

However, many of these cache evictions and write-backs may be unnecessary. Following the generational hypothesis that most objects die young, there is a strong likelihood that many objects are dead before they are evicted from the cache. If the dead objects can be identified, then both unnecessary write-backs of dead objects and unnecessary eviction of valid cache lines can be avoided.

For this purpose, a software algorithm is introduced, called partial tracing, that identifies dead objects in a subset of a nursery with the goal to optimize cache replacements and write-backs.



(a) Full Tracing

(b) Optimized Tracing

(c) Partial Tracing

Figure 4.7: Graphical depictions of how tracing works. Arrows represents pointers that are traversed.

4.4.1 Partial Tracing Algorithm

The tracing algorithm used during the nursery garbage collection can be adapted to determine live objects for a segment of the nursery that is most likely to be in the cache. The nursery is divided into segments that can fit in the cache. During execution, objects are sequentially allocated in the nursery as needed. Once a full segment worth of objects has been allocated, tracing is run to determine which objects are live only for that segment. This information is communicated to the hardware and used for cache replacement decisions and removing unnecessary write-backs.

One challenge in performing tracing to identify live objects for only a segment of the nursery is that pointers for all live objects need to be followed as shown in Figure 4.7(a).

Not only is the computation unnecessary but may also pollute the cache by accessing all of the live objects. As shown in Figure 4.7(b), generational garbage collectors solve a similar problem using a write barrier to limit tracing of old objects to only those old objects that contain pointers to young objects in the nursery. The write barrier works by issuing a callback function when a write to any pointer in the old object occurs. The callback function adds the old objects to a list which is then used as a starting point for tracing instead of having to perform complete tracing through the program roots.

As shown in Figure 4.7(c), write barriers can be extended to young objects in previous segments within a nursery. When performing tracing on a segment to find the live objects in that segment, a write barrier can be added to the pointers of those live objects. If a pointer in any young object in a previous segment is written to, the write barrier invokes a callback to add those objects to a trace list. The callback for old objects is also modified to only add objects that point to the current segment rather than the whole nursery. The list containing old objects and young objects from previous segments can be used to perform tracing for only the segment in an efficient manner.

Decoupling tracing from the full garbage collection also opens the door for running a part of the garbage collection, namely tracing, concurrently with the main application. As discussed before, the copying garbage collectors need to stall the main application in order to copy objects and update pointers. However, partial tracing, which only identifies live objects, does not need to pause the application and can be performed concurrently without incurring performance overhead if an extra core is available.

4.4.2 Identifying Dead Cache Lines

Partial tracing returns a list of live objects in the recently-allocated segment of the nursery that is likely to be in the on-chip cache. Using information about their start addresses and object sizes, valid memory ranges in the nursery can be determined. By additionally using the base and the bound of the segment, invalid memory ranges in the segment can be determined. To do this, the algorithm starts by assuming the whole segment is invalid. Using the valid memory ranges, the initial invalid segment can be split into smaller invalid ranges. By iterating through all of the valid memory ranges, an accurate list of invalid memory regions can be constructed.

4.4.3 Integration with IMRT

The IMRT design that was described in the previous section can be used to track invalid regions for both unallocated/uninitialized nursery regions and dead objects after partial tracing. As shown in Table 4.1, the IMRT interface allows software to add valid and invalid regions to the table. To add the invalid regions from the partial tracing, software first sets the traced segment as an invalid region in the IMRT. Each live object from tracing is added as a valid region to break the segment into multiple invalid regions.

4.4.4 Cache Eviction and Write-backs

To reduce unnecessary cache pollution and write-backs, the IMRT is referenced on cache replacements and write-backs in addition to memory reads (for cache installation). The cache replacement policy prioritizes eviction of cache lines that belong to an invalid memory region so that live objects can stay longer in the cache for reuse. The

Table 4.3: Microarchitectural parameters for simulations.

Core	4-way OOO, 2.66GHz
L1I	32kB, 4-way, 4-cycle latency
L1D	32kB, 8-way, 4-cycle latency
L2	256kB, 8-way, 10-cycle latency
L3	2MB, 16-way, 40-cycle latency
Memory	Micron DDR3-1333

memory addresses from replacement candidates are looked up in the IMRT on a cache miss, and invalidated if found in the IMRT. The invalid cache lines are replaced first before evicting valid cache lines. Note that the IMRT look-ups can be performed in the background over a long LLC miss latency without affecting performance. The IMRT table is also referenced on a write-back. If the address of the evicted (dirty) cache line is found in IMRT, the write-back will be eliminated.

4.5 Evaluation

4.5.1 Methodology

For the evaluation, a simulation infrastructure based on ZSim [121] was used to model cycle-level microarchitecture behaviors of an out-of-order core with a memory hierarchy comparable to modern processors. The processor core was configured to model an Intel Westmere processor. DRAMSim2 [117] was integrated with ZSim to model DDR3-1333 memory. Table 4.3 summarizes the microarchitectural parameters.

For the implementations of dynamic languages, PyPy [15] was used for Python and V8 [52] was used for JavaScript. Both Python and JavaScript are widely used in practice, with Python being used in a range of applications from web servers to scientific

Table 4.4: Summary of the designs used for evaluation.

Design	**Description**
Base	Baseline with no optimization.
IMRT	Cache installation with a 256-entry tracking table.
IMRT+PTO	Cache installation with a 256-entry tracking table along with partial tracing optimization.
IMRT+CPTO	Cache installation with a 256-entry tracking table along with concurrent partial tracing optimization

computing and JavaScript being primarily used for web applications. PyPy and V8 represent the state-of-art implementations for Python and JavaScript. Both use just-in-time compilation to achieve good performance and use a variation of generational garbage collection.

For benchmarks, a wide array of applications were used to get a representative sample of real-world applications. For Python, benchmarks from the official Python performance benchmark suite [109] and benchmarks from the PyPy benchmark suite were used. For JavaScript, JetStream [17] benchmarks were used. In total, experiments were run with 49 benchmarks for Python and 37 benchmarks for JavaScript. For Python, the benchmark was warmed up by running it 2 times and then run 3 times for evaluation. For JavaScript, the benchmark was run 3 times for evaluation.

The design points shown in Table 4.4 were evaluated. The baseline (Base) represents the case without any optimization. IMRT represents the proposed invalid memory region tracking mechanism where only cache installation is enabled. IMRT+PTO represents the case where both cache installation and partial tracing are enabled. Because the partial tracing optimization requires a significant re-write of a garbage collector, partial tracing was implemented only for PyPy, but not for V8. IMRT+CPTO represents the case where partial tracing is run concurrently with the normal program with cache installation. Note that there is no actual implementation of concurrent tracing but it is

Table 4.5: The coverage of a limited-size IMRT table compared to the infinite size table.

	4	**8**	**16**	**32**	**64**	**128**	**256**
PyPy	77.6%	84.7%	89.4%	92.9%	95.2%	96.1%	97.1%
V8	79.5%	79.7%	88.5%	94.7%	96.5%	97.9%	98.8%

modeled by subtracting the computation overhead of running tracing from the execution. The cache pollution from tracing is still included. For the baseline, the default 1MB nursery was used for PyPy and V8. For the optimizations, a 64MB nursery was used for PyPy and 128MB nursery for V8.

4.5.2 Tracking Table Size

This study evaluates how many invalid memory addresses the tracking table can capture with a limited storage size compared to the ideal case with unlimited storage. For this purpose, the percentage is computed of memory reads to invalid memory regions that are captured by a given IMRT table size for cache installation compared to the total number of reads to invalid memory regions.

The results shown in Table 4.5 suggest that a small tracking table can capture nearly all of the invalid memory ranges. Since the memory allocation of objects happens sequentially and initialization also happens mostly sequentially, most memory allocations only need to update the base address of an existing entry rather than creating an entry of a new memory range. An update at the boundary does not split an entry into multiple ranges, so no additional space is required in the tracking table.

A first-order evaluation shows that a 256-entry table would have minimal area and power overhead. The first-order evaluation was performed using CACTI [98], an integrated model for cache and memory access time, cycle time, area, leakage, and dynamic

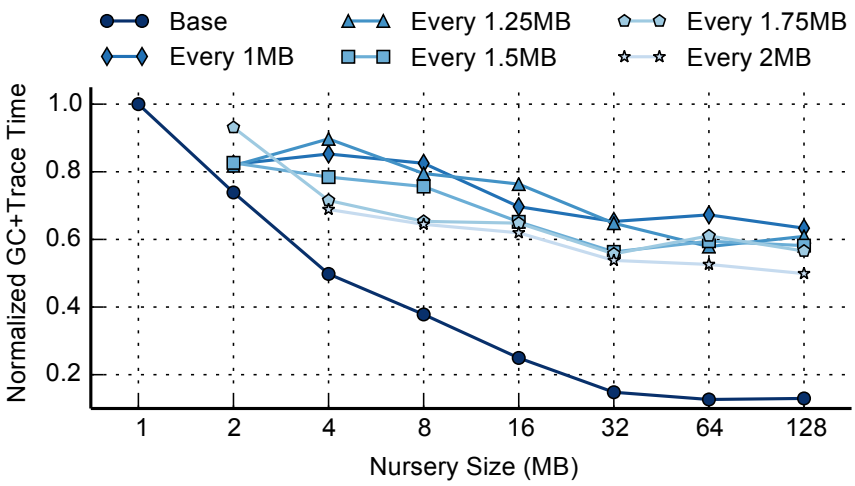


Figure 4.8: Normalized execution time for garbage collection and partial tracing.

power. The main overhead of the table comes from the memory required to store the base and bound addresses (i.e. a pair of 64-bit addresses). This would equate to 4kB of memory and could be reduced by compression of the addresses. If the IMRT is modeled as a fully associative 4kB cache, it uses $0.038mm^2$ area and has $6mW$ leakage power on a $22nm$ node. For comparison, the 32kB L1 data cache in the processor would use $0.408mm^2$ area and would have $15mW$ leakage power on a $22nm$ node.

4.5.3 Partial Tracing Period

Here, the overhead of partial tracing is studied as nursery sizes and partial tracing frequencies are varied. The execution time of the garbage collection with and without partial tracing at each nursery size is normalized to the baseline garbage collection (no partial tracing) with a 1MB nursery. For reference, garbage collection with a 1MB nursery is on average 16.0% of the total program execution time. Figure 4.8 shows how the baseline garbage collection overhead (Base) decreases as the nursery size increases. With a 64MB nursery, garbage collection overhead is reduced by an average of 87.4%.

Adding partial tracing on top of garbage collection enables cache optimizations for replacements and write-backs, but reduces the savings from a large nursery. In the figure, 'Every x-MB' indicates the case where partial tracing is performed once every x-MB in addition to the full garbage collection when the entire nursery gets full. More frequent partial tracing has a potential to more quickly identify and remove dead objects in the cache, but also has higher overhead. However, the results suggest that the performance differences among different tracing frequencies are rather small for the range of values evaluated. In the following studies, the partial tracing period of every 1.25MB is used for IMRT+PTO. This partial tracing period with a 64MB nursery has a potential to remove 42.2% of the baseline garbage collection overhead.

4.5.4 Overall Performance

This study evaluates the overall performance improvements of the proposed optimizations. The performance is presented as the execution time normalized to the baseline with a 1MB nursery. For the IMRT schemes, a table size of 256 entries is used to support both cache installation and write-back elimination. For PyPy, results for both 1MB and 64MB nursery sizes are shown in order to separately evaluate the improvements from cache installation and less-frequent garbage collection.

Figure 4.9 shows the normalized execution time of the various designs for each Python benchmark. The results show that simply increasing the nursery (Base-64MB) leads to a significant slowdown in many applications. On the other hand, cache installation with a 1MB nursery (IMRT-1MB) only reduces the execution time by 4.7% on average. This shows that cache installation by itself does not lead to significant performance improvements. Using a large (64MB) nursery with cache installation (IMRT-

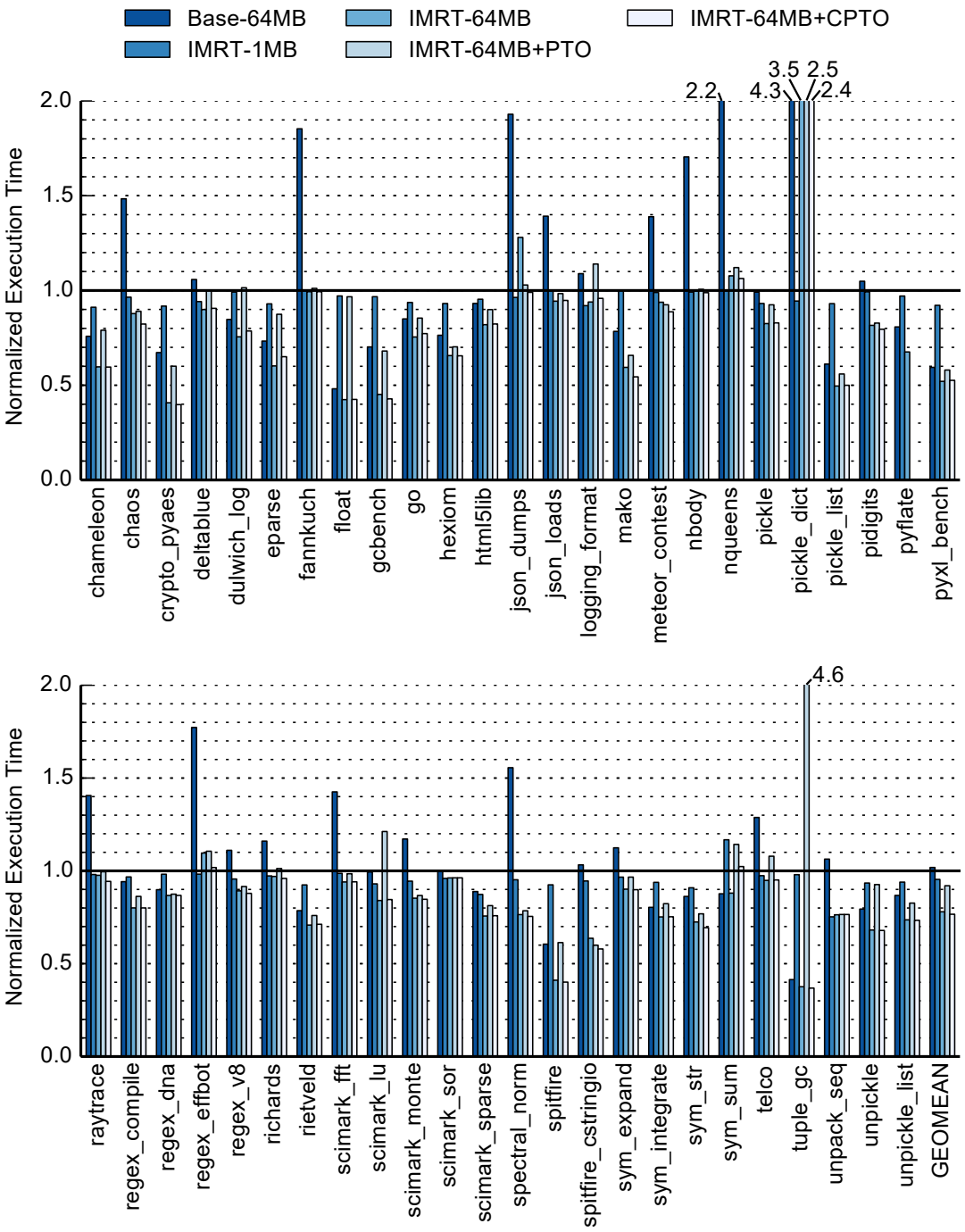


Figure 4.9: Normalized execution time for PyPy. The execution time is normalized to the baseline with a 1MB nursery.

64MB) reduces the execution time by 22% on average compared to the baseline with only a single benchmark showing a noticeable slowdown. Moreover, the execution time is reduced by over 50% for multiple benchmarks. The average improvement is within 2.0% of the possible improvement with an ideal tracker. The results show the benefits of co-optimizing garbage collection period with cache installation.

Cache installation with partial tracing (IMRT-64MB+PTO) shows performance improvements on average of 8.0%, which is less than IMRT due to the overhead of additional tracing. However, IMRT+PTO can still significantly reduce the execution for many applications. Moreover, for applications where the impact on cache performance overhead of a large nursery is particularly significant, IMRT+PTO outperforms IMRT, reducing the execution time by an additional 5.9% for `spitfire_cstringio` or 1.4% for `meteor_contest`.

If partial tracing is performed concurrently with the application (IMRT-64MB+CPTO), then the execution time is always better than IMRT, since the partial tracing improves cache performance and there is little sequential execution overhead for running it. As a result, there is an additional 1.3% average reduction in the execution time over IMRT. Running partial tracing concurrently can achieve significant performance improvement while simultaneously reducing memory accesses.

Figure 4.10 shows the normalized execution time for V8 when the IMRT hardware is used for cache installation with the 128MB nursery size. Cache installation alone reduces the average execution time by 8.2%. When combined with the larger nursery size, the average execution time is reduced by 16.8% with some benchmarks showing reductions more than 40%. The results show that the proposed hardware tracker is general enough to be applied to multiple languages and implementations beyond PyPy.

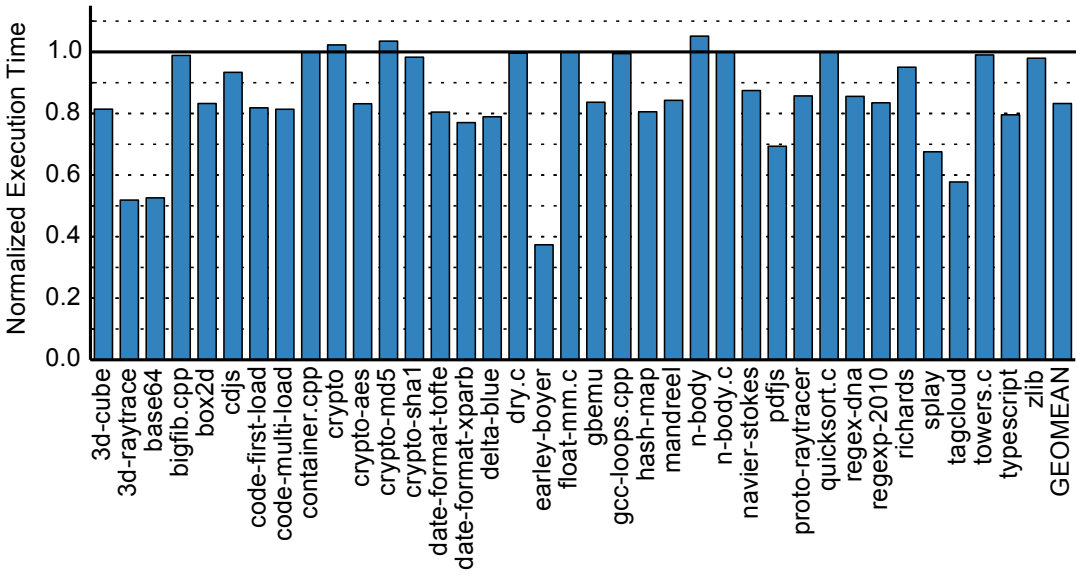


Figure 4.10: Normalized execution time for V8. The execution time is normalized to the baseline with a 1MB nursery.

Table 4.6: Breakdown of cache miss rates at the LLC.

	PyPy			**V8**	
	Base-64MB	**IMRT-64MB**	**IMRT-64MB+PTO**	**Base-128MB**	**IMRT-128MB**
Overall	60.2%	17.6%	17.4%	48.2%	21.2%
Non-Nursery	29.5%	29.5%	26.1%	34.2%	26.5%
Nursery	89.7%	9.5%	9.2%	70.6%	13.8%

4.5.5 Cache Miss-Rate Breakdown

Here, the impact of the optimizations on the last-level cache (LLC) miss-rates for different memory regions is studied. Table 4.6 shows a breakdown of the cache miss-rates for PyPy and V8 for the baseline and IMRT with the same nursery size. The miss-rates are shown for all accesses (overall), nursery accesses, and non-nursery accesses. The results show that the proposed cache installation is indeed quite effective in reducing the cache misses to the nursery. For PyPy, the results suggest that the partial tracing optimization can reduce cache pollution and slightly improve the cache miss rate for both

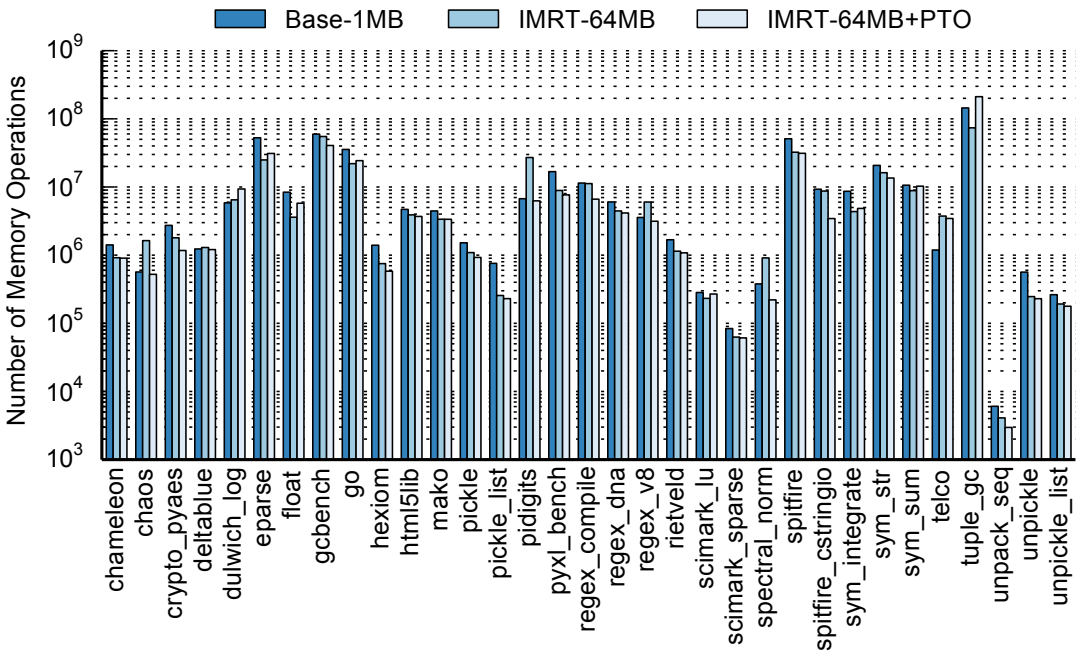


Figure 4.11: Total off-chip memory operations for PyPy.

nursery and non-nursery accesses. For V8, the IMRT table was used for both nursery and non-nursery allocations of large objects. As a result, there is also a reduction in miss-rate for the non-nursery accesses.

4.5.6 Off-Chip Memory Operations

In addition to performance improvements, the proposed optimizations also reduce off-chip memory accesses. Less memory accesses reduce energy consumption in memory. Lower memory bandwidth usage is also important for bandwidth-limited applications or systems where many applications share a memory channel. Figure 4.11 shows the number of memory operations (reads and writes) for various PyPy benchmarks. Here, only benchmarks where the memory bandwidth utilization is more than 5% of the the maximum memory bandwidth are shown. While the IMRT scheme can reduce memory

Table 4.7: Configuration of the low-end processor.

Core	4-way OOO, 2.17GHz
L1I	24kB, 6-way, 3-cycle latency
L1D	32kB, 8-way, 3-cycle latency
L2	512kB, 16-way, 14-cycle latency
Memory	6400 MB/s mem with 210-cycle latency

reads through cache installation, the partial tracing optimization can further reduce the memory writes, especially when there are many dead cache lines being written back upon eviction. On average, partial tracing reduces memory operations by an additional 13.7%. For some benchmarks, the reduction is far more significant (over 4x for `pidigits`). There are some cases where the total number of memory operations increase with partial tracing due to the memory accesses from tracing itself.

4.5.7 Microarchitectural Sweeps

Here, the effect of a different processor configuration and different last-level cache sizes is studied. A configuration for a low-end processor similar to the Intel Silvermont is used as shown in Table 4.7. Low-end processors often have smaller caches and are less tolerant to memory latency due to smaller buffers and queues. As a result, these processors are more sensitive to memory performance and the optimizations are expected to be more effective.

Figure 4.12 shows the average normalized execution time results for PyPy. In the baseline, the nursery size is adjusted to be half of the cache size. With a small cache, garbage collection is run more frequently. As the cache size increases, garbage collection becomes less frequent and a proportionately larger space is available for the non-nursery accesses. The results show that the proposed cache installation scheme with

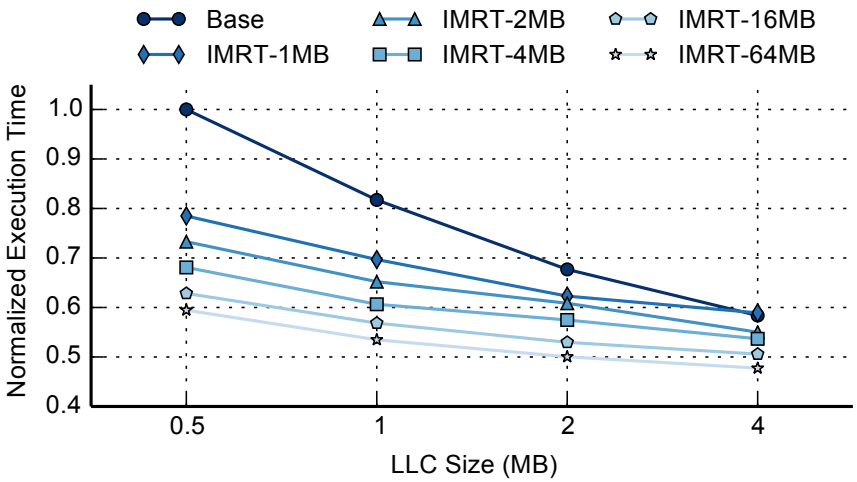


Figure 4.12: Normalized execution time of PyPy on a low-end processor with different LLC and nursery sizes.

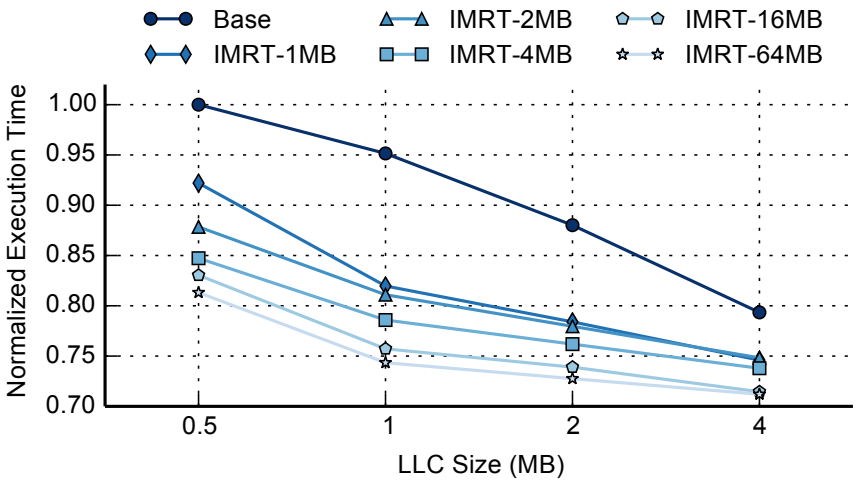


Figure 4.13: Normalized execution time of v8 on a low-end processor with different LLC and nursery sizes.

a large nursery size (IMRT-64MB with 0.5MB LLC) can improve the performance as much as increasing the cache size by 8x (Base with 4MB LLC). Furthermore, the average execution time can be reduced by 40% for a 0.5MB LLC, significantly more than in a high-performance processor. Alternatively, IMRT can perform just as well with half of the nursery size as the baseline (IMRT-1MB with 4MB LLC vs. Base with 4MB LLC).

Figure 4.13 shows the average normalized execution time results for V8. For the baseline, the nursery size is fixed to be 1MB (the minimum V8 allows). As the cache size increases, the cache performance improves and the execution time decreases. Similar to PyPy, IMRT can achieve the same performance with a larger nursery (IMRT-64MB with 0.5MB LLC) as the base with 8x the cache size (Base with 4MB LLC). In addition, the average execution time can be reduced by 19% for IMRT with 0.5MB LLC, slightly more than in a high-performance processor.

The results indicate that the cache performance is more important on low-end processors than high-end processors, and the proposed optimizations lead to more significant performance improvements for processors with smaller caches.

4.6 Other Approaches to Optimizing Application Performance

So far in this chapter, application performance was improved by increasing the nursery size and reducing the cache performance overhead by using cache installation. The advantage of the approach is that it can greatly reduce misses due to uninitialized objects. The disadvantage is that additional hardware must be introduced to perform the optimizations. This section discusses other potential approaches to improving dynamic language performance by adapting nursery allocation to the cache replacement policy and by dynamically sizing the nursery. Experimental results are also shown to evaluate the effectiveness of these approaches.

4.6.1 Adapting Nursery Allocation to Cache Replacement Policy

Since objects are sequentially allocated in the nursery and most objects die young, the access behavior for the nursery looks similar to a sequential access through a large array in the common case. If the nursery is accessed in a sequential fashion, then once the nursery size is greater than the cache size the beginning of the nursery is evicted by the additional nursery blocks allocated. For example, if the LRU cache set can hold two blocks and they are accessed sequentially, then a pattern of [1, 2, 1, 2] will hit for all accesses, but a pattern of [1, 2, 3, 1, 2, 3...] will miss in all accesses since block 3 will evict block 1 before it is reused, block 1 will evict block 2 before it is reused, and so on.

If the access pattern is modified so that it looks like [1, 2, 3, 3, 2, 1, 1, 2...], where the accesses are reversed, then the number of misses can be reduced. In this case, only 1 out of every 3 accesses will miss. As a result, the cache performance can be improved even though the nursery size is larger than the cache size.

The nursery allocator can be modified to emulate the reversing access pattern by allocating from lowest address to highest address one time and from highest address to lowest address after garbage collection and so on. For the study, the ZSim simulator was modified to transform the nursery object addresses to simulate this pattern. When objects are being allocated in the forward direction (i.e. 1, 2, 3) then the original object addresses are used. When objects are being allocated in the reverse direction (i.e. 3, 2, 1) then the addresses are mirrored so that the lowest nursery address becomes the highest nursery address and the highest nursery address becomes the lowest nursery address.

The modified allocation scheme and the original allocation scheme were evaluated using a high-performance OOO processor model in ZSim with 8MB LLC cache. Figure 4.14 shows impact of modification on the average execution time normalized to the

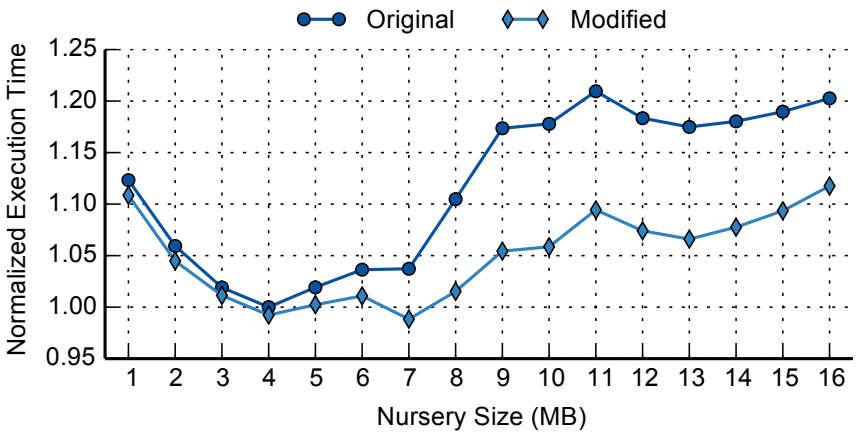


Figure 4.14: Comparing execution times of different allocation schemes for different nursery sizes.

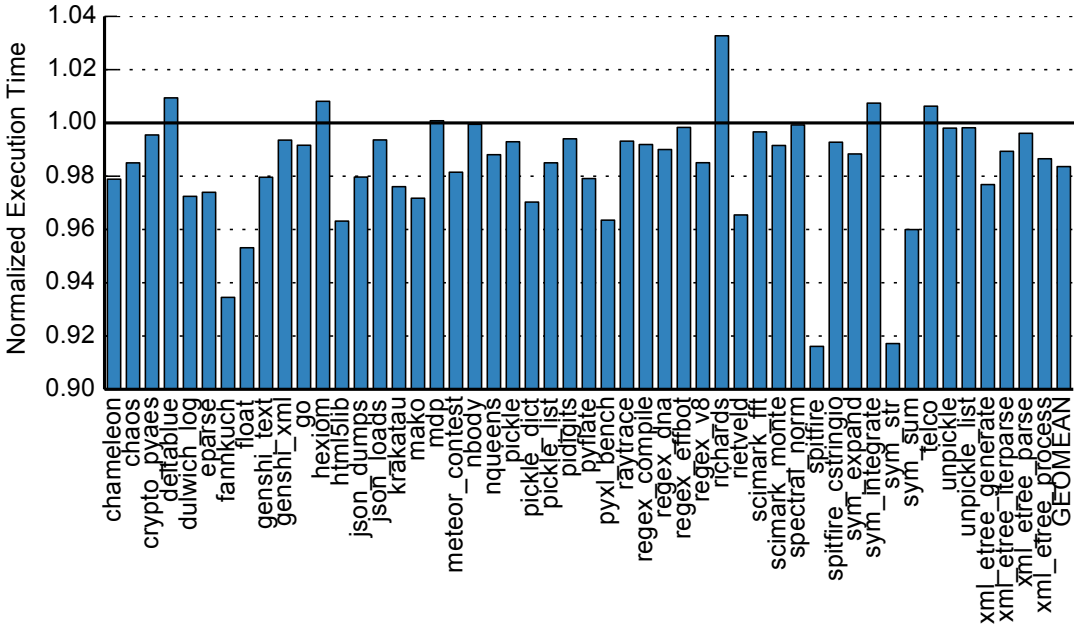


Figure 4.15: Execution time of the modified allocation scheme normalized to the execution time of the original allocation scheme when using optimal nursery sizes.

original allocation scheme with a 4MB nursery. The largest improvements come when the nursery size is larger than the cache size, with an average of 7.1% improvement for a nursery size of 16MB. However, as indicated by the average execution time, running at the largest nursery size may not be the optimal nursery size for all programs.

Figure 4.15 shows the execution time of each program when using the modified allocation scheme and the optimal nursery size normalized to the execution time of using the original allocation scheme and the optimal nursery size. There is an average 1.7% improvement with up to 8.4% improvement. It is worth noting that some benchmarks see a slowdown of as much as 3.3% due to the fact that assuming a purely sequential nursery access pattern may not be correct for all benchmarks. The results show that better adapting the allocation strategy to the replacement policy is a potential way to improve program performance.

4.6.2 Dynamic Nursery Sizing

So far, nursery size has been fixed at program startup. Choosing a best nursery size statically is a coarse-grained decision that does not consider program phase and runtime behavior. This section considers the potential improvements that can be achieved if the nursery size changes dynamically to adjust for program phase behavior. Experimental results suggest that a dynamic nursery sizing scheme has the potential to noticeably improve the performance of programs.

For the experiment, the program execution was broken down into periods that consisted of non-GC execution followed by running GC once. Based on a nursery size, the number of these periods and the length of these periods would vary. Programs were profiled at various nursery sizes and the execution time was recorded for each of the periods. The experiment was run on a native Intel machine with a 3MB LLC.

Profiling was run at nursery sizes of 756kB, 1MB, 3MB, 6MB, and 12MB. Nursery sizes were chosen so that they were multiples of each other. As a result, profiles at different nursery sizes could be mixed and matched offline and the comparisons would

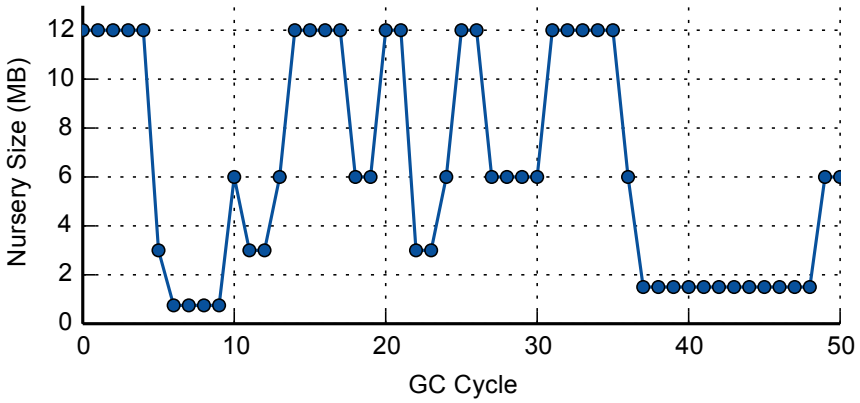


Figure 4.16: Best nursery size for `eparse` over time.

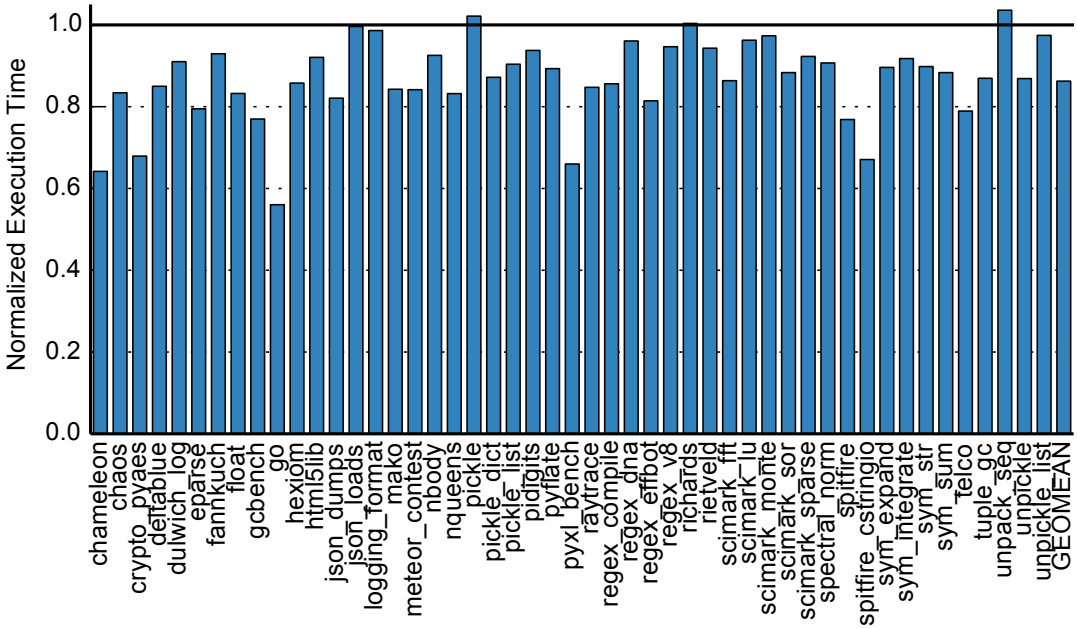


Figure 4.17: Execution time with dynamic nursery sizing normalized to the baseline with a static 1.5MB nursery.

still make sense in terms of how much of the program had been executed. For example, over the same period of execution, a large nursery size of 6MB could be used or smaller nursery sizes of [1.5MB, 1.5MB, 3MB] could be used. In both cases, the application would be allocating 6MB worth of objects in the nursery. In the latter case, there would be three GC cycles whereas in the former there would be a single GC cycle.

Using a script, the schedule of nursery sizes that would yield the best performance was generated for each benchmark. Figure 4.16 shows an example of a schedule of nursery sizes for the benchmark `eparse`. As shown in the figure, there are phases of the program where a large nursery size results in good performance and there are other phases where a small nursery size is needed for good performance.

PyPy was modified to dynamically adjust the nursery based on the schedule of nursery sizes. Figure 4.17 shows the execution time of dynamic nursery sizing across multiple benchmarks normalized to the baseline static nursery sizing of using half of the cache size (i.e. 1.5MB). On average, the execution time improves by 13.8% with up to 44% improvement. The results demonstrate that a dynamic nursery sizing approach that can consider program phase behavior can further reduce the execution time of the programs. In practice, accurately determining best nursery sizes at runtime is a challenging problem that could be addressed in future work.

CHAPTER 5

EFFICIENT NURSERY SIZING ON MULTI-CORE PROCESSORS WITH SHARED CACHES

5.1 Overview

In this chapter, it is shown that the performance of automatic memory management can be significantly affected by cache sharing among multiple concurrent applications, and offline and online schemes are proposed to adjust memory management considering cache interference. Unfortunately, cache interference in the context of managed languages has not been adequately explored. 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. The problem will be increasingly important as the number of cores per chip increase.

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 a traditional multi-programmed workload. In particular, automatic memory management provides parameters that can actively reshape an application's memory access behavior and can be used to effectively manage shared caches. This work proposes to adjust 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 increase GC overhead. As a result, a good nursery size for an application depends on both the application's own

characteristics and other applications that share the cache.

A detailed study is provided on how memory management and the execution time of a managed program interacts with cache sharing among multiple concurrent programs, and two approaches are proposed to determine the nursery sizes to optimize the overall performance considering both cache sharing and garbage collection overhead. In the first approach, the nursery size for each program is statically set based on a per-program model that uses offline profiling to predict near-optimal nursery sizes. By deconstructing the execution time into multiple components related to garbage collection and cache interference, the nursery-size model can predict an application's execution time at various nursery and cache sizes based on the profiling results of individual applications at a single cache size. In the second approach, the nursery allocator is modified to dynamically adjust the nursery size based on program behavior and observed cache pressure. By probing the cache and measuring both garbage collection and application execution times at runtime, the profile of the program can be approximated to determine the nursery size without offline profiling.

The proposed techniques are implemented and evaluated on an Intel i7-based platform running a broad range of Python benchmarks. The experimental results show that the baseline static nursery sizing heuristic without considering 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 the number of concurrent applications is increased. The results also show that the proposed nursery sizing schemes are quite effective in determining good nursery sizes and can significantly increase the performance over the baseline. When four programs run concurrently, the static scheme based on offline profiling improves the system throughput by 18.5% on average and up to 92%. This performance improvement is within 7.5% of

the best nursery sizing on average. The performance improvements for individual programs can be as high as 3.28x. The dynamic scheme also provides average performance improvements comparable to the static scheme. The dynamic scheme outperforms the static scheme in some cases as it can adjust to runtime variations. The static scheme performs better for short programs where the dynamic scheme does not have enough time to learn program's characteristics.

The following summarizes the main contributions in this chapter:

- To the best of the author's knowledge, this work represents the first to identify the performance impact of cache sharing *between* multiple applications in a managed language. It is also shown that using automatic memory management to reshape memory accesses can be an effective tool in reducing cache contention among concurrent programs.
- An analytical model is developed to better understand the interactions between the execution time of a managed language program and the cache and nursery sizes and present a static nursery sizing scheme based on offline profiling.
- A dynamic nursery sizing scheme is presented that can automatically adjust the nursery size at runtime without offline profiling.
- The proposed schemes are implemented and evaluated on a real-world system and demonstrate significant performance benefits.

The rest of this chapter is organized as follows. Section 5.2 discusses the cache-sharing problem that was identified. Section 5.3 describes the analytical model and how it can be used to choose nursery sizes considering cache sharing among multiple programs. Section 5.4 evaluates the effectiveness of the nursery-sizing approaches. Section 5.5 discusses some results on scheduling concurrent applications.

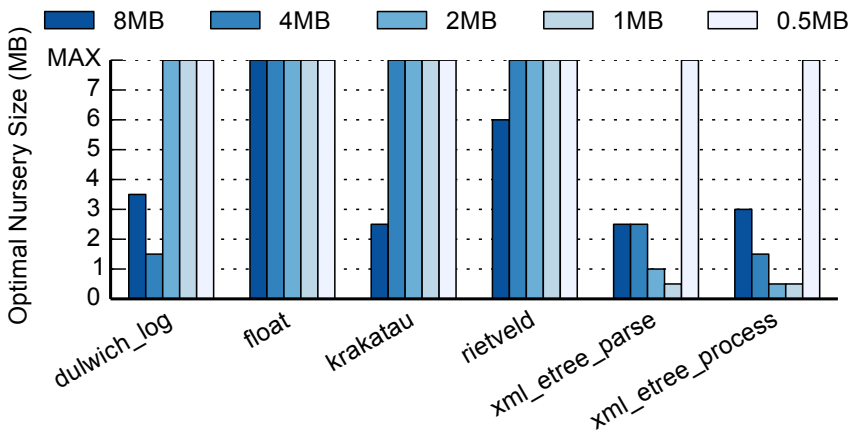


Figure 5.1: The best nursery size for individual benchmarks under different cache sizes.

5.2 Cache-Sharing Effect

5.2.1 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 result, the effective cache size for each program will be less than the actual cache size. Because an application's miss-rate often increases significantly when its nursery does not fit into the cache size, 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 overhead due to poor cache performance is higher than the 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 5.1 shows how the optimal nursery size for different benchmarks changes as

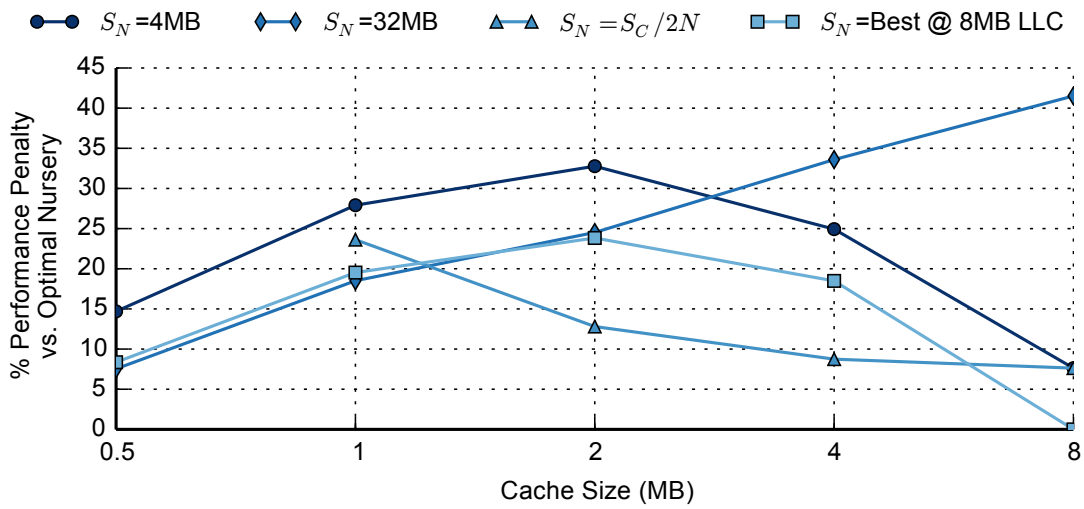


Figure 5.2: The performance penalty of fixed nursery sizing over the optimal nursery sizing at varying cache sizes.

the LLC size is varied. In all of the benchmarks besides `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 becomes the best choice. This cache size, which will be referred to as the *saturation threshold*, depends on the program characteristics and is specific to each program. Some benchmarks such as `rietveld` and `krakatau` reach this threshold at a larger cache size, while other benchmarks such as `xml_etree_*` reach this threshold at a smaller cache size. For the benchmarks with a smaller saturation threshold, it is important for the nursery to fit into the LLC and they will be affected more by cache sharing.

Figure 5.2 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 of using different nursery-sizing schemes compared to the optimal nursery size when the cache size is reduced to less than 8MB averaged over 50 Python benchmarks.

S_N denotes the nursery size. Even the nursery size is set to be the optimal size when the application has 8MB cache to itself, it becomes sub-optimal for smaller cache sizes. The average performance penalty reaches close to 25% when the optimal nursery size for a 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 use the maximum nursery size. Other baseline schemes that use half the original cache size (4MB), the maximum nursery size (32MB), or the half the effective cache ($S_C/2N$) all have significant performance penalty. Note that the penalty can be much higher than the average for individual programs. The results suggest that choosing the optimal nursery size at a single cache size is not sufficient to achieve good performance when cache sharing is considered. There is an opportunity to significantly improve overall performance if the nursery size for each program is carefully chosen considering the reduced effective cache capacity due to sharing.

5.2.2 Limitations of Cache Partitioning

Cache partitioning is a popular technique to handle cache contention among multiple concurrent programs. It can be effective in better allocating cache space and isolating accesses from different programs, but cannot reduce cache thrashing that a program experiences from its own accesses. In traditional languages, the access pattern of the program is fixed and cannot be changed. In managed languages, the memory access behavior can be altered by changing the nursery size.

In order to test the effectiveness of cache partitioning, simulations were run to approximate the throughput of applications when running with cache partitioning. 23 groups of 2 programs and 68 groups of 4 programs were tested, and the throughput was

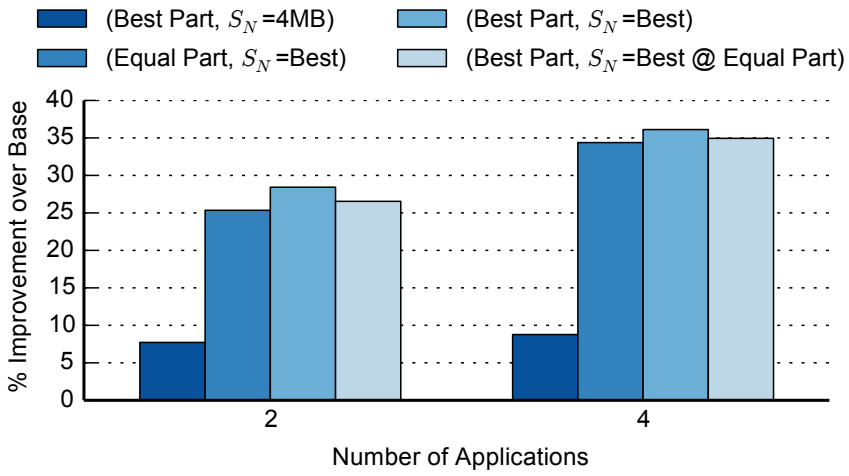


Figure 5.3: The average throughput improvement when running applications with cache partitioning. The first value in the parenthesis indicates the partitioning scheme, while the second value indicates the nursery sizing scheme. The baseline used for comparison is (Equal Part, S_N=4MB).

compared when equally partitioning and optimally partitioning a 8MB last-level cache. Figure 5.3 shows the average throughput improvement over the baseline of equal partitioning and nursery size being half of the cache size (i.e. S_N=4MB). Results for both partitioning schemes are shown when using the baseline nursery sizing scheme (i.e. half the cache size) as well as when using the best nursery size for each application for the given partition configuration. When nursery sizing is not optimal, some programs can be thrashing the cache and there is some benefit from optimal cache partitioning. Average throughput improves by 7.7% and 8.7% for groups of 2 and 4 applications respectively.

On the other hand, optimizing the nursery size can reduce the cache thrashing effect and improves average throughput by 25.3% and 34.4% (for groups of 2 and 4 applications respectively). Using optimal nursery sizes with equal partitioning results in average improvement that is within 2.5% of the average improvement for optimal cache partitioning and optimal nursery sizing. Furthermore, nursery sizing does not need to be very accurate. The optimal nursery size for equal partitions can be used even in the

optimal partitioning case with very little performance degradation ($<2\%$). This insight is used to simplify the estimation of effective cache size. Instead of predicting the exact cache footprint of the program, it is sufficient to estimate the best nursery size assuming equal partitioning of the cache.

5.3 Cache-Aware Nursery Sizing

5.3.1 Overview

This section presents an analytical model that provides intuition for how to adjust the nursery size as the effective cache size changes due to contention. The goal is to predict a good nursery size to use for each program, not to predict the exact execution time. The model is based on the garbage collection and cache performance trade-off curve by considering how different components of the curve are affected by changing cache sizes. Two nursery sizing schemes based on the model are then presented: one static scheme based on offline profiling and another dynamic scheme based on microarchitectural measurements at runtime.

5.3.2 Analytical Model

In order to determine a good nursery size to use, the impact of changing the cache size on an execution time profile of a program is considered. Given a profile $T(S_N, S_C)$ at a single cache size S_C and various nursery sizes S_N:

$$T(S_N, S_C) = T_{GC}(S_N, S_C) + T_{NGC}(S_N, S_C) + T_{int}(S_N, S_C) \tag{5.1}$$

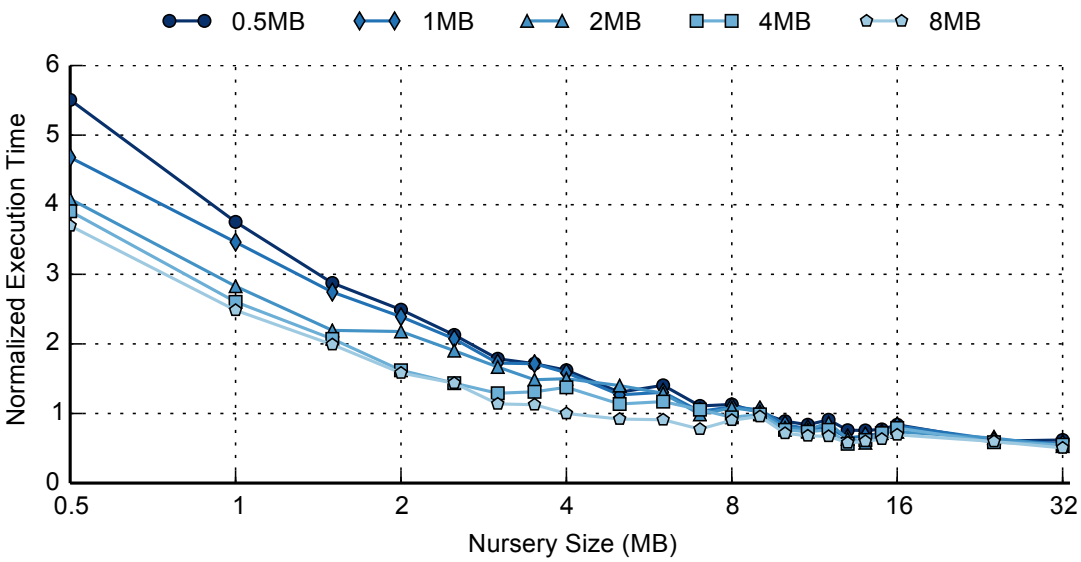


Figure 5.4: Execution time of GC normalized to GC with a 8MB cache and 4MB nursery. The x-axis shows the nursery sizes and the lines represent different memory sizes.

where $T_{GC}(S_N, S_C)$ is the time of garbage collection, $T_{NGC}(S_N, S_C)$ is the time of application (i.e. non-GC time) and $T_{int}(S_N, S_C)$ is the interference on the application from running garbage collection, the goal is to predict the profile $T(S_N, S_C^{'})$ for a different cache size $S_C^{'}$. To do this, each term of the equation is considered separately.

GC Execution Time

To the first order, the nursery size changes the number of instructions that are executed for garbage collection. It 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. The execution time of garbage collection is dominated by instruction count and caching only has a second-order effect. Therefore, to simplify the model, the following approximation is made:

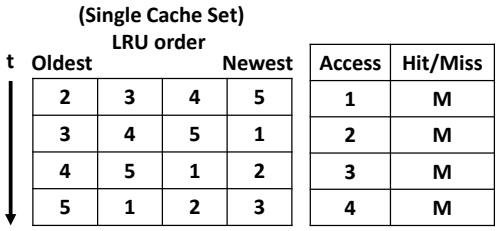
$$T_{GC}(S_N, S_C^{'}) \approx T_{GC}(S_N, S_C) \tag{5.2}$$

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 5.4. Although, the execution time curves for garbage collection at different cache sizes diverge for small nursery and cache sizes, the impact of such errors on the end result (choosing good nursery sizes) is minimal. In the static sizing scheme, the overall execution time would improve by less than 2% on average even if the prediction error in the garbage collection time were to be completely removed.

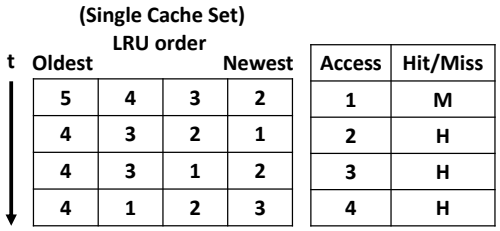
Non-GC Execution Time

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. This interaction is broken down 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.

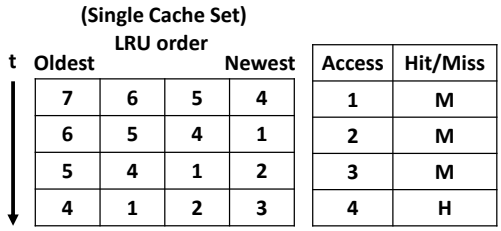
First, consider the case where the program only uses the nursery. In this case, as long as the nursery fits in the cache, there will be no cache miss, and the execution time will be small. Once the nursery size reaches the cache size, the cache miss-rate will increase until it saturates at a certain nursery size. While the saturation point varies based on each program's access patterns, it can be bounded 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 miss-rate saturation will occur when the nursery size (S_N) is equal to $S_C + S_C/N_w$. As shown in Figure 5.5(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 miss-rate saturation will occur



(a) Worst case LRU order, $S_N = S_C + 1$



(b) Best case LRU order, $S_N = S_C + 1$

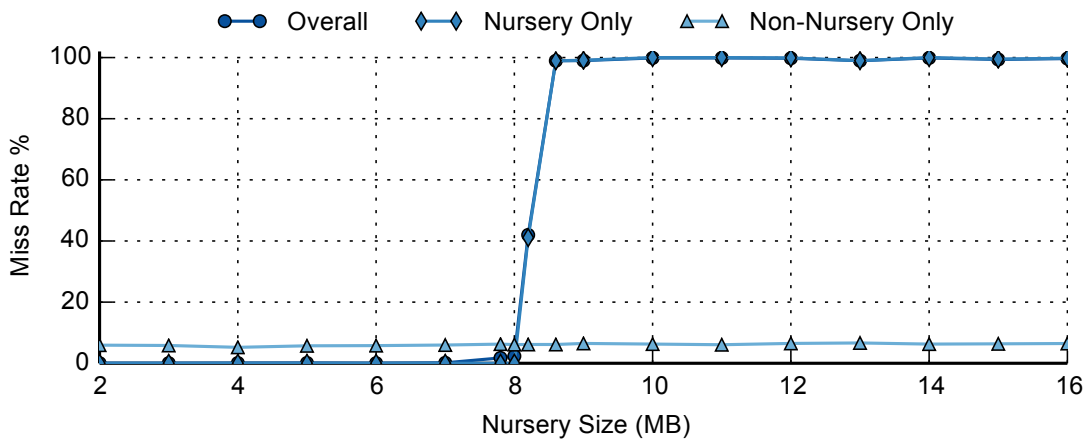


(c) Best case LRU order, $S_N = 2 * S_C - 1$

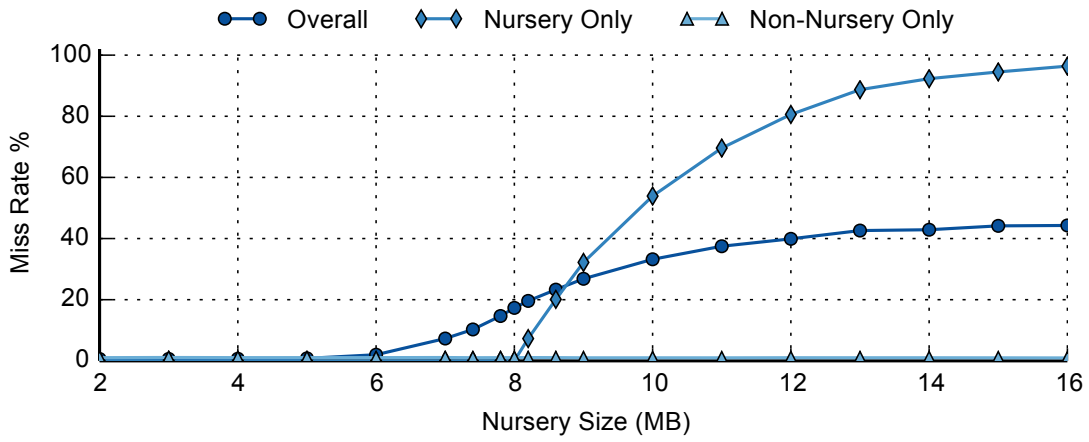
Figure 5.5: Graphical depictions of a stream of cache accesses on a single cache set.

when $S_N = 2 * S_C$. As shown in Figure 5.5(b) and Figure 5.5(c), this case happens if the nursery access pattern has temporal reuses that match the LRU policy. For example, the repeated nursery access 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 9 accesses as shown in Figure 5.5(b). In essence, reuse of earlier nursery blocks can have cache hits even if the nursery is larger than the cache. In the model, the saturation point can be conservatively estimated to occur at $2 * S_C$, and curve in the region between S_C and $2 * S_C$ can be scaled for each application to roughly fit the profiled data.

Figure 5.6 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 non-nursery. The curves of nursery-only and non-nursery only show the miss rates when those accesses were done in isolation on an 8MB cache. Figure 5.6(a) shows close to the worst case access pattern for nursery accesses. Figure 5.6(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.



(a) `fannkuch`



(b) `regex_v8`

Figure 5.6: Miss-rate curves for two benchmarks. The breakdown shows miss rates if the accesses were performed in isolation.

Now, 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 5.6, 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 as the nursery size approaches the cache size. The non-nursery working set of the program (S_W) can be approximated by looking at the nursery size at which the overall miss-rate starts to increase. If the working set is small, then the nursery size can be larger before it begins to interfere.

If the working set is large, the miss-rate begins to increase at a smaller nursery size. Therefore, the working set can be estimated to be $S_W = S_C - S_{N,W}$ where $S_{N,W}$ is the nursery size at the point where the miss rate starts to increase. $S_{N,W}$ will be referred to as the *effective cache size for nursery*. For example, S_W is 0.2MB and $S_{N,W}$ is 7.8MB for Figure 5.6(a); and S_W is 2MB and $S_{N,W}$ is 6MB for Figure 5.6(b). The model assumes that the non-nursery working set size is roughly fixed regardless of cache and nursery sizes, so the execution time curve is shifted to keep a constant S_W when predicting the curve for a different cache size.

The effect of both components can be combined to compute T_{NGC} at a different cache size $S_C^{'}$ as follows:

$$T_{NGC}(S_N, S_C^{'}) = \begin{cases} T_{NGC}(S_C - (S_C^{'} - S_N), S_C), & S_N \leq S_C^{'} \\ T_{NGC}(S_N * S_C / S_C^{'}, S_C), & S_N > S_C^{'} \end{cases} \tag{5.3}$$

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 $2 * S_C$. The curve is flat when $S_N < S_{N,W}$ and $S_N > 2 * S_C^{'}$, so applying the scales and shifts in those regions do not affect the curve in the regions.

Interference Model

Finally, consider the effect of the cache interference from garbage collection. Garbage collection needs to trace through live objects and copies the live objects out of the nursery each time it runs. These GC accesses pollute the caches with data that may not be used by a program anytime soon. Moreover, after the relocation, objects need to be reloaded from their new memory location. This effect occurs more noticeably at the L1 and L2 caches where running garbage collection effectively flushes the caches every

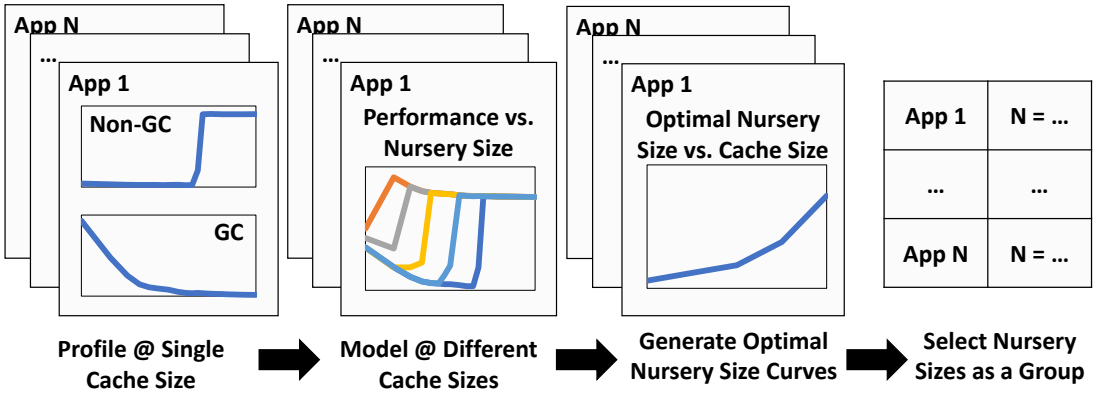


Figure 5.7: Overview of the process of determining good nursery sizes for groups of programs.

time it runs. The GC interference can be modeled with the following equation:

$$T_{int}(S_N, S_C) = \frac{S_{PA}}{S_N} t_{int} \tag{5.4}$$

where t_{int} is the overhead of interference per invocation of garbage collection, S_{PA} is the total number of bytes allocated by the program, S_{PA}/S_N represents the number of times that garbage collection runs, and T_{int} is the overall execution time overhead caused by the cache interference from garbage collection. Since the interference due to running garbage collection primarily affects the private L1 and L2 caches by effectively flushing them, it is independent of LLC cache size to the first order, so:

$$T_{int}(S_N, S_C') \approx T_{int}(S_N, S_C) \tag{5.5}$$

5.3.3 Static Nursery Sizing using Profiles

This section describes a static nursery sizing scheme based on offline profiling. The transformation described in the previous section is applied to determine good nursery sizes for groups of applications running together and those nursery sizes are set statically before the application runs. Figure 5.7 summarizes the static nursery sizing scheme. A profile, $T(S_N, S_C)$, is measured at multiple nursery sizes (S_N) and a single cache size

(S_C) by running each application in isolation without cache contention:

$$T(S_N, S_C) = T_{GC}(S_N, S_C) + T^*_{NGC}(S_N, S_C) \tag{5.6}$$

where $T_{GC}(S_N, S_C)$ is the GC time profile and $T^*_{NGC}(S_N, S_C)$ is the non-GC time profile (T_{NGC}) with GC interference (T_{int}). The following subsection describes how to deconstruct T^*_{NGC} to get T_{NGC} and T_{int}.

Once the profile is in the form of Equation 5.1, $T(S_N, S'_C)$ can be computed for a range of values of S'_C. For each value of S'_C, a range of nursery values are evaluated to determine which nursery size minimizes $T(S_N, S'_C)$. When considering a group of applications running together, the nursery size for each application that minimizes $T(S_N, S'_C)$ is chosen by setting S'_C to be the effective cache size. A following subsection describes how the effective cache size is estimated.

This approach was found to be largely transferable; the model based on application profiles from one system can also be used to predict good nursery sizes on other systems with different microarchitecture configurations. In that sense, profiling can be done once for each program by running in isolation using several nursery sizes.

Deconstructing the Non-GC Profile

Offline profiling generates T^*_{NGC} which is superposition of T_{NGC} and T_{int}. The curve can be decomposed by noting that T_{NGC} should be flat for $S_N < S_{N,W}$. However, at small nursery sizes, the GC interference is highest due to frequent garbage collection. Therefore, regression can be applied using nursery sizes less than $S_{N,W}$ to determine the coefficients of the expression:

$$T^*_{NGC}(S_N, S_C) = \frac{A}{S_N} + B = T_{int}(S_N, S_C) + T_{NGC}(S_N, S_C) \tag{5.7}$$

where A/S_N corresponds to T_{int} and B corresponds to flat part of T_{NGC}.

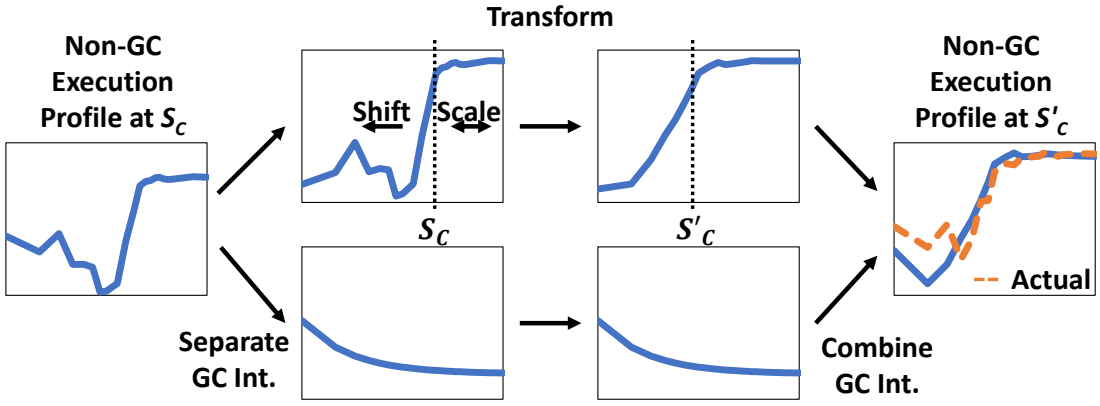


Figure 5.8: Deconstructing the Non-GC execution profile for the benchmark `crypto_pyaes`. The profile is then transformed to model the execution profile at another cache size. The actual execution profile is also shown for comparison.

Figure 5.8 shows how the parts of $T^*_{NGC}(S_N, S_C)$ are deconstructed and then transformed to model a different cache size and recombined to get $T^*_{NGC}(S_N, S'_C)$. The measured version of $T^*_{NGC}(S_N, S'_C)$ is shown for comparison. The figure shows that the techniques are effective in transforming the curve to a different cache size.

Estimating Effective Cache Sizes

In order to optimize the performance for a group of applications, the effective cache size for each of the applications running in a group needs to be determined. A simple approach would be to assume that each application is using roughly the same amount of the shared cache. For example, if there are N applications, then the effective cache size can be simply estimated to be S_C/N. As described earlier in the context of partitioning, the optimal nursery sizes assuming equal partitions of the cache are still close to optimal under ideal partitioning. Therefore, the nursery size predictions assuming equal sharing can still be good predictions even in the case where the cache sharing is not entirely equal.

Previous work by Suh et al. [134] has shown that the effective cache size for an application running as a group is roughly proportional to the number of misses that each application has over a time period. This model works well for traditional applications that have a fixed memory access pattern. Unfortunately, the memory access pattern of an application varies with the nursery size. As a result, the optimal nursery size and the effective cache size cannot be determined independently. To address this challenge, an iterative algorithm was implemented to determine a nursery size; the algorithm starts with the effective cache size of S_C/N, finds the best nursery sizes, updates the effective cache sizes based on the cache misses at the given nursery sizes, and repeats the process until the nursery sizes converge.

Both the simple and the iterative approaches were evaluated, and it was found that the overall performance improvements are comparable even though the iterative algorithm requires far more computations. In fact, the iterative approach did not converge in some cases, possibly due to inaccuracy in nursery size prediction. In addition, the simple strategy of assuming the effective cache size to be S_C/N could achieve better performance than any of the baseline sizing schemes that were tested in the experiments. As a result, the experimental evaluation uses the simple strategy.

5.3.4 Dynamic Nursery Sizing

This section describes a dynamic nursery sizing scheme that can adjust the nursery size during runtime 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. It uses cache miss rate and execution time measurements and insights from the analytical model to predict and set the nursery size at runtime.

According to Equation 5.1, T_{GC} and T_{int} are expected to be monotonically decreasing

functions that have minimal values at the maximum nursery size, S_{MAX}. On the other hand, T_{NGC} is minimal at any point where $S_N < S_{N,W}$. Therefore, the overall equation T often 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_{MAX} where garbage collection overhead is low and cache performance overhead is high. The dynamic scheme first estimates the value of $S_{N,W}$. It then tries to predict the trade-off of running at $S_{N,W}$ vs. S_{MAX} by evaluating whether $\Delta T_{GC} + \Delta T^*_{NGC} > 0$.

Estimating Effective Cache Size for Nursery

In order to evaluate the trade-off, the *effective cache size for nursery* needs to be determined. This point can change at runtime based on program characteristics and other programs that contend for the cache. To determine this size, it can be noted that objects are sequentially allocated in the nursery and long temporal reuse is rare (most objects die young). Therefore, the nursery can be considered as a large array that is being sequentially accessed. Due to cache LRU policies, the beginning of the nursery is most likely to be evicted before the later parts.

To find $S_{N,W}$, the nursery size at which the beginning of the nursery gets evicted by other memory accesses needs to be found. Cache probing can be used at fixed nursery allocation intervals 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 has been allocated, the cache probe will measure small access times, indicating that the beginning of the nursery is still in the cache. As more of the nursery is allocated, there will be a point when the access times increase indicating that the nursery is no longer fitting in the cache. Based on a history of previous probe values,

statistical outlier detection can be used to determine whether the access time increase is significant. If it is, then $S_{N,W}$ is set to be the allocation size at which this happens.

Estimating GC Time

In order to compare the trade-offs between two nursery sizes, ΔT_{GC} between $S_{N,W}$ and S_{MAX} needs to be estimated. Since GC time is amortized by running garbage collection less frequently, T_{GC} at some nursery size S_N can be used to make a first order estimate of T_{GC} at a different nursery size $S_N^{'}$:

$$T_{GC}(S_N^{'}, S_C) \approx T_{GC}(S_N, S_C) * S_N / S_N^{'} \tag{5.8}$$

Historical values of GC time can be used to estimate the current GC time at a nursery size. At program initialization, when there are no historical values, the first GC time is approximated as being a fixed percentage of the measured non-GC time. 25% is used for $S_{N,W}$ 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 a recent point at a very different nursery size (S_{RCNT}) and a less recent point at close to the same nursery size (S_{CLSE}); using S_{RCNT} reflects the current program phase, while S_{CLSE} requires less extrapolation. Since Equation 5.8 tends to over-estimate the GC time at smaller nursery sizes, it was found that using $min(T_{GC}(S_{RCNT}), T_{GC}(S_{CLSE}))$ worked well.

Estimating Cache Performance Penalty

ΔT^{*}_{NGC} corresponds to the cache performance penalty between using the smaller nursery size of $S_{N,W}$ that fits in the cache and the larger nursery size of S_{MAX} that does not

fit. Initially when there are no measurements, a rough estimate of the additional cycle penalty per byte allocated is used based on doing measurements on a small array. The first access of each array element will miss, while the second access of each array element will hit.

It is possible to measure T_{NGC} by running at both of the nursery sizes and take the difference. However, there may be enough time elapsed in the program between the two GC cycles that the program phase could change. Instead, the time elapsed between the measurements can be reduced by noting that in the first period after the nursery size is changed from $S_{N,W}$ to S_{MAX}, allocations to the first $S_{N,W}$ of the nursery still hit in the cache. The remaining allocations that occur after $S_{N,W}$ miss in the cache. Therefore, a measurement of T_{NGC} can be done for $0 < S_N < S_{N,W}$ signifying the nursery hitting and another measurement can be done for $S_{N,W} < S_N < S_{MAX}$ signifying the nursery missing. The measurements are scaled by the number of bytes allocated in each region and subtracted to get the difference in execution time between each byte of allocation that hits and each byte that misses. A running average of the cache performance penalty can be used to eliminate noise across measurements.

Implementation Details

A nursery allocator was designed that could allocate chunks of the nursery at a time to the program instead of the full nursery. A chunk size of 256kB was used to balance between too many allocator calls and fine grain decision-making. At the point where the allocated nursery size reached $S_{N,W}$, estimates of ΔT_{GC} and ΔT^{*}_{NGC} were used to determine whether or not to garbage collect. When running at $S_N = S_{MAX}$, the trade-off comparison always happened at the newest the value of $S_{N,W}$. After running for 15 GC cyles with the same nursery size, the allocator would switch to $S_{N,W}$ for a single

Table 5.1: Microarchitectural details of platform.

	Native Machine	ZSim Config
Core	i7-6700, 3.4GHz	Simple, 3.4GHz
	4 physical, 8 virtual	Single in-order unpipelined
L1I	32kB, 8-way	32kB, 8-way, 1-cycle
L1D	32kB, 8-way	32kB, 8-way, 1-cycle
L2	256kB, 8-way	256kB, 8-way, 12-cycle
LLC	8MB, 16-way	0.5-8MB, 16-way, 42-cycle
Memory	16GB, DDR3-1600	Simple, 173-cycle

GC cycle (regardless of whether or not it was optimal) to update the models. When running at $S_N = S_{N,W}$, the same nursery size would continue to be used unless a new $S'_{N,W}$ was detected that was less than the current $S_{N,W}$ or it was estimated that S_{MAX} was a better nursery size to use. This prevented constant fluctuations in nursery size while allowing for fast reaction to increased cache pressure or garbage collection overhead. After running for 15 GC cycles, the allocator would switch to the new value of $S_{N,W}$.

5.4 Evaluation

5.4.1 Experimental Setup

The experiments were run on two platforms as detailed in Table 5.1. The main experiments were run on an Intel i7-based system. For the profile portability study, ZSim [121] was used with a simple core model. In the simple core model, instruction latency is only affected by misses in the instruction and data caches. The results show that the profile based on the ZSim simulations with a simple core model can be used to predict good nursery sizes on the Intel-based machine, suggesting that execution profiles are portable across systems.

Table 5.2: Summary of nursery sizing schemes used for evaluation.

Scheme	**Description**
Base	Set the nursery size (S_N) to 4MB.
$S_C/2N$	Set S_N to half of the cache size divided by the number of concurrent applications.
Best @ 8MB LLC	Use the optimal nursery size for a 8MB cache.
Static	Proposed static nursery sizing scheme based on offline profiles.
Dynamic	Proposed dynamic nursery sizing scheme.
Best	Best static nursery size combinations from experiments.

The experiments were run using PyPy [15], a high-performance implementation of Python with generational garbage collection and just-in-time compilation. Other languages were not evaluated, but the proposed technique is believed to be 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, PyPy's default static nursery sizing was used where the nursery size was set to be half of the machine's last-level cache size (e.g. 4MB in this case). Other static nursery sizing techniques were also compared as summarized in Table 5.2.

For benchmarks, a wide array of applications were used to get a representative sample of real-world applications. A combination of benchmarks from the official Python benchmark suite [109] and benchmarks from the PyPy benchmark suite were used. In total, designs were evaluated with 50 benchmarks. The benchmarks are listed in Table 5.3 along with the acronyms used in the figures.

To study cache contention among applications, groups of applications needed to be chosen to run concurrently. Instead of manually selecting applications based on known characteristics, groups of applications were randomly selected. In some cases multiple instances of the same application were also run together. Groups of 2, 3, and 4 applications concurrently were evaluated. Experiments were run for 23 randomly-

Table 5.3: 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

chosen groups of 2, 19 randomly-chosen groups of 3, and 68 randomly-chosen groups of 4 benchmarks. For the results of the groups of 4 benchmarks, averages include all 68 groups but only 34 are shown in some graphs for readability. They are selected by sampling every other group after sorting based on performance improvement.

Each benchmark group was run until the longest running benchmark completed at least one execution. Shorter running applications would keep running for more than one execution so they could continue to interfere with the longer running applications. The `rdtsc` instruction was used to get the total execution time of only the first run of each

benchmark to use for performance evaluation and to ensure that the same amount of fixed work was compared for each benchmark.

As a performance metric, the system throughput (STP) was computed using the following equation:

$$Throughput(STP) = \sum_{i} \frac{T_{i,alone,S_N=4MB}}{T_{i,shared}} \tag{5.9}$$

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

GNU Parallel [135] was used to schedule the applications to run concurrently on the machine cores in a way that reduced the possibility of multiple applications running on the same physical core. The processor used in the study had 4 physical cores and 8 virtual cores with hyper-threading. As the study focused on cache contention, hyper-threading was not used and each application got a dedicated physical core.

In order to find the best possible performance, each group of benchmarks was run with all possible combinations of nursery sizes. Since the nursery size was a continuous variable in PyPy (in some runtimes, it would have to be a power of 2), 5 discrete nursery size points were chosen to enumerate possible combinations: 1MB, 2MB, 4MB, 8MB, and 32MB. For groups of 2 benchmarks, there were 25 combinations. For groups of 3 benchmarks, there were 125 combinations. For groups of 4 benchmarks, there were 625 combinations. For the static nursery sizing scheme, one of the five discrete nursery sizes was selected in experiments so that the results could be compared with the best nursery size combination. In the dynamic nursery sizing scheme, any nursery size at 256kB granularity could be selected.

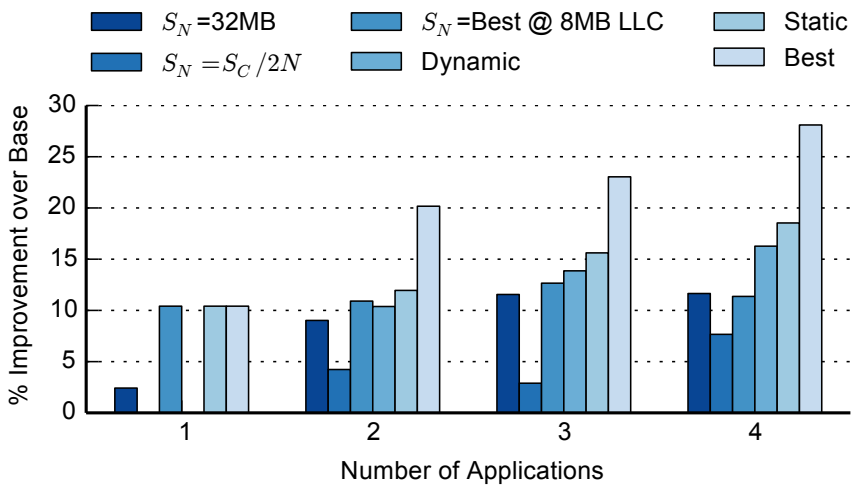


Figure 5.9: Average improvement of different nursery sizing schemes over the baseline nursery sizing scheme with different number of applications running.

5.4.2 Performance Results

This study evaluates the system throughput when applications are run in groups to compare how well the nursery sizing schemes perform. The static scheme used offline profiles from the Intel machine with five nursery sizes 1MB, 2MB, 4MB, 8MB, and 32MB for each application to determine the best nursery size to use with cache contention.

Figure 5.9 shows the average throughput results of varying nursery sizing schemes for different numbers of concurrently running applications. While using the maximum nursery size (32MB) often results in poor performance when running an application individually, the large nursery can improve the system throughput when running multiple applications together. This is because both small (4MB) and large (32MB) nursery sizes may have poor cache performance when multiple applications share the cache. In that case, the 32MB nursery is beneficial because it has lower garbage collection overhead. On the other hand, setting the nursery to be half of the effective cache size helps when reducing cache contention is more important than reducing garbage collection time. Us-

ing the optimal nursery size for the 8MB cache without considering cache sharing is slightly better than the baseline but at best yields 11.4% throughput improvement in the case of four applications.

The proposed sizing schemes consistently outperform other baseline nursery sizing schemes. The results show that the static nursery sizing scheme can improve the system throughput by 12.0%, 15.6%, and 18.5% on average compared to the baseline. The average improvements are within 6.8%, 6.0%, 7.5% of the best for the groups of 2, 3, and 4 applications, respectively. Moreover, as shown shortly, the improvements for individual application groups can be far more significant. The dynamic nursery sizing scheme performs slightly worse than the static approach with average improvements of 10.4%, 13.9%, 16.3% over the baseline. This result shows that the dynamic scheme can adjust the nursery size effectively even without prior profiling.

It is worth noting that when one application is running, the static nursery sizing scheme is equivalent to the best nursery sizing scheme since the profile is collected for a 8MB cache. On the other hand, the dynamic scheme on average shows 0% improvement over the baseline because it must adjust the nursery size without prior profiling.

Figures 5.10, 5.11, and 5.12 show the performance results for individual application groups. For each graph, the groups are ordered by the improvement of the static scheme over the baseline. The static scheme improves the system throughput by over 15% for 5 out of 23 groups of 2 applications, by over 20% for 11 out of 19 groups of 3 applications, and by over 30% for 19 out of 68 groups of 4 applications. In addition, the maximum improvement is 88%, 35%, 92% for groups of 2, 3, and 4 applications respectively. Individual applications can have performance improvements of as high as 3.29x. In many cases, the static scheme can achieve the best throughput or close to the best throughput. There are a few cases where the dynamic scheme can do better than

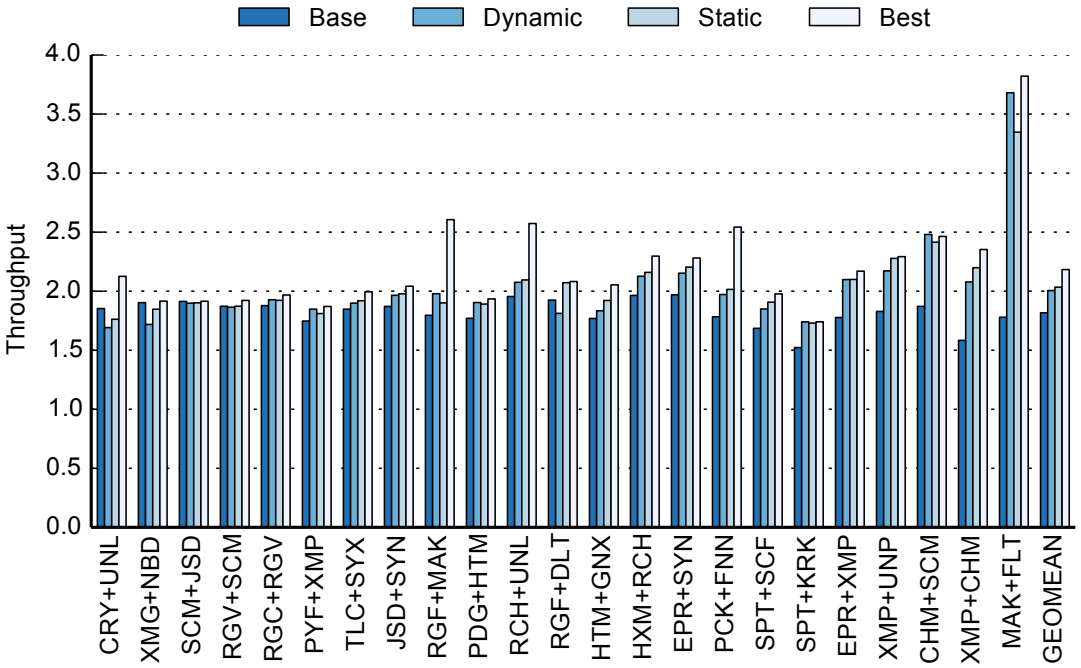


Figure 5.10: Detailed throughput results when running two applications concurrently.

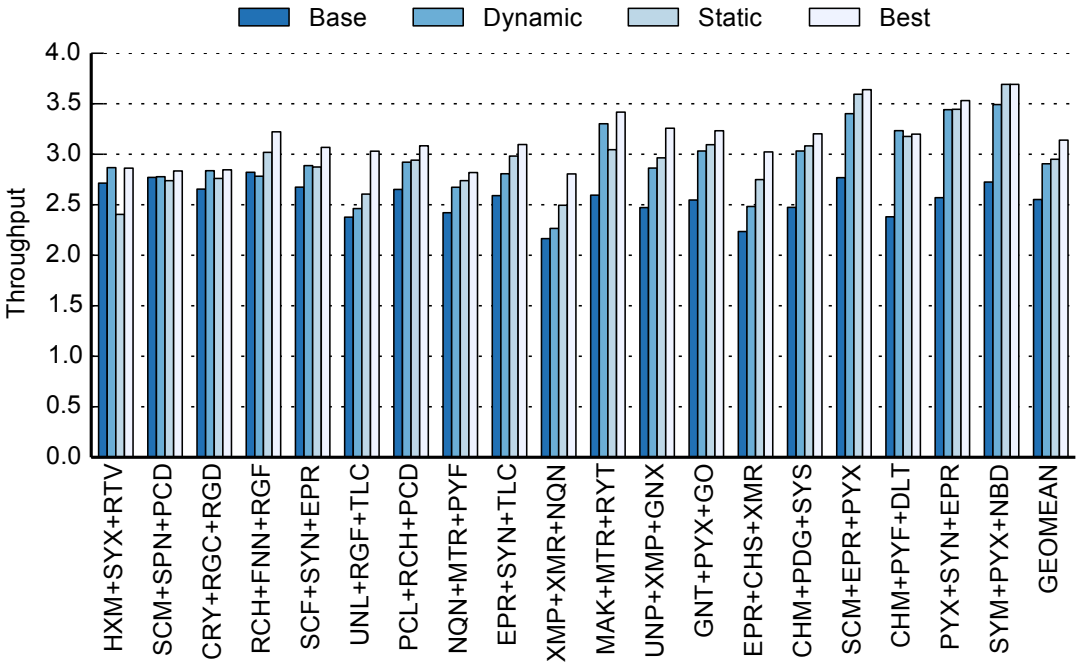


Figure 5.11: Detailed throughput results when running three applications concurrently.

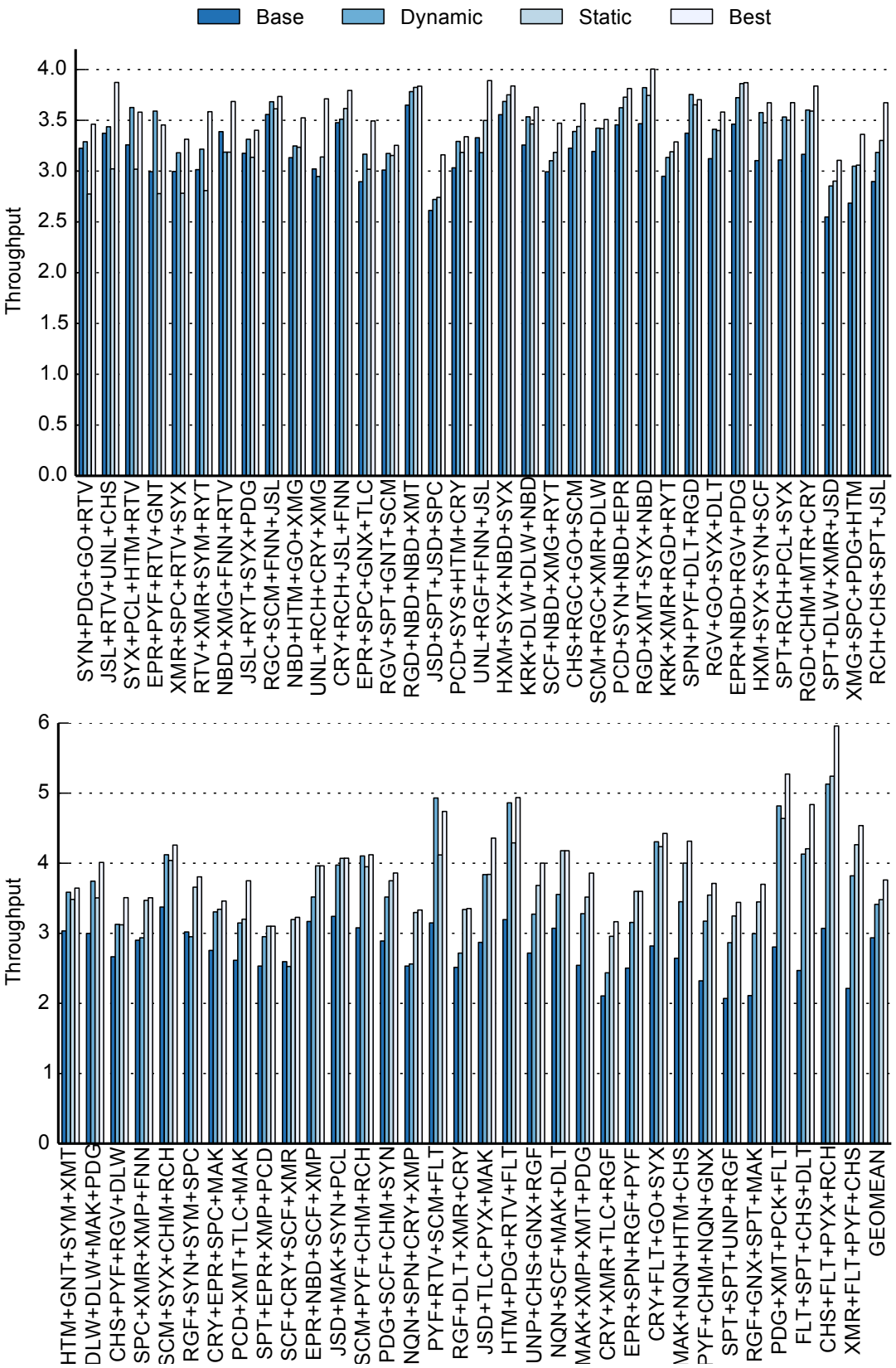


Figure 5.12: Detailed throughput results when running four applications concurrently.

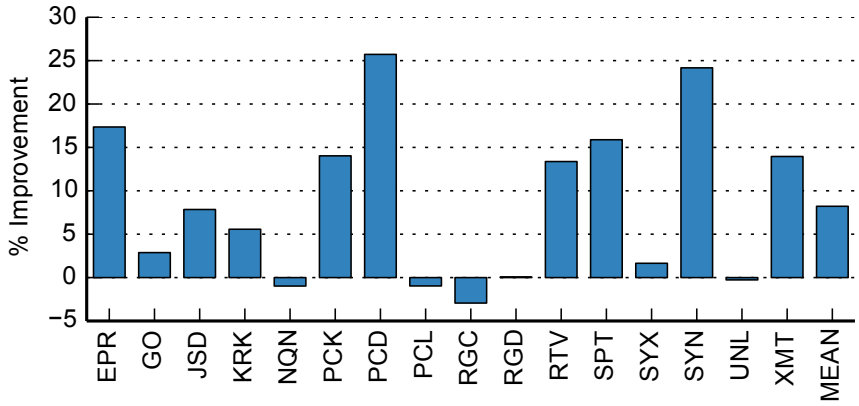


Figure 5.13: Performance improvement of the dynamic scheme over the baseline static nursery sizing scheme for various benchmarks that are scheduled to run together.

even the best static nursery sizing scheme, demonstrating one of the benefits of adapting the nursery size based on the program phase behavior.

5.4.3 Dynamic Scheduling

This study considers the case where many applications are scheduled to run together on the machine. 16 applications were run in alphabetical order with a maximum of 4 running concurrently on the 4 cores of the machine. The cache contention changed over time as applications were starting up and exiting. The proposed dynamic scheme is compared to the baseline scheme of setting the nursery to be 4MB in Figure 5.13. The execution time of the applications can be improved by an average of 8.2% and by more than 10% for 7 of the 16 benchmarks. This further demonstrates the benefits of having a dynamic scheme over a static scheme.

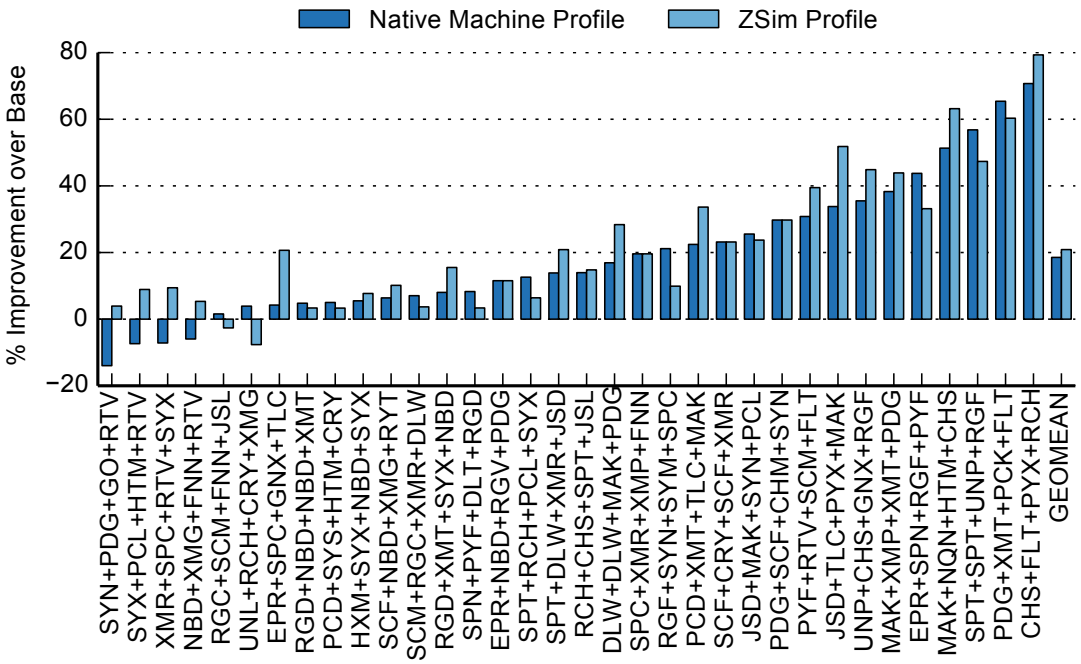


Figure 5.14: Comparison of the improvements of group of four applications when using native machine vs. ZSim to generate the profile.

5.4.4 Profile Portability

This study evaluates whether an execution profile collected on one system can be used to make predictions on a different system for the static scheme. Execution profiles were generated from ZSim simulations with five nursery sizes of 1MB, 2MB, 4MB, 8MB, and 32MB. The ZSim simulator was configured to use a simple in-order core that did not hide any cache miss latency. All instructions besides memory instructions took one cycle.

Figure 5.14 shows a comparison of the system throughput improvements for a subset of applications running in groups of four. Surprisingly, even with a profile collected on a simple simulator, performance improvements are close to those based on profiles on the native Intel machine. There are some application groups where using the simulator profile yields better results and some application groups where using the native machine

Table 5.4: Profiles compared for input sensitivity evaluation.

Profile	Description
Same Input	Use the same input for profiling and evaluation.
Multiple Inputs	Use the average execution time of running with multiple inputs for profiling and use a single input for evaluation.
Different Input	Use a different input for profiling than for evaluation.

profile yields better results, but the difference is less than 2% on average across benchmark groups. The result indicates that application profiles are indeed portable and they do not have to be collected for each target system.

5.4.5 Input Sensitivity

This study evaluates how resilient the modeling for the static scheme is to changes in input sets for the benchmarks. Among the 50 benchmarks shown in Table 5.3, there are some that run the same Python program with different command-line arguments. They are `genshi*`, `*pickle*`, `scimark*`, `spitfire*`, `sym*`, `xml*`. To test input sensitivity, the static nursery scheme was used with two alternative profiles for each of these benchmarks to see if the predicted nursery sizes would be different enough to see noticeable changes in throughput. The profiles that were compared are shown in Table 5.4. The baseline profile was constructed using the same input that was used for running each benchmark. Next, a profile was constructed for each of these benchmarks by running them with multiple inputs and averaging the execution time. Finally, a profile was constructed using an input that was different from the input used for running each benchmark.

Figure 5.15 shows the change in throughput for the group of four applications when using the multiple inputs and different input profiles compared to using the same input

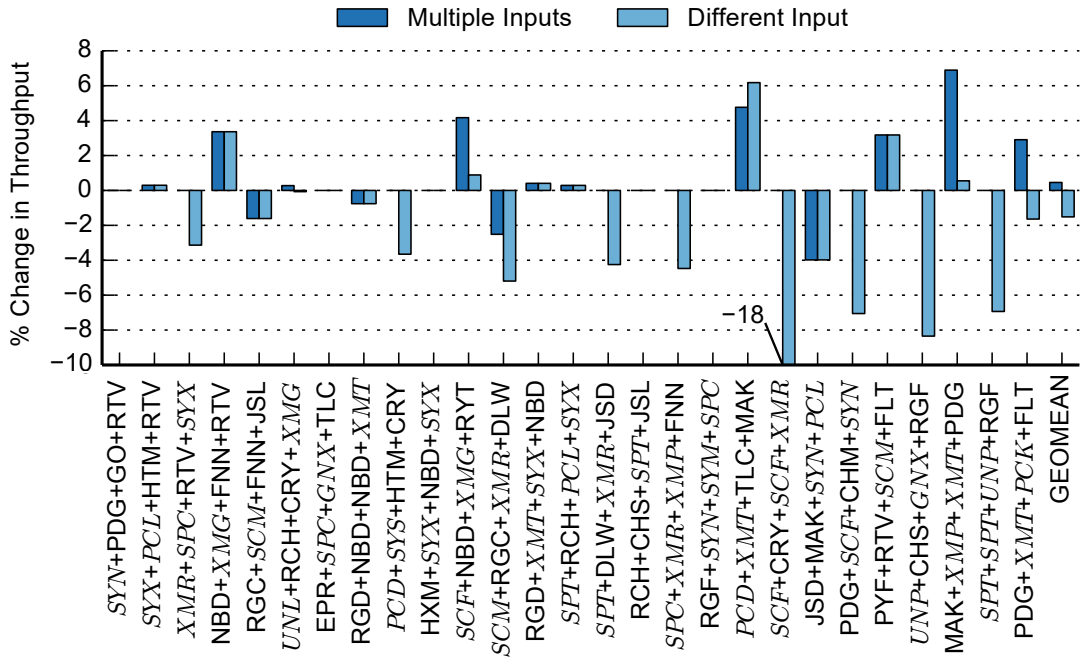


Figure 5.15: Change in throughput in the static nursery scheme for groups of four applications when profiling with multiple inputs and a different input compared to profiling using the same input. Benchmarks in italics had their profiles changed.

profile. In some cases for the multiple inputs profile, there is a drop in throughput of as much as 4%, but in other cases, the multiple inputs profile actually results in better predictions and there is close to 7% improvement in throughput. The average throughput changes by less than 1%. When using a different input profile for each benchmark, the profile using one input sometimes does not fully capture the correct phase behavior of the program running with another input. Although the throughput drops by 18% for `SCF+CRY+SCF+XMR` compared to the static nursery sizing scheme with the same input profile, there is still a 1.1% improvement in the throughput for that group when compared to the baseline nursery sizing scheme. On average, the throughput when using a different input profile changes by less than 2%. These results suggest that changes to program input will not significantly affect the ability of static scheme to predict good nursery sizes.

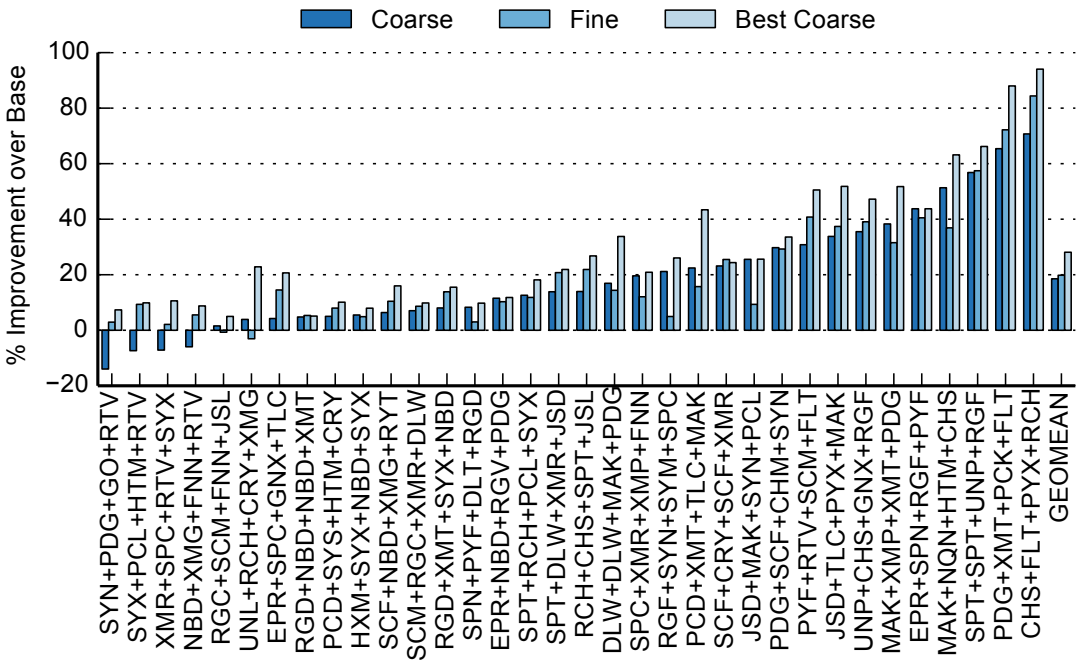


Figure 5.16: Comparison of the improvements of group of four applications when using finer nursery sizes.

5.4.6 Fine-Grained Nursery Sizing

So far, the static scheme has been constrained to select nursery sizes from among a small number of options (powers of 2). However, the nursery size in PyPy can be any value. Since the model is supposed to be able to predict the optimal nursery size, the performance improvements when any nursery size value can be chosen is studied.

Figure 5.16 compares the system throughput improvements of the coarse-grained (i.e. powers of 2) and fine-grained nursery sizing using the static sizing scheme as well as the best coarse-grained nursery selection. The results show that the fine-grained sizing in general performs better than the coarse-grained selection. On average, the fine-grained nursery sizing improves the system throughput by 19.8% over the baseline, which is about 1.3% better than the coarse-grained nursery selection. For a few benchmark groups, the fine-grained nursery sizing outperforms even the best coarse-grain

Table 5.5: Comparison of schedules for groups of eight.

	Group 1	Group 2
Worst	RGF+DLT+SCF+PDG XMR+CRY+TLC+GO	RGF+DLT+GO+RGD XMR+CRY+MAK+FLT
Best	RGF+SCF+GO+PDG DLT+XMR+CRY+TLC	RGF+XMR+DLT+FLT GO+CRY+MAK+RGD
% Improvement	6.43%	8.95%

	Group 3
Worst	RGF+TLC+SCF+RGD XMR+GO+CRY+FLT
Best	RGF+SCF+GO+FLT XMR+TLC+CRY+RGD
% Improvement	10.37%

combination. However, there are a few cases where the fine-grained scheme is worse than the coarse-grained one, possibly due to inaccuracies in the model. The results also show that the proposed static nursery sizing scheme is flexible enough to work with any granularity of nursery sizing.

5.5 Study on Scheduling

In the previous section, groups of benchmarks to run were randomly chosen and the proposed schemes achieved significant performance improvements regardless of which specific benchmarks were running together. This study evaluates the impact on performance of choosing which benchmarks to run together.

For the study, groups of eight benchmarks were chosen and all possible combinations of four running concurrently were tried. Each benchmark combination was run for all possible combinations of nursery sizes of 1MB, 2MB, and 32MB to determine the best nursery sizes to use for each benchmark combination.

Table 5.5 shows the comparison of the best and worst schedule for three groups of eight benchmarks as well as the percentage improvement in throughput for going from the worst schedule to the best schedule. The evaluated throughput is for the best nursery size combinations for a given schedule, but the percent improvements are similar if a baseline nursery sizing scheme is used instead. The results shown represent the maximum additional improvement that can be achieved through intelligent scheduling. They indicate that there is some room for improvement based on scheduling, but it may be secondary to optimal nursery sizing.

CHAPTER 6

RELATED WORK

6.1 Sources of Overhead in Dynamic Languages

There are a few previous studies that break down and quantitatively analyze the language execution of Python [22, 26, 11]. One study by Barany [11] deconstructs the interpreter performance by identifying common sources of overhead in dynamic languages. Their `pylibjit` tool uses a just-in-time compiler that optimizes Python code to quantify the effects of various sources of overhead. The limitations of the previous work is that they require custom annotations in Python programs and as a result can only perform analysis on a small number of benchmarks. Using the approach in the sources of overhead study in this dissertation, any Python program can be run to generate a breakdown of the execution time by annotating the interpreter once. In addition, this dissertation is the first to point out C function calls as a significant source of overhead for CPython.

There exists a number of studies [60, 19, 80] that use benchmarking to understand which kinds of code work well for different Python implementations. For example, Heynssens [60] has a masters thesis on a benchmarking methodology and an analysis for Python programs. He draws his conclusions based on the results from various benchmarks running on different Python implementations. These studies compare execution times but do not breakdown sources of overhead.

Some studies have directly proposed modifications to the CPython interpreter to improve its performance. Cao et al. [22] find that Python dispatch overhead can be 25% of the execution time and use pretranslation to get up to 18% improvement. Power and Rubinsteyn [107] convert the stack-based bytecodes to a register-based format that

exposes more possibilities for optimization. The optimizations focus on improving a specific aspect of the interpreter instead of breakdown of various overheads.

Ilbeyi et al. [67] analyze the performance of Python in the context of a meta-JIT framework. They present overall execution time comparisons between CPython and PyPy with and without JIT in addition to a detailed breakdown of the overhead in the context of the JIT framework. The work in the dissertation is complementary as the focus is on execution time breakdowns of the interpreter and sweep of microarchitectural and runtime parameters. The findings on sensitivity to cache and memory parameters and the interactions of garbage collection with the cache are new insights for PyPy.

There is more extensive work in JavaScript to quantitatively understand the sources of overhead [137, 131, 39, 153, 102, 99, 132]. Some work provides microarchitectural characterization of JavaScript workloads, but they do so to study the differences in benchmark suites and how well the benchmarks match real workloads. For example, Tiwari and Solihin [137] analyze the difference between the Sunspider and V8 benchmarks. Dot et al. [39] identify checks as a major source of overhead in V8. The findings of C function calls and sensitivity to cache and memory designs, which are generalized to V8, discussed in this dissertation are not discussed in the previous work.

6.2 Optimizing Indirect Branches in Interpreter Design

This dissertation points out C function calls as a major contributor of overhead for dynamic languages. The problem of optimizing indirect branches has been extensively studied, because it encompasses virtual calls in static languages such as C++ and Java in addition to interpreters. The work can be roughly categorized in two parts. First, some work deals with improving the BTB performance to improve the overall performance

at the microarchitectural level. Second, other work deals with profiling and identifying virtual functions to inline or convert to direct function calls at the compiler or source code level.

6.2.1 Indirect Branch Resolution

Indirect branches are branches where the jump target is specified through a register. They have been identified as a problem because they are hard to predict and lead to expensive pipeline flushes in out-of-order processors. A common BTB design uses the PC of an instruction to store the most recent branch targets. However, indirect branches usually have many targets that are context specific. Many papers analyze the overhead of these indirect branches [20, 40, 122, 37, 41, 141, 118, 23]. Ertl and Gregg [45, 47, 46] have shown that interpreters behave differently from normal programs in that they have a large number of indirect branches. They show that prediction accuracy for these indirect branches is only 2-50%. Casey et al. [23] discuss how to improve the BTB performance in the context of interpreters. While their proposed optimizations improve the indirect calls, the study in this dissertation finds that indirect calls account for only 11.9% of the C function call overhead.

There have been many variations in BTB designs that can improve the poor indirect branch accuracy. For example, using multi-stage [43] or cascaded predictors [42] which use more history to improve accuracy. There can also be additional data structures which can improve prediction such as a data structure which performs partial matching [76] or an additional cache to store more history [27]. Kim et al. [78] propose to separate a single indirect branch into multiple virtual PCs and use those virtual PCs to index the BTB instead of real PC values to lower misprediction rates. However, in the case of inter-

preters where there could be hundreds of targets for a single branch, the VPC approach would pollute the BTB and could lead to worse overall performance. Farooq et al. [48] generalize the idea to have the BTB indexed by any value. However, these hardware-only approaches cannot reduce the increased number of instructions that account for inefficiencies in interpreters.

Choi et al. [30] and Kim et al. [77] recently proposed to use more runtime information to improve the performance of indirect branches in dynamic languages. Kim et al. [77] use the BTB to bypass part of the dispatch loop which is unnecessary to execute every iteration, while Choi et al. [30] apply a similar strategy to accelerate the dispatch of calls to functions through the software polymorphic inline caches. In both cases, they use the BTB to aid in bypassing part of the execution. However, their techniques only consider a single value for indexing – bytecode in one case and object type in the other – and can therefore only eliminate a limited portion of the overhead. In addition, they do not consider cases where they would need to jump across function boundaries.

There are alternative techniques to improve indirect branch performance like converting branches to predicated execution [73, 72] or separating prediction from resolution through additional ISA instructions [91] or using code generation to expand indirect branches [89].

Finally, it is worth noting that although there is a large volume of work related to improving indirect branch prediction, Rohou et al. [116] recently showed that the newest Intel processors and state-of-art BTB designs actually perform quite well. Their analysis would not directly apply to embedded processors which would more likely run unoptimized interpreters and have simpler BTB designs.

6.2.2 Interprocedural Analysis and Call Graph Construction

Instead of improving the microarchitecture, there have been proposals to improve the code itself and eliminate unnecessary indirect or virtual function calls. Virtual function calls are used in object-oriented languages to simplify programming. The most effective techniques for eliminating virtual function calls are devirtualization and inlining. Devirtualization converts a virtual function call to a direct virtual function call while inlining eliminates function call overhead and allows the compiler to perform more global optimizations.

Davidson and Holler [32] derived analytical expressions for the effects of inlining on the performance. They found through experiments that inlining does not always help overall performance. Global optimization makes the register allocation problem more complex and the compiler may choose a worse register assignment when there are more variables to consider. However, there has been subsequent work which looks at type analysis as a way to aid in inlining [62, 25, 150, 56, 35, 36, 57, 2, 10].

There have been attempts to profile a program through call graph construction [55, 3] or class analysis to perform interprocedural analysis or devirtualization [33, 105, 62, 54, 95, 68, 9, 97]. Interprocedural analysis can perform some optimizations that would be performed with inlining, but without having to actually inline the function. Interprocedural optimizations can be expensive, but DeFouw et al. [34] found a way to improve a O(N^3) algorithm to a lesser complexity. Li et al. [84] and Johnson et al. [75] present more lightweight techniques to perform interprocedural optimizations based on feedback-directed optimizations and summary-based whole-program analysis. Zhuang et al. [154] and Uzelac et al. [139] similarly present ways to more efficiently profile programs to limit the effect on the overall program performance. Since most of this work has been done in the context of static languages, the interprocedural analysis

techniques would need to be adapted to dynamic languages and new approaches would need to be proposed.

6.3 Just-in-Time Compilation Research

There are a few studies with Java which point out the question of when it is appropriate to JIT [111, 112, 133]. Radhakrishnan and John [112] explore this issue in the context of Java and find that method reuse characteristics can help inform whether or not to JIT. Quantitative analysis in this dissertation suggests that intelligent use of JIT compilation in dynamic languages may also lead to better performance gains. Future work would need to explore this issue in more detail.

Most of the research on just-in-time compilers focuses on improving the compiler with feedback-directed optimizations (FDO) and reducing profiling overhead for optimizations. The goal is to reduce the cost of optimization and provide more useful information to the compiler to produce more optimized code. There is some work which focuses on using hardware to aid profile collection. Mock et al. [96] and Arnold et al. [7] focused on automating FDO while limiting the profling overhead to 2-3%.

JIT compilers usually use a loop or a method as the unit of compilation to limit the instrumentation overhead of profiling. Yasue et al. [148] present a way to break the method further and still collect accurate path profiles with low overhead. Waley [143] similarly presents an optimization scheme to avoid compiling the cold regions of a method that is going to be optimized.

There is some work on identifying phases in program execution. On changing phases, JIT optimizations can become invalid and identifying phase changes can help

reduce the guard failure cost of execution. Sherwood et al. [125] describe a profiling architecture to detect phase-based program behavior using hardware-based metrics which they find strongly correlate with phase changes. Barnes et al. [12] also use a hardware profiler to automatically detect execution phases and record branch profile information for each new phase. Their profiler only focuses on hot code execution.

Many JIT frameworks still use basic counters to determine candidates for optimization. Adl et al. [1] present a generic JIT infrastructure for research. While they use profiling to aid in optimization, they use counters to identify hot methods. Similarly, Suganuma et al. [133] use counters to optimize from their lowest optimization level to their middle optimization level. They use profiling of frequently executing PCs to determine hot candidates for optimizing to their highest optimization level.

6.4 Automatic Memory Management

Much of this dissertation focuses on improving dynamic language performance by improving inefficiencies in automatic memory management. There is much work in improving garbage collection algorithms and detailed evaluation of their performance. However, much of the previous work focuses on software-only optimizations. The following sections instead discuss work related to HW/SW co-optimization of memory management.

6.4.1 Nursery Sizing

Some previous work discuss the trade-off of cache performance and garbage collection overhead. Some work suggests that the nursery should fit in the cache [145, 114], while

other work suggests that a larger nursery is better [13].

Wilson et al. [145] explore how different cache designs affect the performance of generational garbage collection. They conclude that careful attention to memory hierarchy issues can significantly decrease the performance impact of garbage collection. They point out that as caches get larger, the best way to achieve high performance is to make the young generation (nursery) fit within the last-level cache.

Reddy et al. [114] similarly propose to pin the nursery, which contains the most recently allocated objects, in the last-level cache. As a result, they improve the cache performance and overall execution time of the program as well as garbage collection pause times.

On the other hand, an in-depth study by Blackburn et al. [13] on the micro-architectural behaviors of various garbage collection algorithms suggests that a larger nursery size will result in better performance. They find that sizing the nursery larger than the last-level cache results in lower garbage collection overhead without significant change to the application performance.

This dissertation shows that the performance of small nurseries that fit in the cache can be further improved by co-optimizing both hardware and software. Nursery sizing should be done in a manner that considers application-specific characteristics, runtime configuration, and cache performance. In some cases, a smaller nursery will be better while in other cases, a larger nursery could be better for overall performance. Nursery sizing is extended to multiple concurrent applications to show that proper nursery sizing is essential for good performance when multiple applications share a cache.

6.4.2 Cache Installation

The idea of directly installing a cache line without going to memory has been studied previously, but existing proposals are not suitable for dynamic languages. PowerPC has an `dcbz` instruction that that can install a cache line directly [66]. However, to use this instruction, software needs to be aware of the cache line size of the underlying architecture and explicitly install cache lines one at a time. This limits the portability and applicability.

Other studies focus on reducing cache misses from stores to newly-allocated memory regions in the context of C and C++. Lewis et al. propose a hardware table to explicitly track mallocs and install newly-allocated cache lines [83]. This works well for C and C++ because dynamically allocated objects are often larger than one cache line. In contrast, dynamic languages frequently allocate small objects, which are smaller than a cache line. The malloc table cannot be used to install a cache line since no assumption can be made on neighboring words in the same cache line. In this dissertation, the proposed tracking table for invalid memory ranges installs cache lines for small objects by assuming later words are unallocated. The same tracking table can additionally be used to reduce cache pollution and eliminate unnecessary write-backs.

Sartor et al. [123] describe using cache installation and scrubbing instructions in the context of reducing DRAM traffic and energy. They use cache installation instructions to eliminate useless read traffic for nursery allocation and scrubbing instructions to invalidate or deprioritize dead cache lines to reduce dead write traffic. Their solution relies on ISA instructions similar to the PowerPC `dcbz` instruction and requires software to be aware of the cache line size. They overcome the limitations of using cache install instructions by installing 32kB regions at a time. This can result in unnecessary cache pollution. The work on optimizing single-application memory management in this dis-

sertation considers nursery sizes that are many times larger than the size of the last-level cache. Scrubbing such nursery ranges after each nursery collection would be ineffective. Instead, partial tracing is used to identify invalid cache lines at more frequent intervals than nursery collection, and hardware is used to efficiently install and "scrub" cache lines on-demand.

Hu and John [63] and Rui et al. [120] proposed store fill buffer designs where they direct store misses to a buffer and only retrieve the cache line from memory if either the cache line is evicted from the buffer before being fully written to or a word is read before it is written. If a full cache line is written in the buffer, the cache line is directly installed into the cache. These designs can work without any information about memory allocation because they only consider the stores to missed cache lines. The store fill buffer can reduce unnecessary memory reads when the initialization of a full cache line happens within a short period. However, the buffer is shared among all stores and an entry may be evicted before its fully initialized when objects are small. The work on optimizing single-application memory management in this dissertation uses software to precisely tell the hardware the areas of memory that are newly allocated. The proposed tracker can also be used for write-back reduction and pollution control, while the store fill buffer can be used only for cache installation.

Yang et al. [147] were the first to identify and present a detailed study on the performance impact of zeroing in modern managed languages on recent Intel processors. They show that existing options of zeroing, whether zeroing in bulk or during object allocation, have different trade-offs but similar performance impacts. By using existing cache-bypassing store instructions in the x86 architecture to perform bulk zeroing, they are able to improve the overall performance of the program. This dissertation similarly finds that object initialization can have high impact on performance if the nursery does

not fit in the cache. In this dissertation, multiple solutions are presented including dynamically adjusting nursery size and using cache installation to load nursery cache lines on-demand as objects are initialized. The bulk zeroing technique in previous work is complementary and can be used during garbage collection to further reduce the overhead of garbage collection.

Zhang et al. [151] identified a strong correlation between the object allocation rate and the memory bus write traffic in partially scalable programs written for Java. They conclude that scalability and performance are limited by the object allocation rate on multi-processor platforms resulting in an "allocation wall." This dissertation confirms their initial experiments and shows that the "allocation wall" can be overcome by directly initializing cache lines without reading from memory using *cache installation*. This dissertation further shows that the effect is amplified when multiple programs run concurrently and proposes nursery sizing schemes to reduce the performance impact. While the dissertation focuses on dynamic languages, the techniques should be applicable to other garbage collected languages. The prior work suggests that the cache optimizations are also important for static-but-managed languages such as Java.

6.4.3 GC Tuning

Previous studies have proposed to improve the locality by shaping memory accesses with garbage collection [64, 61, 79, 29]. For example, Huang et al. [64] 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 a problem that is addressed in this dissertation.

Some previous work considers automatically tuning garbage collection and runtime parameters [140, 71, 82, 130, 129, 128, 21]. Jayasena et al. [71] use an autotuning algorithm to tune up to 600 different JVM parameters related to garbage collection, just-in-time compilation, and class loading. Cameron et al. [21] apply economic theory to perform holistic tuning of heap sizes of multiple applications. This dissertation is the first to consider nursery size in the context multiple applications and show that the nursery size can be tuned for significant performance improvement.

6.4.4 GC Resource Contention

To the best of the author's knowledge, this dissertation includes the first study on the impact of cache sharing on multiple managed-language applications.

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 [4] 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 [136], control theory [144], and forcing garbage collection to limit unnecessary memory usage [59]. In addition, Hertz et al. [58] describe a garbage collection algorithm that works with the virtual memory manager to guide page eviction decisions and reduce overall paging. This dissertation 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 [101] 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. Eizenberg et al. [44] discuss cache coherence issues in JVM as a result of true and false sharing of memory locations in private caches. Their Remix tool can detect both true and false sharing and perform repairs for false sharing to improve application performance.

There has been much work on implementations of garbage collection on multi-threaded applications [88, 113, 5, 152, 38, 149]. The designs follow Doligez and Leroy's work [38] 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. [113] extend this design to setup the heap hierarchy for further optimization in functional languages. More recent work focuses on NUMA-aware data placement in context of automatic memory management [50, 8, 51, 152]. Gidra et al. [51] propose NumaGiC, which attempts to minimize remote accesses to memory during garbage collection through a distributed algorithm.

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

6.4.5 Accelerating Garbage Collection

Previous studies have also proposed hardware support for garbage collection. Some of them aim to accelerate the computational overhead of reference counting [146, 74]. Others use a hardware co-processor to achieve more predictable garbage collection in a real-time setting [124, 92, 93, 100]. Maas et al. [87] propose using hardware to accelerate the most commonly used functions in tracing garbage collection.

6.4.6 Reducing Write-backs

Some previous studies optimize garbage collection to reduce write-backs from caches [106, 126, 127]. They use a garbage bit to track garbage data in the cache. Garbage cache lines are not written back and can be replaced before other valid cache lines. In this dissertation, a novel partial tracing technique is proposed that decouples tracing a small part of a nursery from full garbage collection.

6.4.7 Tracking Memory Regions

The tracking hardware for invalid memory regions discussed in this dissertation is similar to the Range Cache [138], which is designed to store security tags for a range of addresses. The tracking hardware in this dissertation only needs to indicate whether a range of addresses is invalid, which allows the hardware to be greatly simplified. In addition, overlapping ranges do not need to be kept, which allows for fast (single-cycle) hits for lookups.

CHAPTER 7

FUTURE WORK

This dissertation identifies multiple sources of overhead in dynamic languages. While the focus is on optimizing automatic memory management, future work can build off of the studies to develop approaches to improving the various other sources of overhead. In order for dynamic languages to be competitive with static languages in performance, multiple inefficiencies should be optimized using a HW/SW co-optimization approach. This chapter briefly discusses some possible ideas for future improvement.

7.1 C Function Call Overhead

In the study on sources of overhead (Chapter 3), C function call overhead is identified as a large source of overhead that has not been previously identified. Function calls are used to better organize code and reduce the amount of code that the programmer needs to write and debug. At the processor level, they make management of registers easier, since registers can be used according to the calling convention.

One research direction would be exploring how to reduce this overhead. For example, could existing work on interprocedural optimizations based on feedback-directed optimizations [75] be applied to interpreters? Information about the interpreter structure can be use to guide the optimizations. In addition, can hardware be modified so that the cost of calling functions is reduced? For example, hardware may be able to further reduce the overhead of saving state before a function call and restoring it after the function call.

7.2 When to JIT

The study on JIT thresholds (Section 3.5) shows that choosing an appropriate JIT threshold can significantly improve program performance. This direction can be extended to further study the characteristics of loops that make them better candidates for optimization and at what point (in terms of threshold) a loop should be optimized.

For example, JIT optimizes for the common path. Loops with many branches and control flow divergence may not be good candidates for optimization. Branch predictor information can be used to collect statistics on a per-loop basis to determine which loops have high branch predictor accuracy (indicating that the program is executing predictable paths) and that information can be incorporated into the decision of whether or not to JIT.

Even in cases where there is little divergence in control flow, it may not be appropriate to optimize a loop unless it is expected to execute for a long time. Can loops that execute for long enough to amortize overhead be better predicted by inferring the iteration counts of the loops?

Finally, there may be ways to improve the performance of the JIT compilation through hardware acceleration so that the overhead of optimizing code is reduced. As a result, more code could be optimized and there would be more performance improvements from using JIT.

7.3 Further Improving Automatic Memory Management

This dissertation discusses multiple ways that the automatic memory management can be optimized for dynamic languages to improve performance. Future work could build on the approaches in this dissertation for further performance improvements.

The dynamic nursery sizing scheme can be further improved to better predict nursery sizes. For example, can machine learning be applied to train a predictor? The challenge in this case would be to do it with low overhead and with a limited number of features. In addition, other hardware metrics can be used to inform the decision. Finally, some work would be needed to properly train the models to cover various program behaviors.

The dissertation also presents a study on scheduling for multiple applications running concurrently (Section 5.5). A more comprehensive study with more groups of benchmarks should be done to get a better understanding of the potential performance improvements. Which characteristics of applications make them good for running together and which characteristics lead to poor performance? The study shows that balancing benchmarks with large nurseries and small nurseries usually does quite well.

Finally, the current analysis of the cache performance and garbage collection trade-off curve assumes that memory latency is constant. With the current advances in memory architecture, non-uniform memory access (NUMA) machines are becoming increasingly prevalent. Another research direction would be to further explore how these machines change the trade-offs and techniques for achieving good performance.

CHAPTER 8

CONCLUSION

As software becomes more complex and the costs of developing and maintaining code increase, dynamic programming languages such as Python are becoming more desirable alternatives to traditional static languages such as C. This dissertation studies the sources of overhead in these languages and proposes optimizations for improving their performance.

First, an extensive characterization of Python at various levels is performed to provide insights into new opportunities for optimizations in dynamic languages. When looking at the sources of overhead, C function call overhead is identified as a major source of overhead in addition to other sources of overhead identified by previous work. In studying the interaction of the runtime with the underlying hardware, PyPy with JIT is found to be sensitive to cache and memory parameters.Through a further investigation, it is found that nursery size needs to be tuned at the application-specific level, considering the runtime and underlying hardware.

Second, single-application optimizations are discussed for memory management in dynamic languages that consider both the hardware and software together. In particular, it is shown that a large nursery size can be used to significantly reduce garbage collection overhead of a single application if the impact on cache performance can be controlled. A new invalid memory region tracker and partial tracing algorithm are introduced to address the cache performance issue.

Finally, the impact of cache sharing on program performance is studied. Nursery-sizing strategies are found to significantly impact the performance of concurrently running applications. A model is developed to understand the impact of cache size on

nursery-sizing and is applied to an offline static sizing scheme and an online dynamic sizing scheme. The results show that large improvements in the throughput can be achieved and that using automatic memory management to reshape memory access patterns can be an effective tool in addressing cache contention and improving overall performance of concurrent programs.

While Python is primarily evaluated for most of the studies, the main results from the studies should be applicable to other dynamic languages. As dynamic languages continue to see widespread uses, HW/SW co-optimization of these languages will be essential to make them viable high-performance alternatives to traditional static languages.

APPENDIX A

MODEL OF GARBAGE COLLECTION EXECUTION TIME

In this section, a first order model is described of the total garbage collection time, denoted T_{GC}. The model only considers nursery collection of the young space and not the full garbage collection of the old space. The total garbage collection time can be broken down into a series of garbage collection invocations:

$$T_{GC} = N * t_{GC} \tag{A.1}$$

where N is the number of times garbage collection is run and t_{GC} is the average time per invocation.

Garbage collection is triggered when the memory allocated by the mutator matches the nursery size:

$$N = \frac{S_{PA}}{S_N} \tag{A.2}$$

where S_{PA} is the total number of bytes allocated by the mutator in the nursery and S_N is the nursery size.

The time it takes to run a garbage collection invocation can be expressed as:

$$t_{GC} = t_{setup} + t_{cleanup} + (t_{trace} + t_{move}) N_{live} \tag{A.3}$$

where t_{setup} and $t_{cleanup}$ are fixed times for each garbage collector invocation and t_{trace} and t_{move} are the time it takes to trace and move a single object. N_{live} is the number of objects that need to be moved.

N_{live} can be modeled as an exponential decay function:

$$N_{live} = \frac{S_N}{S_O} e^{-\lambda t} \tag{A.4}$$

where S_O is the average size of an object and λ is the death rate of objects.

Since garbage collection is performed at intervals, t is set to be the interval between garbage collection invocations:

$$t = \frac{S_N}{R_{PA}} \tag{A.5}$$

where R_{PA} is the rate the mutator allocates nursery space in *bytes/cycle*.

Through substitution, the above equations are combined to get the following:

$$T_{GC} = \frac{S_{PA}}{S_N}\left[t_{setup} + t_{cleanup} + (t_{trace} + t_{move})\frac{S_N}{S_O}e^{-\lambda\frac{S_N}{R_{PA}}}\right] \tag{A.6}$$

In the equation, there are some variables that are dependent on the application. λ is fully dependent on the application that is running. S_{PA}, S_O, R_{PA} are also dependent on the application that is running, but minor adjustments can be made by changing the object space implementation in the runtime. Decreasing the size of the object will decrease the total space allocated by the program and the allocation rate. This will result in an overall reduction in the garbage collection time.

In terms of the garbage collector implementation, any of t_{setup}, $t_{cleanup}$, t_{trace}, t_{move}, or S_N can be improved. Improving the first four would require optimization of the garbage collector. S_N can easily be adjusted before each run or even at runtime.

By increasing the nursery size (i.e. S_N), t_{setup} and $t_{cleanup}$ is amortized over a longer time period. In other words, if the nursery size is increased by M, the contribution over the same time period becomes $\left(t_{setup} + t_{cleanup}\right)/M$.

Interestingly, the trace and move times are not affected by the nursery size in the same way. The larger nursery size makes the magnitude of the exponent larger indicating a larger decay constant. This means that more live objects will die between garbage collections with a larger nursery size. As a result, the contribution from tracing and moving will be smaller.

The observations from the equation are intuitive. A larger nursery allows for garbage collection to run less frequently and there will be less live objects to trace and move during each execution. As a result, the total garbage collection overhead will decrease.

BIBLIOGRAPHY

[1] Ali-Reza Adl-Tabatabai, Jay Bharadwaj, Dong-Yuan Chen, Anwar Ghuloum, Vijay Menon, Brian Murphy, Mauricio Serrano, and Tatiana Shpeisman. The StarJIT compiler: A dynamic compiler for managed runtime environments. 2003.

[2] Gerald Aigner and Urs Hölzle. Eliminating virtual function calls in C++ programs. In *Proceedings of the 10th European Conference on Object-Oriented Programming (ECOOP)*, pages 142–166. Springer, 1996.

[3] Karim Ali and Ondřej Lhoták. Averroes: Whole-program analysis without the whole program. In *Proceedings of the 27th European Conference on Object-Oriented Programming (ECOOP)*, pages 378–400. Springer, 2013.

[4] Raphael Alonso and Andrew W. Appel. An advisor for flexible working sets. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 153–162. ACM, 1990.

[5] Todd A. Anderson. Optimizations in a private nursery-based garbage collector. In *Proceedings of the International Symposium on Memory Management (ISMM)*, pages 21–30. ACM, 2010.

[6] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, June 1987.

[7] Matthew Arnold, Michael Hind, and Barbara G. Ryder. Online feedback-directed optimization of java. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 111–129. ACM, 2002.

[8] Sven Auhagen, Lars Bergstrom, Matthew Fluet, and John Reppy. Garbage collection for multicore numa machines. In *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC)*, pages 51–57. ACM, 2011.

[9] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 324–341. ACM, 1996.

[10] David Francis Bacon. *Fast and Effective Optimization of Statically Typed Object-oriented Languages*. PhD thesis, University of California, Berkeley, 1997.

[11] Gergö Barany. Python interpreter performance deconstructed. In *Proceedings of the Workshop on Dynamic Languages and Applications (Dyla)*, pages 5:1–5:9. ACM, 2014.

[12] R. D. Barnes, E. M. Nystrom, M. C. Merten, and W. W. Hwu. Vacuum packing: Extracting hardware-detected program phases for post-link optimization. In *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 233–244. IEEE, 2002.

[13] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. 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)*, pages 25–36. ACM, 2004.

[14] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice & Experience*, 18(9):807–820, September 1988.

[15] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 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)*, pages 18–25. ACM, 2009.

[16] Katie Bouwkamp. The 9 most in-demand programming languages of 2016, 2016. `http://www.codingdojo.com/blog/9-most-in-demand-programming-languages-of-2016/`.

[17] Browserbench. JetStream 1.1, 2017. `http://browserbench.org/JetStream/`.

[18] Stefan Brunthaler. Speculative staging for interpreter optimization. *CoRR*, abs/1310.2300, 2013.

[19] Xing Cai, Hans Petter Langtangen, and Halvard Moe. On the performance of the Python programming language for serial and parallel scientific computations. *Scientific Programming*, 13(1):31–56, 2005.

[20] Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in C++ programs. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 397–408. ACM, 1994.

[21] Callum Cameron, Jeremy Singer, and David Vengerov. The judgment of Forseti: Economic utility for dynamic heap sizing of multiple runtimes. In *Proceedings of*

the International Symposium on Memory Management (ISMM), pages 143–156. ACM, 2015.

[22] Huaxiong Cao, Naijie Gu, Kaixin Ren, and Yi Li. Performance research and optimization on CPython's interpreter. In *Proceedings of the 2015 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 435–441. IEEE, 2015.

[23] Kevin Casey, M Anton Ertl, and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(6):37, 2007.

[24] Stephen Cass. The 2018 top ten programming languages, July 2018. `https://spectrum.ieee.org/at-work/innovation/the-2018-top-programming-languages`.

[25] Craig Chambers, Jeffrey Dean, and David Grove. Whole-program optimization of object-oriented languages. 1996.

[26] Mostafa Chandra, Nagy Krintz, Calin Cascaval, David Edelsohn, Priya Nagpurkar, and Peng Wu. Understanding the potential of interpreter-based optimizations for Python. Technical Report 2010-14, UC Santa Barbara Computer Science, August 2010. `https://www.cs.ucsb.edu/research/tech-reports/2010-14`.

[27] Po-Yung Chang, Eric Hao, and Yale N. Patt. Target prediction for indirect jumps. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA)*, pages 274–283. ACM, 1997.

[28] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, November 1970.

[29] Trishul M Chilimbi and James R Larus. Data structure partitioning with garbage collection to optimize cache utilization, November 2001. US Patent 6,321,240.

[30] Jiho Choi, Thomas Shull, Maria J. Garzaran, and Josep Torrellas. Shortcut: Architectural support for fast object access in scripting languages. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 494–506. ACM, 2017.

[31] Python. `http://www.python.org/`.

[32] Jack W. Davidson and Anne M Holler. Subprogram inlining: A study of its effects on program execution time. *IEEE Transactions on Software Engineering*, 18(2):89–102, 1992.

[33] Jeffrey Dean, Greg DeFouw, David Grove, Vassily Litvinov, and Craig Chambers. Vortex: An optimizing compiler for object-oriented languages. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 83–100. ACM, 1996.

[34] Greg DeFouw, David Grove, and Craig Chambers. Fast interprocedural class analysis. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 222–236. ACM, 1998.

[35] David Detlefs and Ole Agesen. Inlining of virtual methods. In *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP)*, pages 258–278. Springer, 1999.

[36] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Using types to analyze and optimize object-oriented programs. *ACM Trans. Program. Lang. Syst.*, 23(1):30–72, January 2001.

[37] Amer Diwan, J. Eliot B. Moss, and Kathryn S. McKinley. Simple and effective analysis of statically-typed object-oriented programs. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 292–305. ACM, 1996.

[38] Damien Doligez and Xavier Leroy. 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)*, pages 113–123. ACM, 1993.

[39] Gem Dot, Alejandro Martinez, and Antonio Gonzalez. Analysis and optimization of engines for dynamically typed languages. In *Proccedings of the 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 41–48. IEEE, 2015.

[40] Karel Driesen. *Software and Hardware Techniques for Efficient Polymorphic Calls*. PhD thesis, University of California, Santa Barbara, 1999.

[41] Karel Driesen and Urs Hölzle. The direct cost of virtual function calls in C++. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 306–323. ACM, 1996.

[42] Karel Driesen and Urs Hölzle. The cascaded predictor: Economical and adaptive branch target prediction. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 249–258. IEEE, 1998.

[43] Karel Driesen and Urs Hölzle. Multi-stage cascaded prediction. In *Proceedings of the 5th International Euro-Par Conference on Parallel Processing*, pages 1312–1321. Springer, 1999.

[44] Ariel Eizenberg, Shiliang Hu, Gilles Pokam, and Joseph Devietti. 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)*, pages 251–265. ACM, 2016.

[45] M. Anton Ertl and David Gregg. The behavior of efficient virtual machine interpreters on modern architectures. In *Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, Euro-Par '01, pages 403–412, London, UK, UK, 2001. Springer-Verlag.

[46] M. Anton Ertl and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 278–288, New York, NY, USA, 2003. ACM.

[47] M Anton Ertl and David Gregg. The structure and performance of efficient interpreters. *Journal of Instruction-Level Parallelism*, 5:1–25, 2003.

[48] Muhammad Umar Farooq, Lei Chen, and Lizy Kurian. Value based BTB indexing for indirect jump prediction. In *Proceedings of the 16th International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–11. IEEE, 2010.

[49] Robert R. Fenichel and Jerome C. Yochelson. A LISP garbage-collector for virtual-memory computer systems. *Commun. ACM*, 12(11):611–612, November 1969.

[50] Lokesh Gidra, Gaël Thomas, Julien Sopena, and Marc Shapiro. 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)*, pages 229–240. ACM, 2013.

[51] Lokesh Gidra, Gaël Thomas, Julien Sopena, Marc Shapiro, and Nhan Nguyen. NumaGiC: A garbage collector for big data on big NUMA machines. In *Pro-*

ceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 661–673. ACM, 2015.

[52] Google. Chrome V8, 2018. `https://developers.google.com/v8/`.

[53] Isaac Gouy. The computer language benchmarks game, 2019. `https://benchmarksgame-team.pages.debian.net/benchmarksgame/`.

[54] David Grove and Craig Chambers. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.*, 23(6):685–746, November 2001.

[55] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 108–124. ACM, 1997.

[56] David Paul Grove. *Effective Interprocedural Optimization of Object-oriented Languages*. PhD thesis, University of Washington, 1998.

[57] Kim Hazelwood and David Grove. Adaptive online context-sensitive inlining. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO)*, pages 253–264. IEEE, 2003.

[58] Matthew Hertz, Yi Feng, and Emery D. Berger. Garbage collection without paging. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 143–153. ACM, 2005.

[59] Matthew Hertz, Stephen Kane, Elizabeth Keudel, Tongxin Bai, Chen Ding, Xiaoming Gu, and Jonathan E. Bard. Waste not, want not: Resource-based garbage collection in a shared environment. In *Proceedings of the International Symposium on Memory Management (ISMM)*, pages 65–76. ACM, 2011.

[60] Ruben Heynssens. Performance analysis and benchmarking of Python, a modern scripting language. Master's thesis, Universiteit Gent, Belgium, 2014.

[61] Martin Hirzel. Data layouts for object-oriented programs. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 265–276. ACM, 2007.

[62] Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 326–336. ACM, 1994.

[63] Shiwen Hu and Lizy John. Avoiding store misses to fully modified cache blocks. In *Proceedings of the 25th International Performance, Computing, and Communications Conference (IPCCC)*, pages 289–296. IEEE, 2006.

[64] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J Eliot B. Moss, Zhenlin Wang, and Perry Cheng. 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)*, pages 69–80. ACM, 2004.

[65] Charles Humble. Twitter's shift from Ruby to Java helps it survive US election, November 2012. `https://www.infoq.com/news/2012/11/twitter-ruby-to-java`.

[66] IBM. *PowerPC Microprocessor Family: The Programming Environments*. IBM Microelectronics, Motorola Corporation, 1994.

[67] Berkin Ilbeyi, Carl Friedrich Bolz-Tereick, and Christopher Batten. Cross-layer workload characterization of meta-tracing JIT VMs. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*, pages 97–107. IEEE, 2017.

[68] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, and Toshio Nakatani. A study of devirtualization techniques for a java just-in-time compiler. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 294–310. ACM, 2000.

[69] Mohamed Ismail and G Edward Suh. Hardware-software co-optimization of memory management in dynamic languages. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management (ISMM)*, pages 45–58. ACM, 2018.

[70] Mohamed Ismail and G Edward Suh. Quantitative overhead analysis for python. In *Proceedings of the 2018 International Symposium on Workload Characterization (IISWC)*, pages 36–47. IEEE, 2018.

[71] S. Jayasena, M. Fernando, T. Rusira, C. Perera, and C. Philips. Auto-tuning the java virtual machine. In *IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 1261–1270, May 2015.

[72] Jose A. Joao, Onur Mutlu, Hyesoon Kim, Rishi Agarwal, and Yale N. Patt. Improving the performance of object-oriented languages with dynamic predication of indirect jumps. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 80–90. ACM, 2008.

[73] José A Joao, Onur Mutlu, Hyesoon Kim, and Yale N Patt. Dynamic predication of indirect jumps. *IEEE Computer Architecture Letters*, 6(2):25–28, 2007.

[74] José A Joao, Onur Mutlu, and Yale N Patt. Flexible reference-counting-based hardware acceleration for garbage collection. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 418–428. ACM, 2009.

[75] Teresa Johnson, Mehdi Amini, and Xinliang David Li. ThinLTO: Scalable and incremental LTO. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 111–121. IEEE, 2017.

[76] John Kalamatianos and David R. Kaeli. Predicting indirect branches via data compression. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 272–281. IEEE, 1998.

[77] Channoh Kim, Sungmin Kim, Hyeon Gyu Cho, Dooyoung Kim, Jaehyeok Kim, Young H. Oh, Hakbeom Jang, and Jae W. Lee. Short-circuit dispatch: Accelerating virtual machine interpreters on embedded processors. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, pages 291–303. IEEE, 2016.

[78] Hyesoon Kim, José A. Joao, Onur Mutlu, Chang Joo Lee, Yale N. Patt, and Robert Cohn. VPC prediction: Reducing the cost of indirect branches via hardware-based dynamic devirtualization. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, pages 424–435. ACM, 2007.

[79] Thomas Kistler and Michael Franz. Automated data-member layout of heap objects to improve memory-hierarchy performance. *ACM Trans. Program. Lang. Syst.*, 22(3):490–505, May 2000.

[80] Hans Petter Langtangen and Xing Cai. On the efficiency of Python for high-performance computing: A case study involving stencil updates for partial dif-

ferential equations. In *Modeling, Simulation and Optimization of Complex Processes*, pages 337–357. Springer, 2008.

[81] Patrick Lee. Open sourcing our Go libraries, July 2014. `https://blogs.dropbox.com/tech/2014/07/open-sourcing-our-go-libraries`.

[82] Philipp Lengauer and Hanspeter Mössenböck. 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)*, pages 111–122. ACM, 2014.

[83] Jarrod A Lewis, Bryan Black, and Mikko H Lipasti. Avoiding initialization misses to the heap. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, pages 183–194. IEEE, 2002.

[84] David Xinliang Li, Raksit Ashok, and Robert Hundt. Lightweight feedback-directed cross-module optimization. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 53–61. ACM, 2010.

[85] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM*, 26(6):419–429, June 1983.

[86] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 190–200. ACM, 2005.

[87] Martin Maas, Krste Asanović, and John Kubiatowicz. A hardware accelerator for tracing garbage collection. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 138–151. IEEE, 2018.

[88] Simon Marlow and Simon Peyton Jones. Multicore garbage collection with local heaps. In *Proceedings of the International Symposium on Memory Management (ISMM)*, pages 21–32. ACM, 2011.

[89] Jason Mccandless and David Gregg. Compiler techniques to improve dynamic branch prediction for indirect jump and call instructions. *ACM Trans. Archit. Code Optim.*, 8(4):24:1–24:20, January 2012.

[90] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM*, 3(4):184–195, April 1960.

[91] Daniel S. McFarlin and Craig Zilles. Bungee jumps: Accelerating indirect branches through HW/SW co-design. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, pages 370–382. ACM, 2015.

[92] Matthias Meyer. A novel processor architecture with exact tag-free pointers. *IEEE Micro*, 24(3):46–55, 2004.

[93] Matthias Meyer. An on-chip garbage collection coprocessor for embedded real-time systems. In *Proceedings of the 11th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 517–524. IEEE, 2005.

[94] Sun Microsystems. Memory management in the Java HotSpot virtual machine, 2006. `http://www.oracle.com/technetwork/java/javase/tech/memorymanagement-whitepaper-1-150020.pdf`.

[95] Nevena Milojkovic. Towards cheap, accurate polymorphism detection. *Pre-Proceedings of the 7th International Seminar Series on Advanced Techniques & Tools for Software Evolution (SATToSE)*, page 54, 2014.

[96] Markus Mock, Craig Chambers, and Susan J. Eggers. Calpa: A tool for automating selective dynamic compilation. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 291–302. ACM, 2000.

[97] Gilles Muller, Bárbara Moura, Fabrice Bellard, and Charles Consel. JIT vs offline compilers: Limits and benefits of bytecode compilation. December 1996.

[98] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Cacti 6.0: A tool to model large caches. *HP laboratories*, pages 22–31, 2009.

[99] Malek Musleh and Vijay Pai. Architectural characterization of client-side JavaScript workloads & analysis of software optimizations. Technical Report 467, Purdue University Department of Electrical and Computer Engineering, 2015. `https://docs.lib.purdue.edu/ecetr/467/`.

[100] Kelvin D Nilsen and William Schmidt. System and hardware module for incremental real time garbage collection and memory management, September 1996. US Patent 5,560,003.

[101] Takeshi Ogasawara. Scalability limitations when running a java web server on a chip multiprocessor. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference (SYSTOR)*, pages 6:1–6:10. ACM, 2010.

[102] Takeshi Ogasawara. Workload characterization of server-side JavaScript. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*, pages 13–21. IEEE, 2014.

[103] Stephen O'Grady. The RedMonk programming language rankings: January 2019, March 2019. `https://redmonk.com/sogrady/2019/03/20/language-rankings-1-19/`.

[104] Hannes Payer and Ross McIlroy. Getting garbage collection for free, 2015. `https://v8project.blogspot.com/2015/08/getting-garbage-collection-for-free.html`.

[105] Igor Pechtchanski and Vivek Sarkar. Dynamic optimistic interprocedural analysis: A framework and an application. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 195–210. ACM, 2001.

[106] Chih-Jui Peng and Gurindar S Sohi. *Cache memory design considerations to support languages with dynamic heap allocation*. University of Wisconsin-Madison. Computer Sciences Department, 1989.

[107] Russell Power and Alex Rubinsteyn. How fast can we make interpreted Python? *CoRR*, abs/1306.6047, 2013.

[108] The PyPy Project. Garbage collection in PyPy, 2014. `https://pypy.readthedocs.io/en/release-2.4.x/garbage_collection.html`.

[109] PyPerformance. Python performance benchmark suite, 2017. `http://pyperformance.readthedocs.io/`.

[110] PyTorch, 2019. `https://research.fb.com/downloads/pytorch/`.

[111] R. Radhakrishnan, N. Vijaykrishnan, L. K. John, A. Sivasubramaniam, J. Rubio, and J. Sabarinathan. Java runtime systems: characterization and architectural implications. *IEEE Transactions on Computers*, 50(2):131–146, February 2001.

[112] Ramesh Radhakrishnan, R. Radhakrishnany, Lizy K. John, Juan Rubio, L. K.

Johny, and N. Vijaykrishnan. Execution characteristics of just-in-time compilers. Technical Report TR-990717-01, University of Texas, 1999.

[113] Ram Raghunathan, Stefan K. Muller, Umut A. Acar, and Guy Blelloch. Hierarchical memory management for parallel programs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 392–406. ACM, 2016.

[114] Vimal K Reddy, Richard K Sawyer, and Edward F Gehringer. A cache-pinning strategy for improving generational garbage collection. In *International Conference on High-Performance Computing*, pages 98–110. Springer, 2006.

[115] Nick Roberts. Picking apart Stack Overflow: What bugs developers the most?, 2019. `https://www.globalapptesting.com/blog/picking-apart-stackoverflow-what-bugs-developers-the-most`.

[116] Erven Rohou, Bharath Narasimha Swamy, and André Seznec. Branch prediction and the performance of interpreters: Don't trust folklore. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 103–114. IEEE, 2015.

[117] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A cycle accurate memory system simulator. *IEEE Computer Architecture Letters*, 10(1):16–19, January 2011.

[118] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. Improving virtual function call target prediction via dependence-based pre-computation. In *Proceedings of the 13th International Conference on Supercomputing (ICS)*, pages 356–364. ACM, 1999.

[119] Rubinius. Concurrent garbage collection, June 2013. `https://github.com/rubinius/rubinius-archive/blob/cf54187d421275eec7d2db0abd5d4c059755b577/_posts/2013-06-22-concurrent-garbage-collection.markdown`.

[120] Hou Rui, Fuxin Zhang, and Weiwu Hu. A memory bandwidth effective cache store miss policy. In *Proceedings of the Asia-Pacific Conference on Advances in Computer Systems Architecture*, pages 750–760. Springer, 2005.

[121] Daniel Sanchez and Christos Kozyrakis. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, pages 475–486. ACM, 2013.

[122] Oliverio J Santana, Ayose Falcón, Enrique Fernández, Pedro Medina, Alex Ramírez, and Mateo Valero. A comprehensive analysis of indirect branch prediction. In *Proceedings of the International Symposium on High Performance Computing (HiPC)*, pages 133–145. Springer, 2002.

[123] Jennifer B. Sartor, Wim Heirman, Stephen M. Blackburn, Lieven Eeckhout, and Kathryn S. McKinley. Cooperative cache scrubbing. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT)*, pages 15–26. ACM, 2014.

[124] William J Schmidt and Kelvin D Nilsen. Performance of a hardware-assisted real-time garbage collector. *ACM SIGOPS Operating Systems Review*, 28(5):76–85, 1994.

[125] Timothy Sherwood, Suleyman Sair, and Brad Calder. Phase tracking and prediction. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, pages 336–349. ACM, 2003.

[126] Jonathan Shidal, Zachary Gottlieb, Ron K Cytron, and Krishna M Kavi. Trash in cache: Detecting eternally silent stores. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*, page 8. ACM, 2014.

[127] Jonathan Shidal, Ari J. Spilo, Paul T. Scheid, Ron K. Cytron, and Krishna M. Kavi. Recycling trash in cache. In *Proceedings of the 2015 International Symposium on Memory Management (ISMM)*, pages 118–130. ACM, 2015.

[128] Jose Simão and Luis Veiga. 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)*, pages 723–728. IEEE, 2012.

[129] Jeremy Singer, Richard E. Jones, Gavin Brown, and Mikel Luján. The economics of garbage collection. In *Proceedings of the International Symposium on Memory Management (ISMM)*, pages 103–112. ACM, 2010.

[130] Jeremy Singer, George Kovoor, Gavin Brown, and Mikel Luján. Garbage collection auto-tuning for Java mapreduce on multi-cores. In *Proceedings of the International Symposium on Memory Management (ISMM)*, pages 109–118. ACM, 2011.

[131] Gabriel Southern and Jose Renau. Overhead of deoptimization checks in the V8 JavaScript engine. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*, pages 1–10. IEEE, 2016.

[132] Aditya Srikanth. *Characterization and optimization of JavaScript programs for mobile systems*. PhD thesis, University of Texas at Austin, May 2013.

[133] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. A dynamic optimization framework for a Java just-in-time compiler. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 180–195. ACM, 2001.

[134] G. Edward Suh, Srinivas Devadas, and Larry Rudolph. Analytical cache models with applications to cache partitioning. In *Proceedings of the 15th International Conference on Supercomputing (ICS)*, pages 1–12. ACM, 2001.

[135] Ole Tange. *GNU Parallel 2018*. March 2018.

[136] Y. C. Tay, Xuanran Zong, and Xi He. An equation-based heap sizing rule. *Perform. Eval.*, 70(11):948–964, November 2013.

[137] Devesh Tiwari and Yan Solihin. Architectural characterization and similarity analysis of Sunspider and Google's V8 JavaScript benchmarks. In *Proceedings of the International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 221–232. IEEE, 2012.

[138] Mohit Tiwari, Banit Agrawal, Shashidhar Mysore, Jonathan Valamehr, and Timothy Sherwood. A small cache of large ranges: Hardware methods for efficiently searching, storing, and updating big dataflow tags. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 94–105. IEEE, 2008.

[139] Vladimir Uzelac, Aleksandar Milenkovic, Milena Milenkovic, and Martin Burtscher. Using branch predictors and variable encoding for on-the-fly program tracing. *IEEE Trans. Comput.*, 63(4):1008–1020, April 2014.

[140] David Vengerov. Modeling, analysis and throughput optimization of a generational garbage collector. In *Proceedings of the International Symposium on Memory Management (ISMM)*, pages 1–9. ACM, 2009.

[141] N. Vijaykrishnan and N. Ranganathan. Tuning branch predictors to support virtual method invocation in Java. In *Proceedings of the 5th Conference on USENIX Conference on Object-Oriented Technologies & Systems - Volume 5 (COOTS)*, pages 16–16. USENIX Association, 1999.

[142] Maira Wenzel, Alan Dawkins, Luke Latham, Tom Pratt, Mike Jones, Michal Ciechan, and Xaviex. Fundamentals of garbage collection, March 2017. `https://msdn.microsoft.com/en-us/library/ee787088(v=vs.110).aspx`.

[143] John Whaley. Partial method compilation using dynamic profile information. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 166–179. ACM, 2001.

[144] David R. White, Jeremy Singer, Jonathan M. Aitken, and Richard E. Jones. Control theory for principled heap sizing. In *Proceedings of the International Symposium on Memory Management (ISMM)*, pages 27–38. ACM, 2013.

[145] Paul R Wilson, Michael S Lam, and Thomas G Moher. Caching considerations for generational garbage collection. In *ACM SIGPLAN Lisp Pointers*, number 1, pages 32–42. ACM, 1992.

[146] David S Wise, Brian Heck, Caleb Hess, Willie Hunt, and Eric Ost. Research demonstration of a hardware reference-counting heap. *Lisp and Symbolic Computation*, 10(2):159–181, 1997.

[147] Xi Yang, Stephen M. Blackburn, Daniel Frampton, Jennifer B. Sartor, and Kathryn S. McKinley. Why nothing matters: The impact of zeroing. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 307–324. ACM, 2011.

[148] T. Yasue, T. Suganuma, H. Komatsu, and T. Nakatani. An efficient online path profiling framework for java just-in-time compilers. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 148–158. IEEE, September 2003.

[149] Tatu J Ylonen. Parallel garbage collection and serialization without per-object synchronization, August 2010. US Patent App. 12/388,543.

[150] Olivier Zendra, Dominique Colnet, and Suzanne Collin. Efficient dynamic dispatch without virtual function tables: The SmallEiffel compiler. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 125–141. ACM, 1997.

[151] Yi Zhao, Jin Shi, Kai Zheng, Haichuan Wang, Haibo Lin, and Ling Shao. 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), pages 361–376. ACM, 2009.

[152] Jin Zhou and Brian Demsky. Memory management for many-core processors with software configurable locality policies. In *Proceedings of the International Symposium on Memory Management (ISMM)*, pages 3–14. ACM, 2012.

[153] Yuhao Zhu, Daniel Richins, Matthew Halpern, and Vijay Janapa Reddi. Microarchitectural implications of event-driven server-side web applications. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, pages 762–774. ACM, 2015.

[154] Xiaotong Zhuang, Mauricio J. Serrano, Harold W. Cain, and Jong-Deok Choi. Accurate, efficient, and adaptive calling context profiling. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 263–271. ACM, 2006.