

Memory Management Techniques for *Dynamic Languages*

by

Joannah Nanjekye

Diploma in Aeronautical Engineering (Avionics), Kenya Aeronautical College, 2018
Bachelor of Science in Software Engineering, Makerere University, 2014

A Dissertation Submitted in Partial Fulfilment of
the Requirements for the Degree of

Doctor of Philosophy

In the Graduate Academic Unit of Computer Science

Supervisor(s): David Bremner, Ph.D., Faculty of Computer Science

Examining Board: Suprio Ray, Ph.D., Faculty of Computer Science
Michael Fleming, Ph.D., Faculty of Computer Science
Eduardo Castillo Guerra, Ph.D., Department of
Electrical & Computer Engineering

External Examiner: Richard Jones, Ph.D., School of Computing University
of Kent

This dissertation is accepted by the
Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

August, 2024

© Joannah Nanjekye, 2024

Abstract

Garbage collection as a field requires well-documented and reusable techniques as well as considerations for the design abstractions of programming languages. This dissertation focuses on *dynamically-typed languages*, whose design warrants specific studies and optimizations. The first theme of the dissertation is the use of the Eclipse OMR framework for the correct, portable, and language-independent implementation of high performing garbage collectors for the RPython Meta-tracing-JIT-based dynamic languages.

We extend the use of the framework-based garbage collectors to study the garbage collection cost for the same dynamic languages, highlighting the trade-off in performance between JIT tracing and garbage collection and thereby proposing a novel *optimal JIT trace sizing* solution.

We further address two problems related to the collection resizing and boxing overhead for dynamic languages. We propose a *calling context aware collection presizing* technique for the former and present *type-based stores*, a novel memory layout to optimize type polymorphism in collection data structures for the latter.

Dynamic languages often also provide foreign function interfaces; the Python language has a C API but challenges of pointer stability, lifetime complexity and memory model compatibility continue to receive little attention in the research community. We propose a garbage collection friendly Python FFI, we call *CyStck*, that combines a stack and light-weight handles along with migration tooling for extensions.

Dedication

To all who can roger this:

"If you're faced with a forced landing, fly the thing as far into the crash as possible." –

Bob Hoover

Acknowledgements

I am highly indebted to my advisor Dr. David Bremner and IBM project lead Aleksandar Micic for the supervision, revisions, guidance and expert technical discussions that allowed me to write this dissertation.

I am especially thankful to Dr. Bremner for allowing me pursue collaborations with the Australian National University (ANU) and Kings College London (KCL) that widened my knowledge on garbage collection and programming language migration. These collaborations paved paths for productive career progressions, so for that I am extremely grateful.

I want to thank Dr. Carl Freidrich Boltz for helping me understand the PyPy aspects. I thank Dr. Steve Blackburn for hosting me in his group at ANU, his research guidance and introducing me to people at ANU and beyond that I have immensely benefited from. I specifically acknowledge Dr. Martin Maas and Dr. Kathryn McKinley from Google for the discussions on stack height that shaped my implementation of the contributions in Chapter 6.

I also thank Dr. Laurence Tratt for hosting me in his group at KCL, and allowing me to learn by observation how to organically work on research as a craft, giving me hope in academia, for his patience in our collaboration and the academic career discussions.

This research was conducted within the IBM Centre for Advanced Studies–Atlantic, Faculty of Computer Science, University of New Brunswick. I am grateful for the

CAS–Atlantic and the Atlantic Canada Opportunities Agency (ACOA) through the Atlantic Innovation Fund (AIF) program in supporting our research. I thank the IBM project manager, Mr. Stephen MacKay for reviewing all papers and this dissertation, as well as Mr. DeVerne Jones for CAS–Atlantic infrastructure support.

I am grateful for further financial support from Google, Rust Foundation, NBIF STEM Social and Innovation and the University of New Brunswick.

Finally, I want to first thank the UNB examining committee, comprised of Dr. Fleming, Dr. Suprio and Dr. Bremner for the valuable feedback on my research right from when I switched to PhD in 2020. This guidance prepared me for what was a successful external defense. I thank Dr. Guerra for the presentation suggestions that improved the quality of this final document. Then I thank Dr. Richard Jones for the external in depth expert review and examination which improved significant technical aspects of this work.

Of course IBM CASA research group lab mates Georgiy Krylov, Harpreet Kaur, Maria Patrou, Nga Tran and many others I can not mention here were very helpful in different ways while I pursued my PhD, so am grateful to all.

Any mistakes in this dissertation are my sole responsibility! And can be your future research topics!

Table of Contents

Abstract	ii
Dedication	iii
Acknowledgments	iv
Table of Contents	vi
List of Tables	xiii
List of Figures	xvi
Abbreviations	xxi
1 Introduction	1
1.1 Dynamic Languages	1
1.2 Thesis	3
1.3 Contributions	4
1.3.1 Garbage Collection with Frameworks	5
1.3.2 Understanding the Garbage Collection Cost	5
1.3.3 Type-based Stores for Collections in Dynamic Languages	6
1.3.4 Calling Context Aware Presizing	7
1.3.5 Memory Management for Native Extensions	8
1.4 Structure	9
1.5 Presentation of Evaluation Results	9

1.6	Publications	10
2	Memory Management and Dynamic Languages	14
2.1	Garbage Collection	15
2.1.1	Categorizing Garbage Collectors	15
2.1.2	Overview of Definitions	17
2.1.3	Garbage Collection Strategies	19
2.2	Garbage Collection Frameworks	22
2.2.1	Eclipse OMR	22
2.2.2	GC-as-a-Service Library	24
2.2.3	The Boehm-Demers-Weiser Conservative Garbage Collector	25
2.2.4	MMTk	25
2.3	Garbage Collection in Python	28
2.3.1	CPython	28
2.3.2	PyPy	29
2.4	Memory Layout of Types in Dynamic Languages	34
2.4.1	Terminology	34
2.4.2	Schemes for Memory Layout of Objects	35
2.4.3	Specialized Optimizations for Collections	37
2.5	Collection Presizing Schemes for Dynamic Languages	40
2.5.1	Mementos	40
2.5.2	Recycling	41
2.5.3	Lazy Creation	41
2.5.4	Language-specific Approaches	42
2.6	Memory Management for Native Extensions	42
2.6.1	Union Types	43
2.6.2	Handles	44
2.6.3	The Abstract Stack	45

2.6.4	The Proxy Object Model	46
3	Eclipse OMR-based Garbage Collection for Meta-tracing-JIT-based Dynamic Languages	47
3.1	Introduction	47
3.2	Design and Implementation	49
3.3	Software Engineering Metrics	53
3.3.1	Methodology	53
3.3.2	Raw Measurements	53
3.3.3	Halstead Measurements	54
3.3.4	LCOM and Maintainability Index	56
3.4	Performance Evaluation	57
3.4.1	PyPy	58
3.4.2	RPython-based VMs	60
3.5	Extending Eclipse-OMR-based Garbage collection	61
3.6	Related Work	63
4	The Garbage Collection Cost For Meta-Tracing JIT-based Dynamic Languages	65
4.1	Introduction	66
4.2	Language-independent GC Workload Characterization	67
4.3	The Garbage Collection Cost for Python Applications	68
4.3.1	Real-world Workloads	69
4.3.2	Experiment Setup	70
4.3.3	Application-specific Analysis	71
4.3.4	Nursery Sizing and Cache Performance	78
4.4	JIT Tracing and Garbage Collection	79
4.4.1	JIT Tracing Overview	79

4.4.2	The Garbage Collection Cost of Tracing	82
4.5	DTS: Optimal JIT Trace Sizing for Virtual Machines	85
4.5.1	Overview	86
4.5.2	Trace Estimation	87
4.5.3	GC Time Estimation	90
4.5.4	Mutator Time Estimation	90
4.5.5	Choosing the Optimal Trace Size	91
4.5.6	Implementation	92
4.5.7	Evaluation	94
4.6	Related Work	96
5	Type-based Stores for Collections in Dynamic Languages	98
5.1	Introduction	98
5.2	The Polymorphism Overhead	101
5.3	The Type-based Memory Layout	104
5.3.1	Memory Layout Analysis	106
5.3.2	Internal Mapping after Restructure	107
5.3.3	Dictionaries	109
5.3.4	Evaluation	110
5.4	Language-independent Type-based Stores	116
5.4.1	Overview of Language-independent Storage Strategies	116
5.4.2	Extending Rstrategies with Type-based Stores	117
5.4.3	Evaluation	119
5.5	Related Work	122
6	Context Aware Presizing for Dynamic Languages	124
6.1	Introduction	124
6.2	Optimal Presize Prediction Challenges	126

6.2.1	Branches	127
6.2.2	Allocation Calling Context	127
6.3	Context Aware Presizing	127
6.4	Approach Overview	130
6.4.1	Object Size Profiling	132
6.4.2	Inferring the Calling Context	132
6.4.3	Accuracy of Context Identifiers	134
6.5	Optimal Size Prediction	137
6.5.1	Installing the Context in the Object Header	137
6.5.2	Reading and Writing to the Context Map	139
6.5.3	Optimal Size Estimation	139
6.6	Implementation	140
6.7	Evaluation	141
6.7.1	Methodology	141
6.7.2	Workload Description	142
6.7.3	Observed Allocation Sites	142
6.7.3.1	Discussion of Results	143
6.7.3.2	The Overhead of using the Profiles	146
6.8	The Call Stack and Garbage Collection	148
6.8.1	Phase Analysis	148
6.8.2	Live Size and Allocation Rate	149
6.9	Related Work	151

7 Towards Correct Memory Management for Python Native Extensions **152**

7.1	Introduction	153
7.2	The Python C API	157
7.2.1	Fixed-Address Object Model	159

7.2.2	Non-Opaque PyObject Structs	159
7.2.3	Exposing GC Implementation Details	160
7.2.4	Borrowed References	160
7.3	CyStek: A Stack-based C API for Python	161
7.3.1	Design	161
7.3.2	Implementation	163
7.4	Garbage Collection	166
7.4.1	Memory Reclamation	167
7.4.2	Overflowing the Array	168
7.4.3	Object Lifetime	169
7.5	Evaluation	171
7.5.1	Methodology	171
7.5.2	Discussion of Results	175
7.6	Migration of Extensions	178
7.6.1	Methodology	179
7.6.2	Case Study	185
7.7	Related Work	189
8	Conclusions	191
8.1	Summary	191
8.1.1	OMR-based Garbage Collection	191
8.1.2	Dynamic Trace Sizing	193
8.1.3	Type Optimization for Heterogeneous Collections	194
8.1.4	Call-stack-based Presizing and Garbage Collection	195
8.1.5	Memory Management for Native Extensions	196
8.2	Future Work	197
8.2.1	Optimal Heap Limits	198
8.2.2	Phase-based Garbage Collection	199

8.2.3	Memory Safety of Foreign Function Interfaces	199
Bibliography		219
A Type-based Stores for Collections		220
A.1	Type-based Stores for PyPy	220
A.1.1	Implementation	220
A.2	PyPy Evaluation	223
A.2.1	Execution Speed	223
B Context Aware Profiling		224
B.1	Motivation	224
B.1.1	Collection Over-allocation Schematic	224
B.1.2	Collection Expansion and Shrinking overhead Schematic	225
B.2	Accuracy of Context Identifiers	226
B.2.1	Ambiguity Example for a Python Benchmark	226
B.2.2	Active Record Resizing Case for a Python Benchmark	226
B.2.3	Call Site Wrapping Case for a Python Benchmark	226
Vita		

List of Tables

3.1	Raw Measurements	54
3.2	Halstead Measurements	55
3.3	MI and LCOM	56
3.4	OMR-based GCs Vs. RPython GCs in PyPy – the geometric mean of software engineering metrics across the seven garbage collection algorithms	63
4.1	Hyper-parameter Tuning with Grid Search Vs. Looping	75
4.2	Nursery Sizing and FPS	79
4.3	PyPy and Racket Benchmarks	96
5.1	A Description of PyPy Benchmarks	102
5.2	Collection Strategy Transitions – an approximate breakdown of collection transitions to the object strategy during program execution while running the PyPy benchmark suite for benchmarks described in Table 5.1 . For some key container type combinations, as much as 40%–50% of collections become heterogeneous from a specific type like integer, string or empty	102
5.3	Hardware Setup	111
5.4	Language-independent Benefits of Type-based Stores – lower values are better, all results are shown as ratios normalized to the baselines. The language-independent type-based stores improve performance for both Ruby and Racket applications	120

6.1	Call Site Analysis for Python – statistics generated from the profiling tool	134
6.2	Performance Gains and Memory Savings for Context Aware Presizing – lower values are better, all results are shown as ratios normalized to the baselines and aggregated based on the geometric mean across 30 invocations. The error values correspond to the lower bounds of the geometric standard deviation. There is a 12% performance improvement and memory savings of 16% by mean for the best median strategy.	145
6.3	The Overhead of using the Profiles – lower values are better, values presented as ratios relative to the PyPy baseline within the given error margins. The columns (WR) and (RD) refer to write and read respectively. (WRD) refers to both read and write. We also present the total overhead for all operations, all values are normalized to the baseline, aggregated across 30 invocations	147
7.1	Description of Python Native Extensions Ported to CyStck – their size of source lines of code (LOC), total number of PyPI downloads. An associated benchmark used for the evaluation is also indicated . .	173
7.2	Unnormalized Results — for the native system Python and total time relative to execution time	174
7.3	Patterns and Transformations for The Hypothetical Example in Section 7.6.1	184
7.4	Migration Metrics for UltraJson — the automatic port covers approximately 60% of the manual port changes	186
7.5	Migration Metrics for PicoNumPy — the automatic port covers approximately 33% of the manual port changes	187

7.6	Migration Metrics for NumPy — the automatic port covers approximately 75% of the manual port changes	187
7.7	Migration Metrics for Matplotlib — the automatic port covers approximately 90% of the manual port changes	187
7.8	Migration Metrics for KiwiSolver — the automatic port covers approximately 90% of the manual port changes	188
A.1	Benchmark Speed results in seconds, lower is better	223

List of Figures

2.1	An Object Graph Before Collection	17
2.2	The Object Graph After Collection	19
2.3	Eclipse OMR GC Architecture	23
2.4	MMTk Design	27
2.5	The RPython Translation Toolchain	30
2.6	Storage Strategies [24]	38
2.7	Element Kinds in V8 [34]	39
3.1	Design of the OMR GC in the RPython Framework	51
3.2	Mutator Time	58
3.3	Total Time	59
3.4	GC Time	59
3.5	GC Overhead for RPython-based Virtual Machines – this is normalized to the baseline PyPy and PyHP implementations. In relative terms the overheads here range from 0.2% to about 4%	61
3.6	OMR-based GCs Vs. RPython GCs in PyPy – the total execution time (GC time + Mutator time) as a geometric mean across seven garbage collection algorithms	62
3.7	OMR-based GCs Vs. RPython GCs in PyPy – the allocation, garbage collection and mutator time for the semispace policy	63
4.1	Architecture of the Server Benchmarks	71

4.2	The Mean Time Taken to Process 445 Requests on the Django and Webpy Server Processes	73
4.3	The Mean Memory Used for Both Django and Web.py	74
4.4	Time Per Frame With Major GC	77
4.5	Time Per frame Without Major GC	77
4.6	The Relationship Between the Cache and Execution Time at Varying Nursery Sizes Across all Workloads	78
4.7	Phases of a Tracing JIT	81
4.8	AI benchmark — The cost of changing the trace limit on resident set size	82
4.9	IObasic benchmark — The cost of changing the trace limit on execution, minor GC and major GC time in seconds	83
4.10	The Effective Trace Limit is Application Specific — blue bars show the best trace size for each of the benchmarks which is the effective trace limit while the red dotted line is the default RPython trace limit	85
4.11	The GC-aware Dynamic Trace Sizing technique	88
4.12	PyPy Performance — The execution time is measured in seconds. Dynamic trace sizing improves performance for Python applications to as high as 12% across the subset of PyPerformance benchmarks we evaluated	95
4.13	Pycket Performance — The execution time is measured in seconds. Dynamic trace sizing improves performance for Racket applications to as high as 5% across the Racket benchmarks we evaluated	95

5.1	The Overhead of Storage Strategies – the unoptimized heterogeneous collections and the impact of strategy transitions slows down applications. This leads to an overhead of 4% by mean for the four benchmarks we ran from the PyPy benchmark suite described in Table 5.1. Transitioning a list of one million integers to the object strategy, takes the program 25 seconds with storage strategies while it takes less than five seconds without storage strategies	103
5.2	Type-based Stores vs. Storage Strategies – consider a list <code>lst = [1, "foo", 2, "bar", 3, "zar"]</code> , (a) shows the memory layout of the list after applying storage strategies; and (b) shows the memory layout of the list after applying type-based stores and storage strategies	105
5.3	Internal Mapping to Support Operations – the map tracks positions of items in the source data structure to help with processing access operations. This map is useful for sequential collections	108
5.4	Type-based Stores for Dictionaries – we only optimize keys to reduce the cost of handling every possible key/value type combination when unwrapping	109
5.5	Speedup for PyPy – the results are normalized to the storage strategies baseline, lower is better, the type-based stores technique accelerates execution times for most of the benchmarks	112
5.6	Memory Savings for PyPy – the results are normalized to the storage strategies baseline, lower is better, the type-based stores technique reduces the memory footprint for some benchmarks and increases the memory footprint for some others	112
5.7	The Overhead of Collection Operations – the results are normalized to the storage strategies baseline, lower is better	115

6.1	Call Graph for Listing 6.3 – collections are indicated at the respective call sites. Arrows indicate the calling context which is a chain of calls with origins from the main method	129
6.2	Context Aware Collection Size Profiling	131
6.3	Proving the Stack Height Hypothesis – the profiler generates four logs in this experiment, one without disambiguation, one after applying only active record resizing, another after applying only function cloning and call site wrapping and the last log file is the one after applying all the three techniques. The configurations correspond to the disambiguation methods; <i>active record resizing (ARR)</i> : the intermediate operand of the instruction is modified, changing the function’s active record size; <i>call site wrapping (CSW)</i> : replacing a call on one of the edges with a wrapper function that then calls the original function; and <i>function cloning (FC)</i> : replacing a call to a duplicate function with a call to a copy of the function. We observe that the context identifier can uniquely identify a call path with a 79.9% accuracy level by mean.	136
6.4	Installing the Context in the RPython Object Header – when a collection object is allocated, memory is allocated and reserved for the context identifier. The set and unset is for the hash field	138
6.5	Number of Observed List Sizes per Line Number of Allocation Sites – the Y (collection count) and X (line number) axes show the size of collections and allocation sites respectively, while the orange and green lines show the mean and median	144
6.6	Phase Monitoring for Python applications — based on the stack height. A point on the X-axis maps to a call-site in the program . . .	149
6.7	Stack Height vs. Live Memory Size and Allocation Rate	150

7.1	The Design of the CyStck Prototype — we show the two spaces that correspond to the native and the VM environments, including the stack and array data structures used to communicate between C and Python, described in Section 7.3.1	162
7.2	Unnormalized Rate of Bytes — copied generally in Python code, native code but also across the Python/C boundary	175
7.3	The System Flow for the Migration Tool	180

List of Symbols, Nomenclature or Abbreviations

JVM	Java Virtual Machine
GC	Garbage Collection
JIT	Just-In-Time Compiler
AST	Abstract Syntax Tree
API	Application Programming Interface
AMM	Automatic Memory Management
MREs	Managed Runtime Environments
GaS	GC-as-a-Service
BDWGC	Boehm-Demers-Weiser Conservative Garbage Collector
MMTk	Memory Management Toolkit
GIL	Global Interpreter Lock
VMs	Virtual Machines
LOC	Lines of Code
LLOC	Logical Lines of Code
SLOC	Source Lines of Code
CC	Cyclomatic Complexity
LCOM	Lack of Cohesion of Methods
MI	Maintainability Index
DLL	Dynamic-link Library

Chapter 1

Introduction

The topic of garbage collection (GC) is about seventy years old (first GC in 1958) but remains a popular field of research for both industry and academia because GC algorithms have diverse effects on applications, architectures and programming languages [14, 39, 81, 128]. This dissertation focuses on the effects of garbage collection and associated memory management optimizations for dynamically typed programming languages.

1.1 Dynamic Languages

Dynamic languages like Python and JavaScript continue to be widely used and for a wider number of domains. Their main characteristic is that they use dynamic type systems to allow rapid prototyping, but also provide desired features like: easier syntax, automatic memory management, a robust ecosystem of standard libraries and third party packages, support for dynamic code generation and execution, interactive execution environments, and advanced introspection capabilities [77].

These productivity-oriented features come at a cost compared to static languages. This is because dynamic languages trade performance for dynamism due to the extra runtime tasks required for aspects like type checking and method binding among

others. We observe and speculate that the implementation of dynamic languages in response to the recurring performance challenges will likely adopt the following course.

The first trend is that as user workloads become numerous from all domains, it will be unsound for a language, dynamic or not, to only support one GC algorithm. Evolving to another policy will not be sufficient; users will need a way to configure a policy that works best for their workloads. Garbage collection frameworks like Eclipse OMR will be the most feasible way to effectively implement new correct, safe, modular, maintainable and high performing garbage collection policies.

Dynamic languages will also remain popular for the foreseeable future and as more domains are explored, it will become mandatory to have a just-in-time (JIT) compiler. Implementations that do not have JITs will have to support them to improve the performance for their users as a way of offsetting some of the overhead from dynamism. There is no silver bullet to the engineering complexity of JITs; most JIT work will be custom for existing languages, especially those with large communities and adoption. However, newer languages can use repurposed JITs from frameworks like the meta-tracing RPython framework for less complexity. JIT compilation needs more novel software-only optimizations to avoid the diminishing performance gains acknowledged in the literature [77].

Optional static typing has shown that users have use cases for static typing in dynamic languages and due to the lack of runtime support, static typing will continue to be used mostly for linting and debugging. However, runtime effects of type checking in dynamic languages due to polymorphism will continue to require evolution through optimizations.

Most dynamic languages also have C APIs that have not seen improvements in the last decades. The rise of alternate implementations and demand for memory safe interoperability between virtual machines and native environments like C/C++ will

require novel solutions to address challenges of encapsulation and memory safety. Finally, as languages evolve, dynamic or not, their communities will grow weary of rewriting their projects and libraries due to drastic feature changes between versions. Pushing for compatibility with no new features is not realistic; instead, tools to help migrate users to newer versions will be inevitable.

1.2 Thesis

To extend existing research towards the trends discussed in the previous section, this dissertation focuses on a study *to optimize and understand automatic memory management for dynamic programming languages* and posits the thesis that:

The design of correct, memory safe and high-performing garbage collection should build on language-independent frameworks, account for the disparate memory and structural models in the case of native interoperability and minimize the garbage collection overhead in the interaction with other components of a runtime.

We propose several techniques to defend this thesis while answering the following research questions:

- **RQ1:** Can the use of the Eclipse OMR garbage collection framework lead to the correct, portable, and language-independent implementation of high performing garbage collectors for the RPython Meta-tracing-JIT-based dynamic languages?
- **RQ2:** Can we attach garbage collection overhead to any specific implementation aspects of meta-tracing-JIT-based dynamic languages?
- **RQ3:** Can heterogeneous collections in dynamic languages be stored in an unboxed form in memory without significant overhead?

- **RQ4:** Can we achieve context aware prediction of the final size of a collection for dynamic languages with minimal overhead?
- **RQ5:** Can combining a stack and light-weight handles, along with object introspection, lead to correct memory management for Python native extensions?

1.3 Contributions

We address three main research gaps related to garbage collection; namely, modularity, performance and interoperability with native code.

Modularity is a long aspired principle of software engineering and code reuse for garbage collection subsystems is now mature but remains lethargic to adoption by production runtimes due to the integration cost. Towards modularity, we present one of the first studies in the context of dynamic languages on the use of frameworks for garbage collection.

Programming languages also aspire to provide acceptable performance. First, we extend the use of profiling and prediction to reduce the memory overhead in tracing JIT compilers and second, we make two contributions to improving the performance of collections.

Overhead can also be related to communication between different programming languages. In most cases this communication presents both performance and safety difficulties. We present experimental work that provides immediate safety benefits and a path to better performance in the future.

To appreciate the thesis statement, we introduce and summarize our five main contributions to the research and practice of memory management next, explicating the claims.

1.3.1 Garbage Collection with Frameworks

Dynamic languages like Python have supported automatic memory management or garbage collection for a long time; however, there still remains areas where they have not been studied as much as static languages like Java. The focus of garbage collection research for mainly Java and the JVM [81, 15] has led to better performing and sophisticated garbage collectors for the Java ecosystem and not as much for dynamic languages. One of the areas that has been studied extensively for Java and the JVM is the idea of implementing garbage collection through a framework. This analysis alone contributed to developments and insights into the performance impact of GCs for Java applications [15], new GC algorithms informed by the behaviour of Java programs especially [123, 16] and even looking at the safety aspects of GC implementation [93].

The first outcome of this research is a detailed study to understand if garbage collection through GC frameworks has any benefits to offer dynamic languages, like Python, and other languages that are based on the RPython framework, shipped with PyPy. The RPython framework allows development of interpreters using a restricted dialect of Python called *RPython*. Interpreters developed with this framework get a free Just-in-Time (JIT) compiler and garbage collector [117]. We study the use of the Eclipse OMR framework in Chapter 3. We significantly note that GC frameworks come with overhead in calling the library, though fast path optimizations for allocations and write barriers can alleviate this to a degree.

1.3.2 Understanding the Garbage Collection Cost

The literature has many studies on the overhead related to several aspects for both garbage collection and dynamic languages [7, 15, 69, 26, 122]. We take a different turn on GC analysis in this research, focusing on the garbage collection cost for meta-tracing JIT-based dynamic languages for real-world workloads, well known in

the Python community.

We revisit previously studied topics like the effect on heap sizing, nursery sizing, and the cache but on realistic applications of web servers, machine learning and gaming. We further do analysis for an area not currently well investigated, the garbage collection cost related to JIT tracing, identifying a trade-off between tracing and garbage collection, where without any optimization, the garbage collection overhead can cancel out the benefits of tracing JITs due to long traces.

Tracing JIT implementations dictate a static trace size, in part to help with this overhead but as we show in Chapter 4, the effective trace size is application specific, and setting it to the maximum is not the solution due to reasons like the negative effect of guards inserted to specialize the trace based on the types observed during tracing in tracing JITs [118] in some cases. To this end, we propose and discuss a novel *dynamic trace sizing (DTS)* technique that uses profiling, to dynamically set an optimal trace size, at a point where the application benefits from the highest level of JIT optimizations and the lowest level of garbage collection overhead.

1.3.3 Type-based Stores for Collections in Dynamic Languages

Dynamic languages also introduce garbage collection pressure due to their object representation which impacts collections as an example. Heterogeneous collections are common in both statically and dynamically typed languages, where a single container can contain values of different types. Boxing is a technique that is used to represent the objects with a general and uniform type to support this flexibility. However, boxing inhibits compiler optimizations and introduces both performance and memory overhead to these languages. Optimizations like storage strategies [24], discussed in the literature to address this overhead, only work for homogeneous collections.

We present *type-based stores* as our third contribution in Chapter 5, which is a novel memory layout that can exploit storage strategies as an optimization for heterogeneous

collections. With type-based stores, the memory layout of the collection data structure is restructured storing objects by type in contiguous storage areas, we call *stores*. These stores can each be managed by a storage strategy.

We implement type-based stores first as a collection optimization in PyPy. We also implement language-independent type-based stores in RPython and perform evaluation for Topaz, a Ruby implementation and Pycket, a Racket implementation. Our evaluation shows that type-based stores can improve the performance and memory consumption for Python applications while the language-independent type-based stores also improve performance for Ruby and Racket applications.

1.3.4 Calling Context Aware Presizing

Garbage collection pressure in both static and dynamic languages is also attributed to size allocation for collections [34, 9]. Due to their flexibility, most languages support re-sizable collections, but internally, the collections are stored in a fixed-sized data structure. Collection expansion requires creation of a larger internal structure and an extra copying cost from the original structure to the larger structure.

Presizing is a technique used to allocate collections with an optimized size to avoid the overhead involved with shrinking and expanding collections when their sizes change but can also avoid over-allocation of internal slots. Existing work uses allocation-site-based methodologies, some using the GC to achieve presizing [34], and in other cases canonical profiling [68]. These solutions all have limitations in capturing the allocation calling context and branches in the control flow of the program.

Towards our fourth contribution, in Chapter 6, we argue that by combining the context-identifier through call-site analysis with profile information dynamically or from a prior run on allocation sites, we can predict and optimally size collections to reduce the overhead of resizing.

1.3.5 Memory Management for Native Extensions

Most dynamic languages are also extensible and can be embedded in other languages; a common case is to provide a C API to interface to the language virtual machine. The Python language is one such language where most of the high-performance libraries for Python are written in C so this interface is important [10]. The C API allows developers to write better performing programs in C that can be invoked from a dynamic language that has a C API. Garbage collection should also be handled for code that calls the C API.

It is common for C APIs like Python's and Ruby's to expose internal details of the VM and tightly couple garbage collection details in the API. CPython for example requires that C programmers call reference counting operations as a way of supporting garbage collection for native extensions, a process that is error-prone. This has led to a C API that is not flexible enough to evolve towards tracing garbage collection without a rewrite or modifications that break backward compatibility of existing native extensions. Over the years, the Python C API has generally grown to be a challenge for the evolution of the Python ecosystem.

As our final and fifth contribution in Chapter 7, we implement a new Python FFI, we call *CyStck*, by combining a stack and light-weight handles, to support efficient garbage collection (GC) in Python native extensions. Further, we demonstrate complementary garbage collection automation for FFIs with a memory introspection tool called *Liballocs*. Five large, real-world Python extensions are ported to CyStck, thoroughly profiled with the Scalene profiler, comparing CyStck to the current CPython C API and another Python C API implementation, HPy. We also implemented a tool to automate the migration of extensions from the CPython C API to CyStck using pattern matching and static analysis, with a success rate as high as 90%.

1.4 Structure

This document continues with a discussion of background work in Chapter 2. The basics of garbage collection and existing GC frameworks are described as well as aspects of memory layout, representation and management of native extensions.

Chapter 3 presents the design, implementation and experimental evaluation of the Eclipse OMR based garbage collection for RPython-based dynamic languages. The evaluation covers both performance and software engineering.

In Chapter 4, we extend the work in Chapter 3 with a detailed analysis of the impact of garbage collection for selected Python real-world applications, proposing an optimization that mitigates the garbage collection overhead related to JIT tracing. Two collection optimizations for dynamic languages are proposed in Chapters 5 and 6. The first one allows the storage of items in heterogeneous collections without boxing, while the other allows for collections in dynamic languages to be allocated with their optimal size to avoid the resizing overhead.

Chapter 7 discusses the memory management challenges that surround the design of C APIs and proposes a stack-based Python C API.

Finally, Chapter 8 has the dissertation summary and discussion on directions for future work.

1.5 Presentation of Evaluation Results

We provide error data for most of the values presented in the evaluation sections of each chapter. Each error value represents the variability in results spread across multiple runs of the workloads, to serve as an indication of the uncertainty in the reported measurements.

For unnormalized results, error values always correspond to the standard error of the geometric mean unless stated otherwise. For normalized data, error values are

generated by computing the product of the the geometric standard deviations (GSDs) of the baseline (denominator) and numerator measurements. This is a conservative error measure which can appear large when both sets of measurements have significant variation. Error bars are constructed as $[r/f, r * f]$, where r is the normalized ratio and f is the product of the two GSDs.

Showing error values on graphs and tables is beneficial for independently repeated experiments, but not very relevant for single runs of experiments. We therefore do not show error bars and data for experiments that involve a single run, especially where we are interested in pattern analysis rather than performance comparison metrics.

1.6 Publications

The work in this dissertation appears in the following peer-reviewed publications, where the dissertation author is the primary author:

1. Joanna Nanjey, David Bremner, and Aleksandar Micic. 2021. Eclipse OMR garbage collection for tracing JIT-based virtual machines. In Proceedings of the 31st Annual International Conference on Computer Science and Software Engineering (CASCON '21). IBM Corp., USA, 244–249.

The PyPy on OMR project was initiated by the advisor, Dr. David Bremner and execution was done as a collaboration with IBM Canada, with IBM project lead Aleksandar Micic advising on correctly calling the OMR API.

2. Joanna Nanjey, David Bremner, and Aleksandar Micic. 2022. The Garbage Collection Cost For Meta-Tracing JIT-based Dynamic Languages. In Proceedings of the 32nd Annual International Conference on Computer Science and Software Engineering (CASCON '22). IBM Corp., USA, 140–149.

Following the OMR work above, the dissertation author designed the GC experiments in this paper and proposed the dynamic trace sizing technique. The work was supervised by both Dr. Bremner and Aleksandar Micic.

3. Joannah Nanjekye, David Bremner, and Aleksandar Micic. 2023. Towards Reliable Memory Management for Python Native Extensions. In Proceedings of the 18th ACM International Workshop on Implementation, Compilation, Optimization of OO Languages, Programs and Systems (ICOOOLPS 2023). Association for Computing Machinery, New York, NY, USA, 15–26. <https://doi.org/10.1145/3605158.3605849>

The dissertation author proposed the *CyStck* Python C API implementation and the migration experiments in the paper. The work was supervised by both Dr. Bremner and Aleksandar Micic.

The following artifacts were published with the above peer-reviewed papers:

1. Joannah Nanjekye, David Bremner, and Aleksandar Micic. 2023. Reproduction Package for Article: Towards Reliable Memory Management for Python Native Extensions. <https://doi.org/10.1145/3554356>

The author of this dissertation is also the primary inventor of the following filed patents and defensive publications, that contain techniques proposed as part of this research:

1. Joannah Nanjekye, David Bremner, and Aleksandar Micic. International Business Machines Corporation. File Date: 29-Sept-2023. Optimal JIT Trace Sizing for Virtual Machines. Docket No. P202204294US01.

Aleksandar Micic sponsored the disclosure, while IBM Canada hired the lawyers who prepared and filed the patent. The work was supervised by both Dr. Bremner and Aleksandar Micic.

2. Joannah Nanjekye, David Bremner, and Aleksandar Micic. International Business Machines Corporation. Invention Title: Type-based Stores for Data Structure Polymorphism. Invention Publication Link: <https://priorart.ip.com/IPCOM/00274036D>. Invention Reference: P202305693 (Record ID 99220386). Publication Date: 22-March-2024. Invention Publication Number: IPCOM000274036D.

Aleksandar Micic sponsored the disclosure, while IBM Canada assigned the technical writers who prepared the final defensive publication. The work was supervised by both Dr. Bremner and Aleksandar Micic.

3. Joannah Nanjekye, David Bremner, and Aleksandar Micic. International Business Machines Corporation. Invention Ref: P202305693. Context Aware Presizing for Dynamic Languages. Record ID: 99220386. Status: IBM Editorial processes.

Aleksandar Micic sponsored the disclosure, while IBM Canada assigned the technical writers who prepared the final defensive publication. The work was supervised by both Dr. Bremner and Aleksandar Micic.

The following submitted papers also have research that is discussed in the contributions of this work and primarily authored by the author of this dissertation:

1. Joannah Nanjekye, David Bremner, and Aleksandar Micic. 2024. Data Structure Polymorphism and Presizing Optimizing Techniques for Dynamic Languages. The Journal of Object Technology (JOT 2024). Submitted on January 22, 2024, Revisions Requested on April 26 2024.

The *type-based stores* technique in this paper was proposed by the dissertation author, the name of the technique was suggested to

the author by Dr. Bremner. The work was supervised by both Dr. Bremner and Aleksandar Micic.

Chapter 2

Memory Management and Dynamic Languages

Memory management can be manual or automated; the latter as supported by most mainstream programming languages, is responsible for abstracting the allocation and deallocation of memory from developers. This eliminates challenges such as remembering to free the memory allocated to an object. It also avoids the severe danger of using memory that is already released. Garbage collection (GC), the main theme of this dissertation, is a well studied form of automatic memory management. To place the research contributions in context, this chapter provides the relevant background on memory management. Section 2.1 starts with an overview of garbage collection (GC) and its well-known algorithms that form the basis of the dissertation. Section 2.2 describes the common garbage collection frameworks, while Section 2.3 details garbage collection in Python. The garbage collection frameworks are the basis for the work in Chapters 3 and 4. Section 2.4 discusses the memory representation of types in dynamic languages, which is the basis for the work on the efficient layout of element types for collection data structures in Chapter 5, while Section 2.5 discusses presizing schemes which form the background for Chapter 6. Section 2.6 discusses how

memory is managed for native extensions for languages like Python, Ruby, JavaScript, Lua, and so forth, which is the basis for the work in Chapter 7.

2.1 Garbage Collection

The first garbage collector was written about 60 years ago and shipped in the first versions of LISP [99]. Many programming languages like Python, Java, etc., have now adopted garbage collection. In the rest of this document, and as is the norm in the literature [81], we also use the terminology coined by Dijkstra et al. [47] to discuss the different aspects of garbage collection. The *collector* refers to the aspect of the application that reclaims memory for objects that are considered garbage, while the *mutator* refers to the application program, and the *allocator* is responsible for the allotment of memory for objects.

2.1.1 Categorizing Garbage Collectors

As identified in several publications, garbage collection algorithms are categorized by how they allocate objects, identify unused objects and free unused memory [121, 53, 55, 139].

Objects are allocated in memory by either *contiguous* or *free-list allocation*. Contiguous allocation places objects in memory in the order in which they are allocated. It achieves this by incrementing the allocation pointer based on the size of the object to be allocated. Algorithms based on this technique have good locality because objects are allocated and used together. Free lists are lists of variable-size cells of memory. Free-list allocation places objects in these cells. Objects are allocated in memory based on their size relative to the cell, in a first-fit fashion rather than allocation order. Free-list allocation permits non-contiguous allocation, which is prone to fragmentation and poor locality. An improved alternative to free-list allocation, is the *segregated-fits*

allocation mechanism, which divides a free list into several subsets, depending on the size of the free blocks, a freed block is placed on the appropriate list, and an allocation request is serviced from the appropriate list. Segregated-fits allocation is less prone to fragmentation and provides faster allocation [81].

Garbage collectors are commonly classified as *reference counting* or *tracing*, based on how they identify garbage. Garbage identification by reference counting involves tracking the number of times an object is referenced by other objects. For every new reference to an object, the reference count field of the referenced object is incremented. An object's reference count is decremented when it loses a reference. An object is considered garbage if its reference count drops to zero, at which point it is deallocated freeing up space for a new object [81]. Reference counting has been critiqued for its poor performance due to the overhead of maintaining reference counts [16, 114]. It also cannot reclaim memory for objects involved in a cycle. Tracing garbage collectors are the most popular garbage collectors; in fact, *garbage collection* often refers exclusively to tracing. Tracing garbage collection identifies objects as garbage by scanning the object graph for objects that are not directly or indirectly referenced from a root object. Any object that is not reachable by reference from the root objects is considered garbage and thereby collected. The rest of the objects are considered in-use and therefore kept in memory [81, 30].

Memory reclamation is achieved by one of these strategies: *back to a free-list*, *sweeping*, *compaction* and *copying*. For objects tracked by reference counting, memory is returned to the free-list at the time of deallocation. Sweeping, compaction and copying are not commonly used with reference counting. Sweeping involves traversing the object heap, marking unused blocks of memory as free, and is performed in conjunction with tracing that marks reachable objects as live. Compaction re-arranges the remaining objects after a collection to avoid fragmentation [81]. In most cases the live objects are moved into a contiguous region of free memory [121, 53, 55, 139].

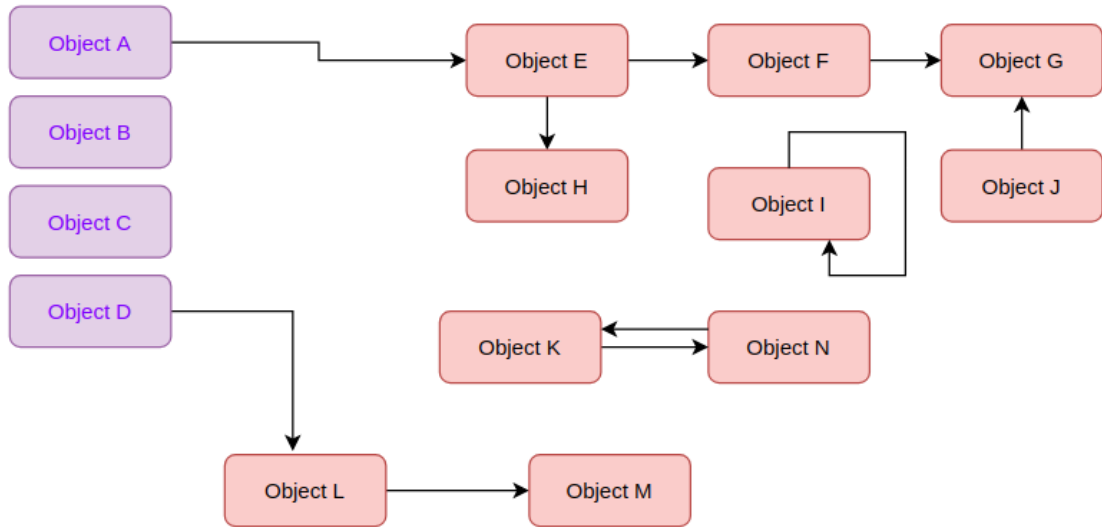


Figure 2.1: An Object Graph Before Collection

Copying moves objects from one region in memory to another, freeing up the former. This is the common case for generational garbage collection that we discuss later.

2.1.2 Overview of Definitions

The following terminology is used to describe the different GC concepts in this section of the document.

Object: This is simply an instance of data stored on the heap.

Object graph: This is the layout of objects in memory; the objects make up the nodes of this graph as shown in Figure 2.1. An object can reference itself as is the case for *object I*. Objects can also reference each other like objects *K* and *N*. Objects *I*, *K*, and *N* are involved in cycles.

The Root Set: This is a set of objects in the object graph from which references originate and are directly accessible by the mutator. In most cases, these are references

held in registers, stack-allocated local variables, and global variables. Objects *A*, *B*, *C*, and *D* make up the root set in the object graph depicted in Figure 2.1.

Reachable or Live Objects: These are objects that have an incoming edge referencing them from one of the root sets or edges from other reachable objects. Some literature distinguishes between reachable objects, and the subset of those that will be used again, live objects. The live objects are an ideal, typically impossible to compute, and the reachable objects are one approximation. In Figure 2.1, objects *E*, *F*, *G*, *H*, *L* and *M* are reachable and are not freed when the garbage collector runs.

Unreachable or Dead Objects: The opposite of live objects, i.e., objects that do not have any incoming edge referencing them from the root set or edges from other reachable objects. Objects *J*, *I*, *K* and *N* in Figure 2.1 are unreachable objects. These are freed when the GC runs.

Collection: The process of reclaiming memory that is occupied with unreachable objects. After collection, the new object graph is as shown in Figure 2.2.

Barrier: An operation that is invoked before reading or writing to a pointer. Garbage collectors may use read or write barriers, or both.

A Conservative Collector: A garbage collector that works with minimal information about pointers.

A Meta-circular garbage collector: A garbage collector that is written in the same language as the language it performs collection on.

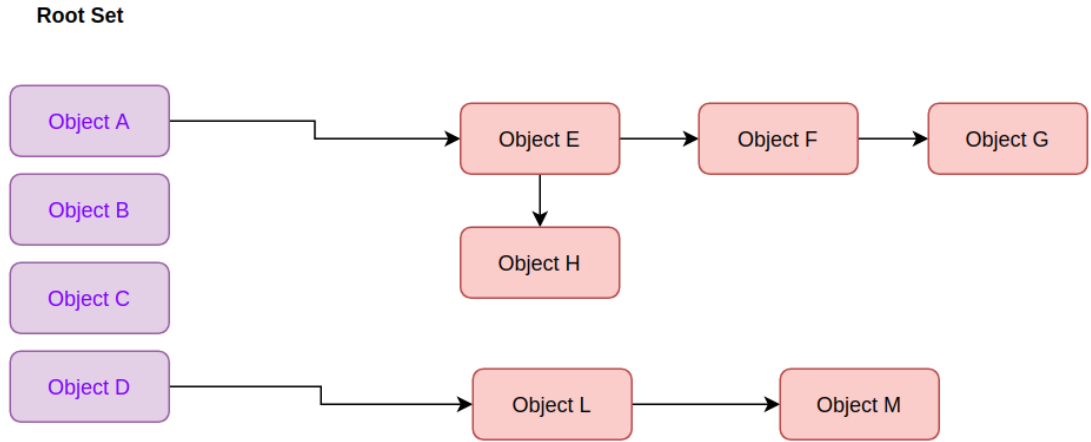


Figure 2.2: The Object Graph After Collection

2.1.3 Garbage Collection Strategies

There are four recognized garbage collection algorithms on which most of the existing GC work is based [81, 139]. These include *reference counting*, *mark-and-sweep*, *copying* and *mark-and-compact*. We discuss each of the algorithms next, including the generational technique, which partitions the heap into regions.

Reference Counting

As described earlier, reference counting works by tracking a count of all incoming references to an object, collecting objects whose reference count decreases to zero. A write barrier is used to synchronize changes to every pointer, decrementing the object reference count when the pointer is overwritten and incrementing it on pointer creation. Reference counting garbage collectors have been used since the 1960s due to their low memory overhead and ease of implementation [122, 114] even in mainstream programming languages like PHP, Perl, and Python. Reference counting has two main limitations, it cannot collect garbage for objects in a cycle and it incurs performance overhead as a result of tracking pointer mutations. Significant research has attempted to address some of the known challenges of reference counting [16, 123, 136]; tracing collectors (in practice mainly generational collectors, described below) have been

adopted for better throughput and handling of cyclic garbage [91, 114, 123, 136].

Mark-and-Sweep Garbage Collection

A basic mark-and-sweep garbage collector performs memory management activities in two phases, i.e., the mark and sweep phases. Before the mark phase, the GC collects and identifies a list of all references held in registers, global variables, stack-allocated local variables and function arguments, etc., which comprise the root set.

Tracking of roots is external to the GC and, instead, handled by the runtime. The *mark phase* involves traversal of the object graph from the root set, marking all objects or nodes that have an incoming reference from the root set or other reachable objects. The pseudocode in Listing 2.1 depicts the marking phase.

```
1 def mark(*roots)
2     for each rt in roots:
3         mark(rt)
4         for each obj referenced by rt:
5             mark(obj)
```

Listing 2.1: The GC Marking Algorithm

At the end of the mark phase, every reachable object should have been visited and its mark bit set accordingly. The sweep phase then traverses the whole heap, freeing any memory with unmarked objects. The mark bit for all objects is reset in preparation for the next GC cycle. Listing 2.2 shows the sweep algorithm.

```
1 for each object obj in heap:
2     if marked = true then
3         clear(obj)
4     else
5         free(obj)
```

Listing 2.2: The GC Sweeping Algorithm

Copying

Copying collectors divide the heap into two regions. Allocation of objects is done in the first region called the *from-space* and when it runs out of space, collection takes place copying any live objects to the second region called the *to-space*. The pointers to the moved objects are updated. Memory in the from-space is freed since it now contains only dead or unreachable objects.

Compaction

Compacting garbage collectors rearrange objects in memory after a collection cycle. Compaction is commonly used in the mark-and-sweep collector. In the mark-and-sweep GC, after the marking phase, compaction can be used in addition to a normal sweep. As discussed earlier, sweeping without rearranging objects creates fragmentation and compaction solves this problem. A simple compaction algorithm uses *sliding* to compress reachable objects into a contiguous memory space while maintaining their order in the heap [81, 53].

Generational Garbage Collection

The weak generational hypothesis states that most of the objects live for a short time [81]. Based on this hypothesis, generational collectors are region-based GCs and similar to the semi-space collectors. In a basic case, generational collectors partition the heap in two generations, the *nursery* (also known as *the young generation*) and the *old* generation. The nursery contains the newly allocated objects while the old generation contains the rest of the objects. Based on the hypothesis, generational collectors focus the collection effort on the newly allocated objects in the young generation¹. This region is frequently collected while the old generation is less often collected. When the weak generational hypothesis holds, generational collectors are efficient as a small fraction of objects have to be copied to the old generation [81].

¹The nursery has to be large enough in order for the frequency of minor collections to be small

2.2 Garbage Collection Frameworks

Different components and areas of runtime environments for managed high-level programming languages are complex to implement. This makes these components and areas hard to design, extend, modify, reuse, and understand. One of the areas affected by this complexity is automatic memory management.

To address this complexity, GC frameworks and libraries have been built over the years. The libraries have a language-independent interface that any runtime can use to implement any supported GC algorithms and policies. They enhance the separation of concerns in the design and implementation of garbage collection for Managed Runtime Environments (MREs) by allowing us to build new GCs from reusable components. The most commonly used GC frameworks are described in this section.

2.2.1 Eclipse OMR

The Eclipse OMR project is a set of open source C/C++ components that can be used to build robust language runtimes that support several hardware and operating system platforms [49]. One of the components in Eclipse OMR is the GC framework, which is a component for implementing automatic memory management. In the rest of the document, component and framework are used interchangeably in the context of GC frameworks. The framework is used to implement GCs for runtimes with managed heaps.

Architecture

The OMR GC exposes a C interface that any runtime can call to implement garbage collection as shown in Figure 2.3. The C callable API is backed by a C++ implementation, which is commonly referred to as the *glue code*. The glue code provides function implementations that the GC will call to perform certain functions like

scanning, sweeping and freeing unused memory from the heap. Each runtime using the OMR GC policies requires a different set of this glue code. Often, this requires a few modifications to existing glue code to allow the OMR GC to understand the runtime specific object model.

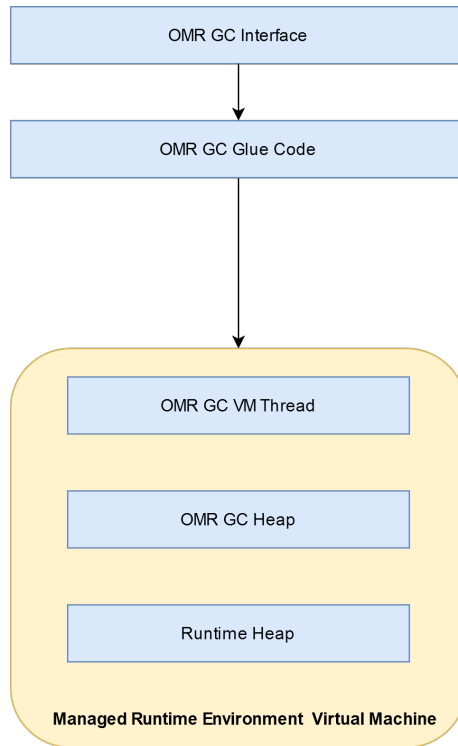


Figure 2.3: Eclipse OMR GC Architecture

Supported GC Algorithms

The OMR GC framework API supports different GC policies, designed to work for different workloads. The following schemes are supported:

Mark-sweep This is the standard mark-and-sweep GC which works as described in existing GC literature [81].

Compaction Avoids fragmentation by allocating memory contiguously.

Generational GC This allows for the implementation of GCs with a heap partitioned into generations.

Parallelism This uses multiple GC threads and halts mutator threads while the GC runs.

Concurrency This is a low pause algorithm that allows GC and mutator threads to mostly run simultaneously.

2.2.2 GC-as-a-Service Library

Another alternative for GC reuse is the GC-as-a-Service Library (GaS) by Wegiel and Krintz [142]. GaS takes the route of moving the GC logic into a modular library. The framework facilitates high levels of code reuse, but only allows a non-moving garbage collector to be utilized. This means that it supports virtual machines that assume objects do not move in memory [110, 142].

It provides a shared C library, which is accessible via the GaS interface that can be used by MREs for different languages (e.g., Java, Python, Ruby) to integrate garbage collection into their runtimes. Each MRE dedicates some number of threads to the GaS GC and maps a virtual memory region that GaS manages. MREs also have the option of allocating certain types of objects (e.g., immortal objects or internal data structures) in their private heaps and managing them independently of GaS [142]. GaS allows integrating high-quality GCs in MREs that do not have quality GCs. However, by treating GC as a component, it also supports GC research and experimentation for other non-GC MREs [142].

2.2.3 The Boehm-Demers-Weiser Conservative Garbage Collector

The *Boehm-Demers-Weiser Conservative Garbage Collector (BDWGC)* was initially intended to be a replacement for C's `malloc` and C++'s `new` [66, 65, 20] but also provided a C/C++ library to facilitate easy interoperability [64]. Over time, several programming languages now use the collector's C libraries and interface to build robust GCs. RPython has a BDWGC-based GC implementation, but we do not consider it for any of our evaluation.

The BDWGC is a conservative garbage collector that uses a modified mark-sweep algorithm and allows finalizers to be invoked when objects are collected. It is optimized to use type information if provided to locate pointers while also supporting incremental and generational collection [66].

The BDWGC supports a platform-dependent memory allocator that performs allocation at two levels, in large and small block sizes, using an estimated best fit algorithm by preserving free lists for several large block sizes. The allocator also permits allocation of various *types or kinds* of objects; each kind processes the GC differently depending on whether the objects are scanned for pointers or not. Separate free lists are maintained for each block size and object kind [66].

The collector also supports generational collection, using a basic concurrent and generational GC algorithm. For concurrent collection, the collector does not have a separate thread but instead runs in the allocation thread [20].

2.2.4 MMTk

Grounded on the theory that *complexity is the enemy of performance and security*, MMTk stands for *memory management tool kit* and is a library that helps in the rapid prototyping and evaluation of garbage collectors.

MMTk in Java

MMTk has been a researched GC framework for about 21 years, having its history in a research JVM built in Java called *JikesRVM*. Blackburn et al.[16] started MMTk GC research in JikesRVM trusting that any idiomatic Java code could be optimized by the compiler and if the compiler failed, then fixes would be made to the compiler to allow the optimizations. Therefore early versions of MMTk were written in a static dialect of Java. It was static, not allowing for the use of Java's `new` keyword because this led to infinite regression for especially meta-circular garbage collection. It also allowed for greater re-use and debugging of MMTk-based GCs [14].

The early versions of MMTk also greatly used Java's strong typing to create special types that identify addresses and guarantee the correctness of the GCs in the framework. A comprehensive evaluation of MMTk was done in *inko* and *Scala* runtimes and lead to significant findings on how to design high performance GCs [15]. This was possible because one would hold constant the role of the runtime, the compiler and the GC algorithm, to analyze several aspects of the GC. For example by holding the compiler and runtime constant and varying the GC algorithms, it is possible to study several aspects regarding locality across micro-architectures, which gave insights into how to develop new GC algorithms [14].

The results from this earlier analysis showed that from a software engineering perspective, there was a reduction in complexity cited from number of lines of code used in the implementation. This reduced code complexity for new MMTk-based GCs built in runtimes came without performance degradation [15].

MMTk in Rust

Following research that revealed that the *Rust* programming language gave safety and high performance guarantees when used for implementing garbage collectors [93], MMTk was re-written in Rust, and was released in November, 2020 as a shared library with VM-specific bindings [18].

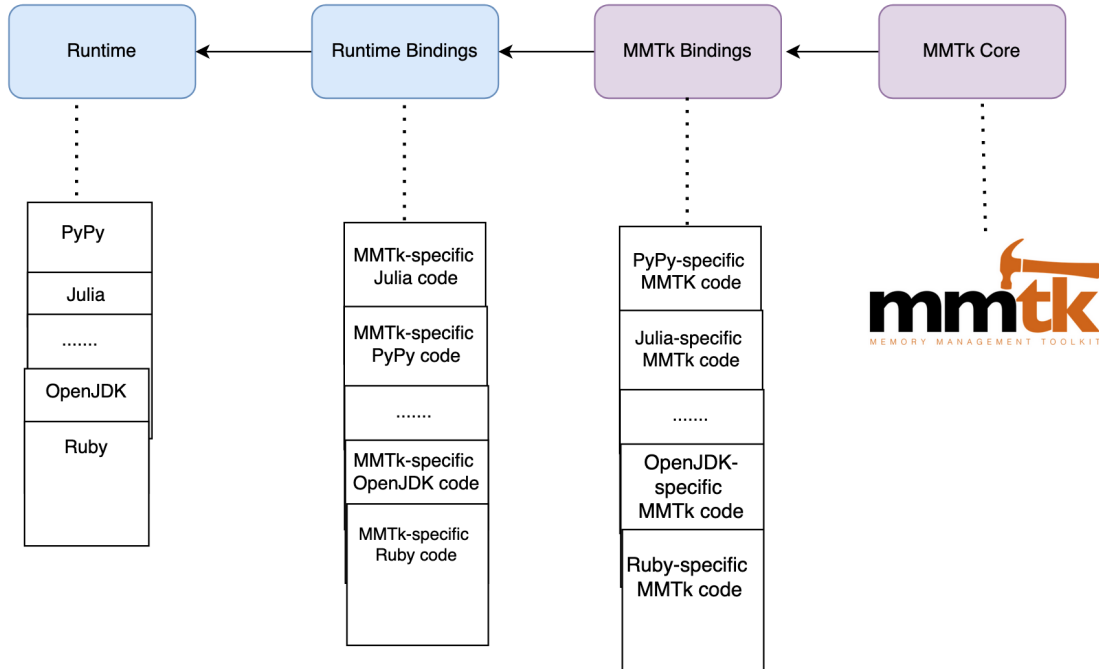


Figure 2.4: MMTk Design

With integrations in industry-level runtimes like JikesRVM, OpenJDK, V8 and many other runtimes, the new MMTk architecture is shown in Figure 2.4.

MMTk has four discrete components, the MMTk library, the runtime, the language runtime bindings and mutator code. The language runtime bindings and mutator code are not MMTk components. The runtime can be one of OpenJDK, V8, PyPy etc. that uses the MMTk library, which is the core of the garbage collector.

The runtime bindings consist of code that is written in Rust and is specific to the runtime; for example, object processing operations like scanning are written in the bindings because this code has domain knowledge about object model details that are specific to the runtime. The code in the runtime bindings is therefore specific to a runtime but is also in the hot path in MMTk.

The mutator code allows the runtime to have certain aspects in its hot path like write-barriers and allocation sequence to be written in C++ and inlined in the runtime, calling out to MMTk on its load path.

2.3 Garbage Collection in Python

The main topics and experiments of the dissertation detailed in Chapters 3, 4, 5, 6 and 7 are implemented in Python runtimes. The solutions are, however, general enough to be applied to other systems. This section gives a background on garbage collection for CPython, the reference implementation of Python, and PyPy, one of the alternative implementations written in Python. The work in Chapters 3, 4, 5 and 6 is based on PyPy, while Chapter 7 is based on CPython.

2.3.1 CPython

CPython is the reference Python implementation developed by Guido Van Rossum, and has been in existence for about 30 years [138]. As the name suggests, it is implemented in C. CPython’s design was aimed to be simple for most of its components, including the memory management system [46, 138]. Memory management in CPython is automatic with reference counting and complemented with a generational backup GC to handle cyclic garbage.

The Reference Counting GC

CPython uses the reference counting algorithm described in Section 2.1.3, and tracks references to every object using a reference count field in the object header. The default object header in Python has two fields, the *object type and reference count*. This object layout changes to accommodate extra fields required to perform garbage collection for cyclic garbage, discussed in the next section.

The Cyclic GC

CPython has a three-generation GC to handle objects that are involved in cycles since reference counting does not take care of this. The cyclic GC works for the most part until there are objects with finalizers, in which case garbage will not be

collected. The GC keeps track of container objects: integers and strings are ignored. The object layout is modified to accommodate additional linked list fields, `gc_prev` and `gc_next`, for all objects tracked by this GC.

The tracked objects contain another field, `gc_refs`. Initially, this field is equal to the object's reference count. For each container object, `gc_refs` is decremented for every reference to another object in the set of the already tracked container objects. This is the equivalent of the marking phase, since at the end of this process, objects with a zero reference count are considered garbage.

After this reference adjustment process, objects with a reference count value that is greater than one for the `gc_refs` field are moved to a different set as they cannot be freed since they have other references outside the set of container objects, i.e., other objects that are not tracked by the cyclic GC. Any objects referencing moved objects have their pointers updated and can also not be freed. The rest of the container objects in the original set are freed as they are garbage.

2.3.2 PyPy

PyPy is an RPython-based Python interpreter and as such uses garbage collection from the RPython framework.

The RPython Framework

The RPython programming language was developed simultaneously with the PyPy Python implementation [117]. In fact, earlier work on PyPy described the RPython framework and translation process as one of the components of PyPy, the standard Python interpreter being the other component [110, 117]. The framework is now maintained as a project of its own.

Therefore, for the purposes of the discussion in this work, PyPy is defined as just the Python implementation written in RPython. RPython is a restricted and statically

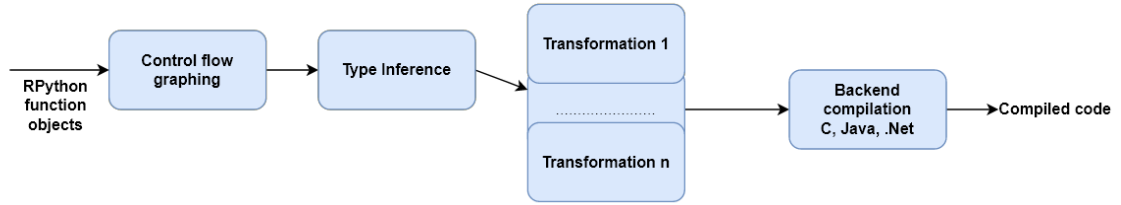


Figure 2.5: The RPython Translation Toolchain

typed subset of the Python language. It is not limited in syntax but more in how it handles objects of various types [117]. RPython-based languages and interpreters are compiled to various supported environments and platforms, using the RPython translation toolchain.

The Translation Toolchain

The RPython translation tool-chain takes in RPython function objects as input. To obtain the RPython function objects (essentially Python bytecode), the RPython program is run using, for example, a CPython interpreter. The translation toolchain, shown in Figure 2.5, consists of a frontend analysis, a sequence of transformations, and a backend that compiles the RPython program to a target environment like C [117].

The frontend part of the translation process analyzes the input RPython program, converting the bytecode to control flow graphs and performs type inference. The outputs of the frontend analysis are control flow graphs whose variables are typed. The first of the transformation steps takes the typed flow graphs, converting them to flow graphs with operations that are more familiar to the target environment. Transformations can be added to make more changes to the control graphs. One use case is as an alternative to adding a GC. The GC is implemented entirely in plain restricted Python, RPython, and will manipulate pointers and addresses of objects [117].

The RPython Tracing JIT

The output of the translation process is an executable that is a fully functional virtual machine with an optional JIT compiler. The JIT compiler is automatically generated from the interpreter written in RPython, but the JIT generator needs some JIT hints in the RPython program to facilitate JIT compiler generation.

An RPython target interpreter wishing to add JIT compilation must implement the `jit_merge_point` and `can_enter_jit` hints, to signal the start of opcode dispatch and the end of an application level loop respectively. The former allows the JIT to fall back to the interpreter when executing machine code is no longer desirable. The target interpreter should also define *green variables*, which are loop constants, that are used to identify the current loop iteration and *red variables*, which are anything else in the execution loop [23].

After type inference in the translation process, the translation driver adds a JIT compiler to the translating interpreter or program, and decides what assembler backend to use. Then, another routine searches the function graphs for the two JIT hints. The graph containing the `jit_merge_point`, also known as the portal graph, is rewritten to handle JIT exceptions, that give control to the interpreter on JIT exit. The `can_enter_jit` hint is replaced with a check to verify if the current loop is *hot* and suitable for compilation. The interpreter graphs starting with the portal graphs, are converted into an intermediate form (JIT bytecode) and then included in the final binary [23].

Tracing in the RPython JIT is performed by the metainterpreter, which interprets the JIT bytecode, recording each executed operation to a list of operations, called the *trace*. This interpretation continues until the `can_enter_jit` hint is encountered, which signals that a whole iteration of the application level loop has been traced. The trace produced by the metainterpreter is optimized before the JIT backend translates it into machine code [23].

The JIT compilation process performs GC related rewrites and transformations to lower some instructions, e.g., C++ *new* statements are changed to *malloc*. The trace also contains a recording of where the GC pointers are located, i.e., on either the stack frame or registers, to facilitate instructions that trigger garbage collection [23]. The integration of GC frameworks in PyPy described in Chapter 3 elaborates further on these rewrites.

Supported Garbage Collectors

PyPy garbage collectors are written in RPython, which assumes automatic memory management. The GC program is analyzed like any other program during translation. Low-level calls, like C code, do not assume automatic memory management. The GC needs to support allocation for both Python visible objects and internal interpreter objects, e.g., lists, instances, etc.

According to the PyPy community documentation [25], existing GCs written in languages other than RPython can be reused in PyPy. A translation step can be designed that turns these GCs into RPython or, even better, into an RPython-level intermediate representation. The translation toolchain discussed earlier can then integrate the GCs into the virtual machines it generates.

PyPy has several GC policies implemented in its garbage collection framework. The GC to be used in the translated RPython program is set using the `--gc=NAME` option [36]. This option is used when translating an RPython program. For typical PyPy use, the interpreter is translated with this option. The available GC options in the framework used in PyPy are detailed here.

The Semispace Copying Garbage Collector It uses two arenas of equal size. Objects are allocated in one arena. When it is full, the live objects are copied into the second arena using Cheney’s algorithm [30].

The Generational Garbage Collector This collector is a two-generation GC and is implemented as a subclass of the *semispace* copying collector. It adds a nursery, which is a part of the current semi-space [36].

Hybrid Garbage Collector This is a three-generation GC and is implemented as a subclass of the Generational Garbage Collector. It can handle both objects that are inside and objects that are outside the semi-spaces. Large objects are allocated outside the semi-spaces as external objects to avoid costly moves. Small objects that survive for a long enough time are also made external so that they stop moving [36].

Minimark Garbage Collector The *minimark* is a two-generation GC. The main difference with the Hybrid Garbage Collector is that the non-external objects are directly handled by the GC's custom allocator, instead of being handled by `malloc()` calls as external objects [36].

Incminimark Garbage Collector PyPy currently uses the Incminimark GC by default, which is an incremental, generational moving collector. It has two generations. The nursery size can be varied depending on the workload and desired cache sizes by configuring the `PYPY_GC_NURSERY` environment variable [36]. The live objects that survive a minor collection are promoted to the old generation i.e., arenas, or malloced if they are very large. The major collection for the old generation is incremental, that is to say, it is done in intervals. The Incminimark GC is a fairly coarsely incremental GC which piggybacks a bit of major garbage collection work onto the end of each minor GC while most other incremental GCs are more fine grained and mainly piggyback on allocation.

2.4 Memory Layout of Types in Dynamic Languages

Contrary to static languages, types in dynamic languages are indeterminable at compile time. Types are instead associated with runtime behavior, which results in a need to devise schemes to identify the types of values at runtime. This category of languages is said to support *dynamic typing*. The memory layout in dynamic languages is such that there is a uniform representation of all objects regardless of type with a common *object* type.

In practice, dynamic languages have to also ensure the possibility of converting between specific types like integers, strings, etc., and the uniform *object* type, as well as determine what type of object is being represented. For this reason, Just-In-Time (JIT) compilers defer optimizations until run-time, when such languages can identify types of objects [24, 60, 45].

This flexibility comes with a performance cost relative to statically-typed languages, a problem this dissertation investigates in Chapter 5. Here we discuss the different ways types are represented in dynamic languages.

2.4.1 Terminology

We use the following terminology described by Gudeman [60] to describe representation schemes for types in dynamic languages.

Type This is an implementation type like integer, string, etc.

Wrapping Conversion from a specific type to a dynamic or general type, e.g., *integer to object*.

Unwrapping Converting back to a static type.

Wrapper Representation of a wrapped value.

Boxing Creating an indirect wrapper to a type, e.g., representing an integer with a tagged pointer is boxing.

Cost of an operation The number of machine cycles required to execute the operation on a machine.

Word This is the size of an object that fits in a general-purpose register.

2.4.2 Schemes for Memory Layout of Objects

Gudeman surveyed and evaluated several strategies for representing types in dynamic languages [60]. The canonical schemes can be classified as *tagged words*, *partitioned words*, *object pointers*, *large wrappers*, and *typed locations*. A combination of these techniques can be used for a hybrid representation.

Tagged Words

This representation scheme uses a *tag field* to represent the type of an object. The object is represented as a sequence of bits, with one or more tags storing the type of the object and the rest storing other object details (value).

This scheme leads to compact memory layouts and relatively good access time, but not all unwrapped values can be represented as wrapped values. Type information must also be restricted to have enough representations for values of each type, which is a problem when there is a need to support user-defined types.

Extraction of the details of an object is called *untagging*, while *tagging* is the process of setting the tag and value fields of the tagged word representation of an object. There are different considerations for tagging various types of objects, and care must be taken for each case. Tagging pointers and unsigned values have similarities while tagged integers and floats have special considerations [60].

Partitioned Words

This scheme allocates each type to a certain subset of the available bit patterns. Each type is therefore restricted to representing those values that it can represent in the bit patterns allocated to it. The entire word in this strategy is legitimate as opposed to only the value field being legitimate for the tagged word representation [60].

Object Pointers

This representation scheme uses a machine pointer to a memory block containing all type information to represent each wrapped value. The object is therefore a structure that consists of the required information to identify what type of value the block represents [60].

Large Wrappers

The schemes discussed above all use a single word to represent the object. Large wrappers use more than a single word to represent both the type and a complete machine value. This scheme requires more registers, uses more memory, and incurs higher costs in loading and storing wrapped values than the single word schemes. There are optimizations to reduce this cost for large wrappers but they are not applicable for single word schemes [60].

Typed Locations

A *typed location* allows for determining the type of the pointer based on where it is located on the stack, register, or any place in memory. In many static languages, locations rather than values have types, i.e., the type of any value is determined by whether the value is on a stack, register etc.,. Dynamic systems can also have a memory layout where types have type codes, but the value and the type code are located in separate places [60].

Hybrid Schemes

To achieve the advantages of more than one representation scheme, it is possible to combine one or two of the schemes discussed above.

Pure object pointer and tagged word schemes are rare as for the former, some values are small enough to be represented in one word while the latter are very restrictive, thereby not providing a large enough range of type codes for all of the types that are needed, especially in languages that support user-defined types. This necessitates a two-stage scheme where the first is based on tagging using the lower two bits while the second is an object pointer scheme [60].

The tagged words and partitioned words schemes can also be used together by using a value field to represent some types of unwrapped values and utilizing the whole word to represent other types [60].

In some cases with the assumption that not all wrapped values have the same size, some values can be represented as tagged words and others as double large wrappers [60].

For collections, a hybrid with typed locations can be used for structures that have elements of the same type. In this case, the cost of storing a type for each element is eliminated since the element type is determined from where it is placed. A hybrid of type locations can be used with tagging and double wrapper representations [60].

2.4.3 Specialized Optimizations for Collections

An integral part of recent research on the representation of types for dynamic languages has focused on optimizations targeting the design of efficient collection libraries [24, 34, 9, 42, 68, 94]. We discuss two of these prominent optimizations, namely *storage strategies* and *element kinds*.

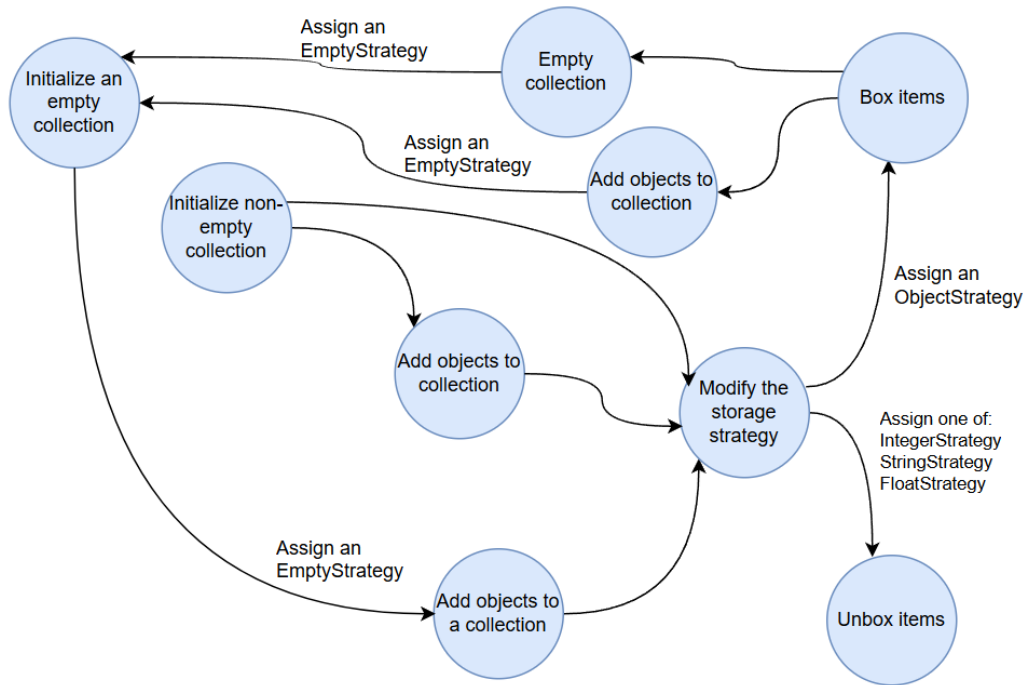


Figure 2.6: Storage Strategies [24]

Storage Strategies

Aiming to solve the challenges of pointer tagging for type representation, storage strategies optimize the treatment of collections [24]. Storage strategies were proposed with two assumptions, 1) that homogeneous collections seldom de-homogenize and 2) when they de-homogenize, this happens when a collection has a small number of elements.

The design is such that each collection references a storage strategy and storage area in memory. Similar to the strategy design pattern, the storage strategy manages all operations related to a collection but also how data is laid out in the storage area. Figure 2.6 shows that collections can evolve through several storage strategies during program execution starting with an *EmptyStrategy* for an empty collection. A collection gets a specific strategy when items are added to it. The specialized strategy unwraps the elements and stores them.

When an element of a different type is added to a collection, the collection is assigned a general *ObjectStrategy* and each element is boxed returning to the equivalent of the

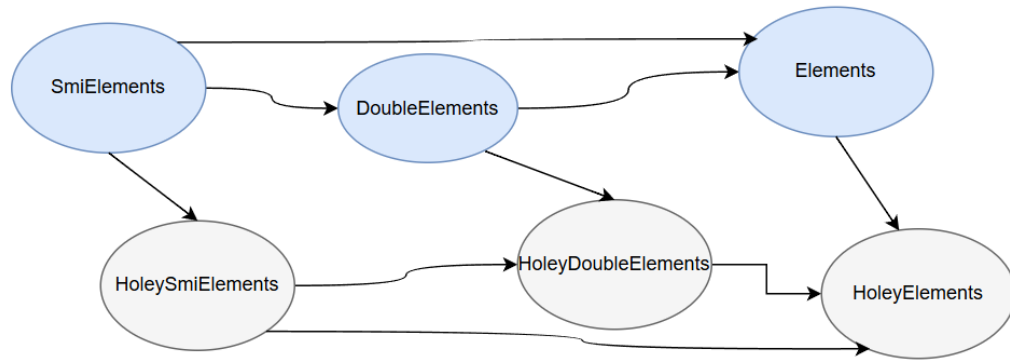


Figure 2.7: Element Kinds in V8 [34]

default representation of the collection in dynamic languages.

Storage strategies are generally more efficient than pointer tagging but for some corner cases, they actually perform worse compared to when they are not used. One of the cases is where a large homogeneous collection de-homogenizes. This results in the re-boxing of each element in the collection, an operation with high overhead. The changes in strategies are also a source of overhead. To solve this, the *V8's element kinds*, discussed next, limit the frequency of strategy transitions.

Element Kinds

V8's element kinds were developed during the same time as storage strategies [34]. Both approaches share the same underlying object model where collections with elements of the same type are unboxed and stored in a designated area in memory. The representation of elements of a collection is called *ElementKinds* as shown in Figure 2.7.

Elements in a collection are represented in six ways depending on their type and whether the collection has holes. A spatial array is an example of a collection with holes, i.e., with missing items in some memory cells.

A list of integers will have a special kind of *SmiElements*, adding a float to the list will transition it to a *DoubleElements* kind. Similarly, when the list gets a hole, it will be transitioned to a respective *Holey* kind like *HoleySmiElements* for integers.

The paper on ElementKinds discusses support for only *Integer* and *Double* types and it is not confirmed anywhere if other types like *Strings* are supported [3].

A shortcoming for both approaches is the fact that they both do not optimize heterogeneous collections. We propose an alternate approach to handle this in Chapter 5.

2.5 Collection Presizing Schemes for Dynamic Languages

The performance overhead and resource wastage due to shrinking and expanding collections in dynamic languages has been addressed using presizing together with other optimizations like pretenuring and pretransitioning [27, 94, 68, 96], with anecdotal details on implementation and evaluation.

Presizing is a technique in which profiling information is used to create an optimal internal size of a collection. We discuss some of the techniques from the literature on this problem, which include *mementos*, *recycling*, *lazy creation* and other language specific methods.

2.5.1 Mementos

Allocation mementos as proposed by Clifford et al. [34], are temporary objects allocated next to an object they track for purposes of storing profiling information about the object to allow for further optimization. Mementos are created at either the object's call-site or allocation site and live for a short time typically, only surviving the closest garbage collection cycle.

For presizing, mementos are used to store the sizes of collections from the last run at a given allocation site or call site so that in future, collections from a given site

are allocated an internal data structure with slots of an observed size to avoid the overhead of having to resize the collection.

Mementos do not take up any space in the object and are observed to have no engineering complexity, having been achieved in roughly 150 lines of code for presizing in the V8 runtime but they have overhead associated with their operation. They are also tied to the garbage collector and so runtimes like Graal VM could not directly emulate or use them for presizing [68].

2.5.2 Recycling

When a collection expands from a user request to a point where the internal structure has to be resized, the process involves creating a new larger collection, copying the items from the older collection and garbage collecting the old unused collection.

Bergel et al. propose not garbage collecting the unused collections [9], and instead reusing them for future requests that fit the specifications of the now unused internal collection. This may avoid creating many internal collections but does not take away the overhead from copying of items from the old to the new internal collection.

2.5.3 Lazy Creation

When a collection is initialized in dynamic languages, an internal collection is immediately created and over-allocated with the assumption that the collection will expand eventually, which is not true in some cases.

Lazy collection creation or initialization is a technique where the internal slots of the collection are not allocated until a request to add items to the collection is encountered [9]. This avoids extra slots that may have been allocated, which may have been more than required, hence wastage.

2.5.4 Language-specific Approaches

In the GraalPython project, due to the garbage collector requirements of mementos, the creation and append operations of Python lists were intercepted, observing the sizes of collections, lists in this case, and using this information to compute the optimal sizes for all other collection instances using the same allocation site [68].

This profiling information is collected during interpretation to reduce performance overhead. This approach generally avoids most of the bookkeeping overhead of mementos but does not consider the allocation calling context, limiting its applicability. The calling context captures the control flow of the program in branches.

2.6 Memory Management for Native Extensions

Most language virtual machines (VMs) for dynamic languages provide a C/C++ library that allows host programs and applications written in C/C++ to communicate with the language [76]. These host programs are what we refer to as *native extensions*. An example is NumPy and SciPy that call Python's C API [116, 92, 90]. Several other programming languages like Perl, Ruby and Lua support similar native extensions by implementing an API between the language and C [127, 62, 63].

As discussed by Ierusalimschy et al. [76], the ability to call languages like Lua, Ruby, etc. to and from C is what makes them *extension*, *embeddable* and *extensible* languages. They are *extension* languages because they are used for extending applications through configurations, customizations and macros by calling the language from C.

The ability to call C from the language allows extending the language with new features by the use of C functionality, making it *extensible* [76]. Embeddability makes it possible to enrich C/C++ applications by implementing some of the functionality in another language like Python or Ruby. Extension is different from embedding since for the latter, the main program may not be written in the language its embedded in.

Supporting extension modules for most languages has two main challenges, 1) reconciling C's static typing and the language's (typically) dynamic typing and 2) automatic memory management, a concept foreign to C. This dissertation focuses on the memory management challenge [116, 76, 62, 63] in Chapter 7.

Memory management and garbage collector design to support native extensions depends on the communication between the language VM and C. We provide a background on how garbage collection is achieved depending on how a language represents its values in C. The topics covered include *union types*, *the abstract stack*, *handles* and finally the *proxy object model*.

2.6.1 Union Types

Ruby, Perl, Python and even earlier versions of Lua have a corresponding C object that represents a language value or object. For example, *lua_object*, *PyObject* and *VALUE* for Lua, Python and Ruby respectively. They are, in most cases, abstract types that refer to language values opaquely in C. The semantics of these types may change but they are called *union types* because they refer to any type of object or value of the VM language.

Representing language values as union types makes the design of a garbage collector onerous since it is difficult to identify roots and eventually all referenced objects, because it is not trivial without extra knowledge for the garbage collector to know if a value is referenced by a union type from a C routine or program. The garbage collector may collect a value leading to a dangling pointer, the union type [76, 62].

Python and Perl manage this memory by requiring the programmer to perform explicit reference counting of the union objects using provided functions in the C API. By incrementing the reference count of a language value, the garbage collector does not collect it until its reference count drops to zero through a decrement. This is a complex and error-prone task for the programmer.

Another alternative is to scan the C stack and registers for the union type. The scanning algorithm should ignore any data that is not the union type, and being conservative like the Boehm GC, it must devise ways of identifying the addresses that correspond to the start and end of a stack. This algorithm helps identify GC roots associated with C Code. Ruby uses this technique but it has a drawback of being non-portable and platform-specific. To identify all references between objects, Ruby extension authors must implement *mark* functions for every Ruby data type in C that references other Ruby objects [76, 62], a task that is also error-prone. The mark functions help with tracking live objects in the context of a mark-and-sweep garbage collection.

2.6.2 Handles

Handles are a common concept used in Javascript (V8), C/C++ API and HPy, an alternative C API aimed at solving Python's C-API challenges. Handles hold a C/C++ reference to language objects. Handles play the same role as the union types like Ruby's `VALUE` and Python's `PyObject` but with a difference. The difference is that union types refer to any language object and can thereby be interchangeably passed to C API functions while in most cases each type of handle should be managed independently [63, 41, 38].

Handles are significant and simplify memory management for native extensions in two ways, 1) they act as GC roots and 2) they provide a level of abstraction that allows objects to be moved in memory without breaking any C references. From a GC perspective, GC roots therefore correspond to live language objects and any handles from the native extensions. There is also no need for programmer intervention in finding references between objects since they exist as regular references [63, 41, 38]. By design, several kinds of handles can be supported, V8 supports three types of handles, i.e., *local*, *persistent* and *global*, each having different semantics. Local

handles are those created on the C++ stack while persistent handles are those not on the C++ stack and global handles are similar to persistent handles, but can be moved according to defined C++11 move semantics [63].

2.6.3 The Abstract Stack

A significant challenge with using the union type naively is that, if the union type is a pointer to the language's internal data structures, it is difficult to implement a garbage collector that moves objects around in memory. To make the union type more correct and allow for moving of objects, the semantics can be changed so that it is instead an index into an internal array storing values designed to be passed to C. The array is erased when the C function returns thereby triggering garbage collection for the values used by the C function. This was used in Lua 2.1 but it came with two main challenges, namely sometimes the union type (`lua_object`) value remains active for longer or shorter than the life cycle of the C function that produced it. These challenges associated with the union type led to its replacement in Lua 4 with an explicit *abstract stack* [76].

An abstract stack is different from the C stack and like the union type is used for interoperability between the language and C. Each C API function has a stack frame that contains the function arguments and any results returned by the function to the language. The stack holds values of any type of the language and so the C API functions also operate with any language type. Language values used by C code are also stored on the stack, the garbage collector does not collect any language values on the stack. Any language values used by C code are collected when the C function returns if no other references to them exist, and not if otherwise [76].

2.6.4 The Proxy Object Model

A common design in the support of native extensions specifically for garbage collection is to maintain two *views* or *representations* of the object depending on where we are i.e., the language side or the C side. The idea is to have a language object, like a Python or Lua object, to act as a proxy for a C API object. This technique has been used in both Lua and PyPy. When a language object is passed to C, the C counterpart is initialized and allocated [76, 135].

Synchronization mechanisms are required to ensure a link between the C object and its proxy; this is usually achieved by a reference, but this reference should not prevent collection of the proxy object when it becomes inaccessible. Also, the lifetimes of the proxy and the C object should be synchronized; the C object should live in memory as long as the proxy object lives and vice versa [135].

Chapter 3

Eclipse OMR-based Garbage

Collection for Meta-tracing-JIT-based

Dynamic Languages

Mainstream programming languages have adopted garbage collection over the years, freeing programmers of the burden to ensure that unused memory is freed, and that released memory is not accessed. However, implementation of garbage collectors (GCs) still remains a complex task. This complexity leads to GCs that are incorrect, compromised in security (because implementation of security features is hard), and also difficult to maintain over time [93]. Effective evaluation of GC algorithms requires that garbage collectors are easier and faster to implement, but this is hard to achieve without modularity and transferability in GC design. This chapter explores GC modularity using the Eclipse OMR GC framework.

3.1 Introduction

The Eclipse OMR project solves modularity and transferability by providing cross platform components that developers can use to easily implement different aspects of

a runtime. One of the components it provides is a GC framework for managed heaps, with a wide range of robust, sophisticated and high performance GC policies [49].

The Eclipse OpenJ9 project is the biggest consumer of the Eclipse OMR project, and has taken advantage of the sophisticated GC policies in the framework to improve GC performance. However, it is unclear if garbage collection modularity through a framework like Eclipse OMR has any advantages to offer to dynamic languages.

We discuss the design, implementation and initial evaluation of OMR-based garbage collection for RPython-based dynamic languages and Virtual Machines (VMs). The RPython framework provides a translation toolchain with reusable tracing Just-In-Time (JIT) compilation, and garbage collection (GC) that VMs written in it inherit. This toolchain has provided an efficient way for implementing dynamic programming languages and VMs. The VMs are written in RPython, a statically-typed language for writing dynamic languages, and translated by the translation toolchain, inserting the JIT and GC functionality in the process. The result is an interpreter compiled to one of the supported backends, i.e., C, Java or .NET [117]¹.

We apply the OMR GC implementation to two RPython-based languages, namely, PyPy, a Python implementation, and PyHP, a PHP implementation. We then compare an OMR-based mark-and-sweep garbage collector to an equivalent highly optimized mark-and-sweep GC in the RPython GC framework. The RPython mark-and-sweep GC is used for purposes of fair comparison, but it is no longer part of the RPython GC framework.

The software engineering metrics agree with existing work that using a GC framework results in a GC that is simple, easier to maintain, and has fewer bugs [14], and yet the OMR GC still implements a wide range of GC algorithms. On Python benchmarks, evaluated on PyPy, the OMR GC improves collection speed by typically 6% to 20% (with outliers as high as 74%) compared to the purely Python-based

¹PyPy's CLI and Java backends are not as complete as the C backend

one in the original RPython mark-and-sweep GC attributable to an underlying high performance C/C++ implementation. Due to structural and algorithmic differences that affect allocation speed and in some workloads traversal speed, the OMR GC performs worse in total time by about 20% to about 35% with some outlier workloads having overhead as high as 53%. C's `malloc` and `calloc` allocation API perform at a rate of about 10% better than OMR's allocation of objects. C's allocation API is used in the RPython GC framework to allocate large objects in some cases.

Therefore, the main contributions in this chapter can be summarized as:

- We implement an OMR-based garbage collector for tracing JIT based virtual machines that are written in RPython.
- We apply the OMR GC to PyPy, and PyHP.
- We show that a GC framework like OMR can achieve modular and maintainable GCs that can still be high performant for dynamic languages.

The rest of this Chapter is organized as follows: Sections 3.2, 3.3 and 3.4 describe the design, implementation and evaluation of OMR GC for RPython-based interpreters. Section 3.6 details related work.

3.2 Design and Implementation

Transformations in the RPython framework are used to achieve different tasks, like optimizations, exception handling, etc. As aforementioned in Section 2.3.2, after the type inference phase of the translation process, the first transformation converts the typed flow graphs to a low-level format, close enough to the target platform, e.g. C, .NET and Java.

The graphs are updated with platform-specific operations that match the corresponding RPython routines in the program being translated. The translation toolchain

supports C and object-oriented backends. There are therefore two different implementations of the first transformation. One of them changes the typed graphs to a format closer to C; this first transformation variant is called *LLTyper*. The other variation, called *OOTyper*, converts the graphs to the object-oriented platforms.

We complement the existing transformations by adding another transformation to the toolchain to handle OMR-based garbage collection as shown in Figure 3.1. We also implement a new exact OMR-based garbage collector for the RPython framework. The OMR GC transformation converts all C-level *malloc* operations to allocation calls from the OMR garbage collector. The OMR GC transformation also invokes collection routines from the OMR garbage collector to free memory for all dead objects.

The mapping of routines between the OMR GC and its associated transformation is not completely obvious. For example, a simple *malloc* leads to modifying the control flow graph with new functions from the OMR garbage collector. This code is also written in Python, but is capable of manipulating objects that are typed with a low-level type system [117].

The garbage collector calls operations from an intermediate OMR layer, the glue code. We implement all RPython object model specific details in the glue code, and any other runtime dependent aspects. While the GC in RPython is shared by all the resulting interpreters, we do not need to implement a separate set of glue code for each of the specific languages.

In terms of implementation, the steps of GC integration can be summarized as: we add a new OMR-based GC to the RPython framework by adding a new transformation, as pointed out earlier, instead of using the default *framework* transformation. We also implement the actual OMR-based GC, configure the GC, and perform JIT compiler integration.

Like the other GCs in the RPython framework, the OMR-based garbage collector is

one call.

As discussed earlier, the OMR GC component exposes a C interface for use by runtimes. It is a C interface backed by the C++ code that is the glue code. The RPython framework requires another set of glue code that replicates and customizes the code in the OMR GC glue so that the OMR-based GC can understand the implementation of objects specific to RPython. This replacement glue code is also written in C++ with a C interface.

For example, as shown in Listing 3.1, the OMR GC transformer defines a method `gct_fv_gc_malloc` (line 13–26), which replaces allocation calls in the translating program with a call to an OMR object allocation routine, denoted by the function pointer `omr_gc_alloc_ptr` (line 9). The function pointer points to the GC allocation functionality in the OMR GC, defined in a GC class accessed on lines 8–12.

```
1 class OMRGCTransformer(GCTransformer):
2     malloc_zero_filled = True
3     FINALIZER_PTR = lltype.Ptr(lltype.FuncType([llmemory.GCREF],
4         lltype.Void))
5
6     def __init__(self):
7         ...
8
9     if hasattr(GCClass, 'omr_gc_alloc'):
10         self.omr_gc_alloc_ptr = getfn(GCClass.omr_gc_alloc_ptr,
11             im_func,
12             [s_gc], SomePtr(lltype.Ptr(ARRAY_TYPEID_MAP)),
13             minimal_transform=False)
14
15     def gct_fv_gc_malloc(self, hop, flags, c_size):
16
17         funcptr = self.malloc_fixedsize_ptr
18         opname = 'omr_malloc'
```

```

18     tr = self.translator
19
20     if tr and tr.config.translation.reverse_debugger:
21         v_raw = hop.genop(opname, [c_size], resulttype=llmemory.
22                             GCREF)
23     else:
24         v_raw = hop.genop("direct_call",
25                             [omr_gc_alloc_ptr, flags, c_size],
26                             resulttype=llmemory.GCREF)
27     return v_raw

```

Listing 3.1: The OMR GC Transformation

3.3 Software Engineering Metrics

Using the Eclipse OMR GC framework for implementing a new GC in the RPython framework has evident impact in code metrics and performance. This evaluation uses standard formulas from the literature [106, 13].

3.3.1 Methodology

The results used in the comparison are generated using the *radon* tool [89]. Radon is a Python tool and accurately accounts for the Python/RPython code of the GCs. The glue code accessed using *rffi* is accounted for to a small degree, i.e., the initialization. The full metrics for the C/C++ glue code are determined through manual inspection.

3.3.2 Raw Measurements

We measure the number of methods, number of bytes, lines of code (*LOC*), logical lines of code (*LLOC*), standard lines of code (*SLOC*), and the cyclomatic complexity (*CC*) for both GCs in Table 3.1.

	OMR Garbage Collector	RPython Garbage Collector
Number of Methods	22	26
Size in Bytes	11.5KB	26KB
LOC	338	630
LLOC	224	538
SLOC	249	567
$\sum CC$	42	98
\overline{CC}	1.75	3.5

Table 3.1: Raw Measurements

The OMR GC uses four fewer methods and 14.5Kb fewer bytes compared to an equivalent mark-and-sweep GC implemented from scratch in the RPython Framework. This is slightly more than 50% fewer bytes per method in favour of the OMR GC. These raw metrics reflect the Python implementation of both GCs. There is additional C/C++ code that constitutes the glue code for the OMR GC with about 15 methods. The OMR GC is also implemented in about 50% fewer LOC, LLOC and SLOC. LOC is the number of physical lines of code, including any line breaks, while SLOC, excludes the breaks, thereby corresponding to the actual lines of code. LLOC measures the total number of logical statements, i.e., one line of code may have one or more logical lines of code, or zero if it's a comment. The framework mark-and-sweep GC has about 10.5 more LLOC, 16.8 more SLOC and 8.8 more LOC per method. We also recorded a 57% better measurement in terms of the cyclomatic complexity [106] for the OMR-based GC implementation. The cyclomatic complexity is calculated by analysing the abstract syntax tree (AST) [1] of the Python program that implements the GC. The difference in the cyclomatic complexity mean is also 50%, in favour of the OMR GC.

3.3.3 Halstead Measurements

Table 3.2 shows a measure of the relations between software properties, i.e., the Halstead metrics [106]. The numerical conclusions are drawn statistically from the

	OMR Garbage Collector	RPython Garbage Collector
n_1	8	15
n_2	44	153
N_1	29	135
N_2	56	252
Program vocabulary	52	168
Program length	85	387
Calculated program length	264.21	1168.98
Volume	484.54	2860.82
Difficulty	5.09	12.35
Effort	2466.74	35339.63
Time	137.04	1963.31
Bugs	0.16	0.95

Table 3.2: Halstead Measurements

source code. By definition, n_1 refers to the number of unique operators, n_2 is the number of unique operands, N_1 is the total number of unique operators, N_2 is the total number of unique operands. The following are the key formulas for the properties shown in Table 3.2. Given:

1. Program vocabulary, n
2. Program length, N
3. Calculated program length, L
4. Volume, V
5. Difficulty, D
6. Effort, E
7. Time required to program, T
8. Number of delivered bugs, B

	OMR GC	RPython Garbage Collector
Lack of cohesion of methods (LCOM)	2	1.5
Maintainability Index (MI)	31.28, A	14.12, B

Table 3.3: MI and LCOM

Then :

$$n \equiv n_1 + n_2 \quad (3.1)$$

$$N \equiv N_1 + N_2 \quad (3.2)$$

$$L \equiv n_1 \log_2 n_1 + n_2 \log_2 n_2 \quad (3.3)$$

$$V \equiv N \log_2 n \quad (3.4)$$

$$D \equiv \frac{n_1}{2} \cdot \frac{N_2}{n_2} \quad (3.5)$$

$$E \equiv D \cdot V \quad (3.6)$$

$$T \equiv \frac{E}{18} \quad (3.7)$$

$$B \equiv \frac{V}{3000} \quad (3.8)$$

$$(3.9)$$

From the metrics in Table 3.2, the OMR-based GC has a 60% better difficulty level and requires about 14× less effort to implement. It also takes about 14× less time to develop the OMR-based GC with an 83% decrease in terms of defects. This is highly due to the fact that the OMR GC calls a framework, which simplifies implementation and takes relatively less time.

3.3.4 LCOM and Maintainability Index

We also compare the Lack of Cohesion of Methods (LCOM) and Maintainability Index (MI) [106] for both GCs in Table 3.3.

The OMR-based GC performs worse in terms of the LCOM, that is the implementation

is such that methods are not very interrelated. As described in Section 2.3, RPython GCs reuse code from the framework transformation, and some inherit other GCs, so an OMR-based implementation for one policy like the mark-sweep is bound to not be as connected with the rest of the code.

The following formula is used to compute the maintainability index.

$$MI = \max\left\{0, 100 \frac{171 - 5.2 \ln V - 0.23G - 16.2 \ln L + 50 \sin \sqrt{2.4C}}{171}\right\} \quad (3.10)$$

Where:

1. V = Halstead volume
2. G = Total cyclomatic complexity
3. L = SLOC
4. C = Fraction of comment lines in radians

The maintainability index does not usually reflect actual maintainability of software [106]. It is an approximation that should not be as trusted as the other metrics presented [126]. From this calculation, the OMR-based GC has a higher maintainability index advantage over the framework GC. This is due to a high level of modularity and reuse.

3.4 Performance Evaluation

We implement a set of Python memory-intensive benchmarks [104] that are modeled after the *Pyperformance* [113] benchmark suite for comparison between the OMR and the RPython GC. We choose Pyperformance benchmarks because they are standard Python benchmarks and are modeled after real-world Python applications.

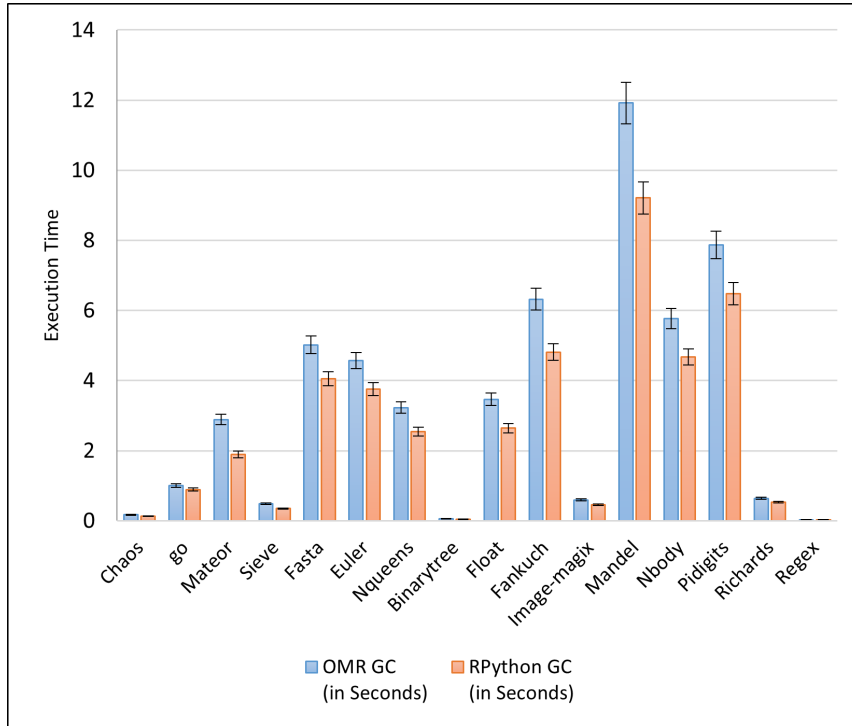


Figure 3.2: Mutator Time

To account for the JIT warm-up, we run each benchmark 25 times, ignoring the first 5 iterations. This is not aimed at determining accurate steady-state because it is impractical, but rather a large number of iterations increases the likelihood of getting close to steady-state. All benchmarks are run on an Intel(R) Xeon(R) Gold 6248 CPU @ 2.50GHz machine, running 64-bit Debian 10.2.1 with GCC 10.2.1, using 10 cores, as well as caches characteristics of L1d: 192 KiB (6 instances), L1i: 192 KiB (6 instances), L2:1.5 MiB (6 instances), L3:12 MiB (1 instance).

3.4.1 PyPy

Figures 3.2, 3.3 and 3.4, show the mutator, total and GC time for the PyPy interpreter. The mutator time includes application, compilation and allocation time, while GC time includes traversal time and time taken to collect dead objects. The total time includes both mutator and GC time.

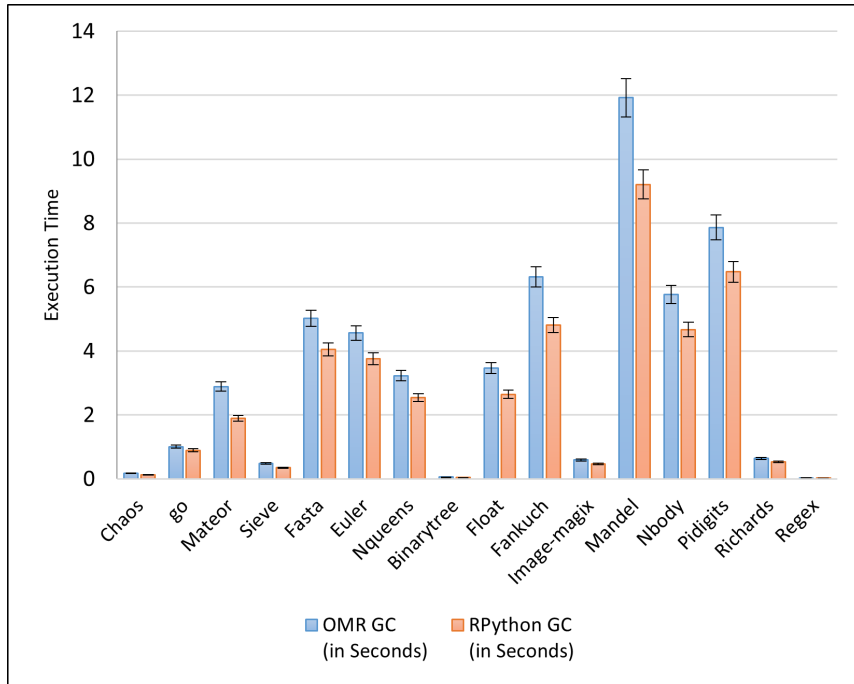


Figure 3.3: Total Time

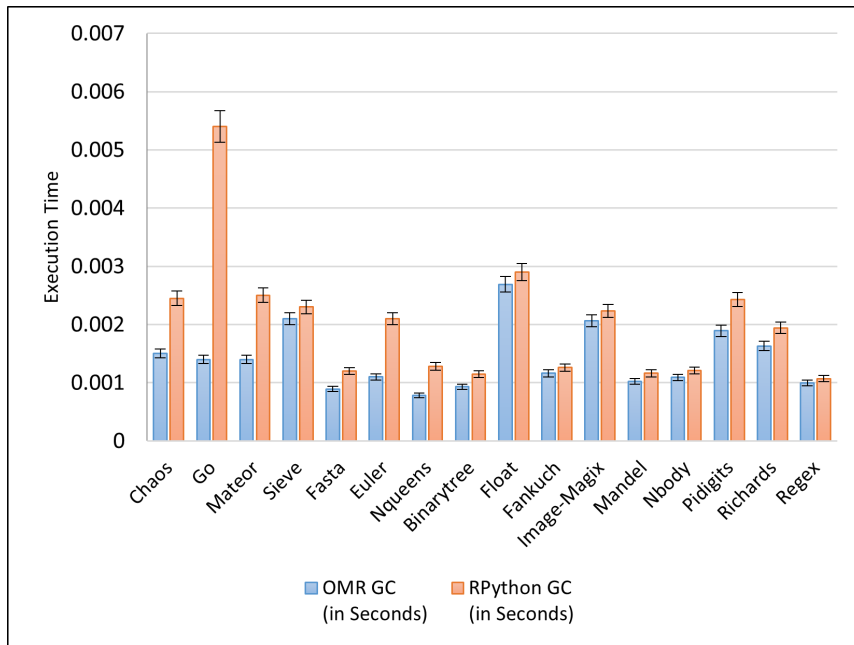


Figure 3.4: GC Time

Due to algorithm differences, the OMR-based GC is typically 20%–35% slower in mutator time compared to the RPython GC, mostly due to allocation overhead. This overhead is as high as 53% for workloads such as *meteor*. Algorithmic impacts can mean differences in implementation of garbage collection routines and structural aspects like FFI calls. The RPython mark-and-sweep GC we compare to is highly tuned, and supports thread cloning, something not supported yet in the OMR implementation. The OMR GC typically performs 6%–20% better during collection with some outlier workloads like *go* having speedups as high as 74%, but because the magnitude is smaller than the degradation from allocation, the total speed still shows worse performance. For some workloads, traversal during GC is also 2% slower for the OMR GC, which also contributes to the poor execution time in some benchmarks. This speed is likely to improve if we make use of OMR’s better GC policies, and we further gain from the fact that we are calling a C/C++ library instead of purely Python code. This is likely advantageous given that in RPython, some components of the interpreter are written in C when performance is critical.

3.4.2 RPython-based VMs

We also implement eight benchmarks written in a similar way for both Python and PHP, shown in Figure 3.5. The benchmarks contain simple calls, list and loop processing, string concatenation and Fibonacci. We find that even if we keep constant the GC, the application and how a language is implemented, the garbage collector has different performance impact for two different RPython-based interpreters. The OMR GC has a smaller overhead for PyHP than PyPy as shown in Figure 3.5, even though the PyHP implementation is not as robust and complete as the PyPy one, in terms of using the RPython toolchain. The difference in performance can be due to interpreter specific aspects, for example, some image-based virtual machines like Squeak load an entire live programming language before invoking the first bytecode,

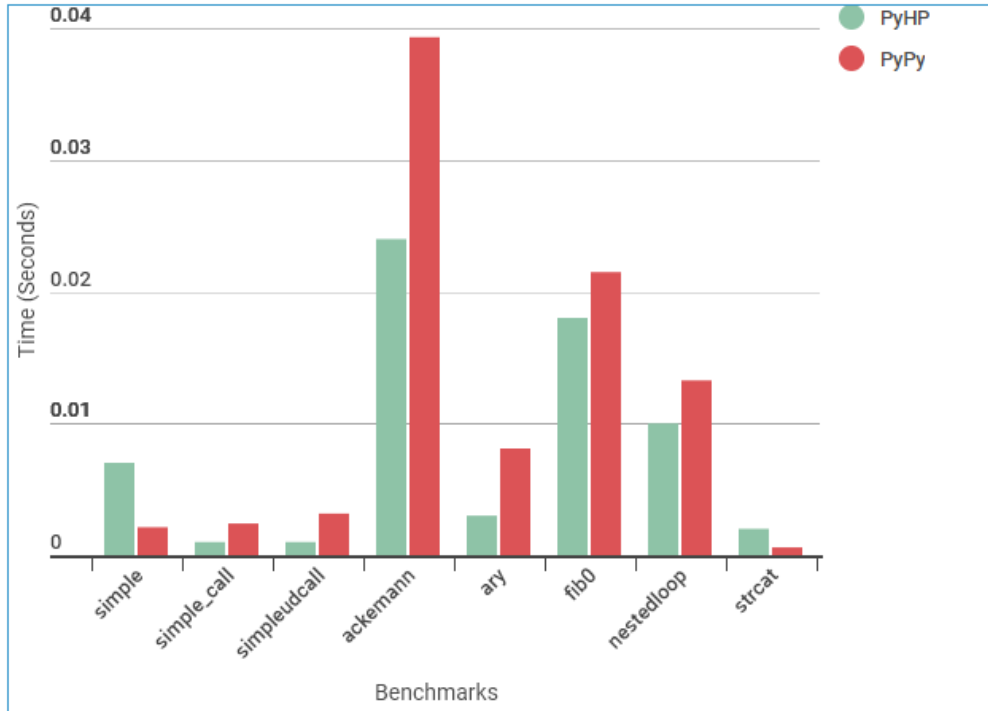


Figure 3.5: **GC Overhead for RPython-based Virtual Machines** – this is normalized to the baseline PyPy and PyHP implementations. In relative terms the overheads here range from 0.2% to about 4%

so this requires more study to understand any GC-specific aspects [108].

3.5 Extending Eclipse-OMR-based Garbage collection

We implemented six other GC policies of the described GC algorithms in Section 2.3.2, using OMR to facilitate fairer comparisons. From the analysis shown in Figure 3.6, we find that, consistent with results by Blackburn et al. [14, 15] and our paper [105], GC frameworks can introduce overhead that is a direct result of calling the framework API as shown in Figure 3.6.

The software engineering benefits are still impressive as shown in Table 3.4, characterized with lower lines of implemented code for the OMR GC and thereby reduced effort and difficulty.

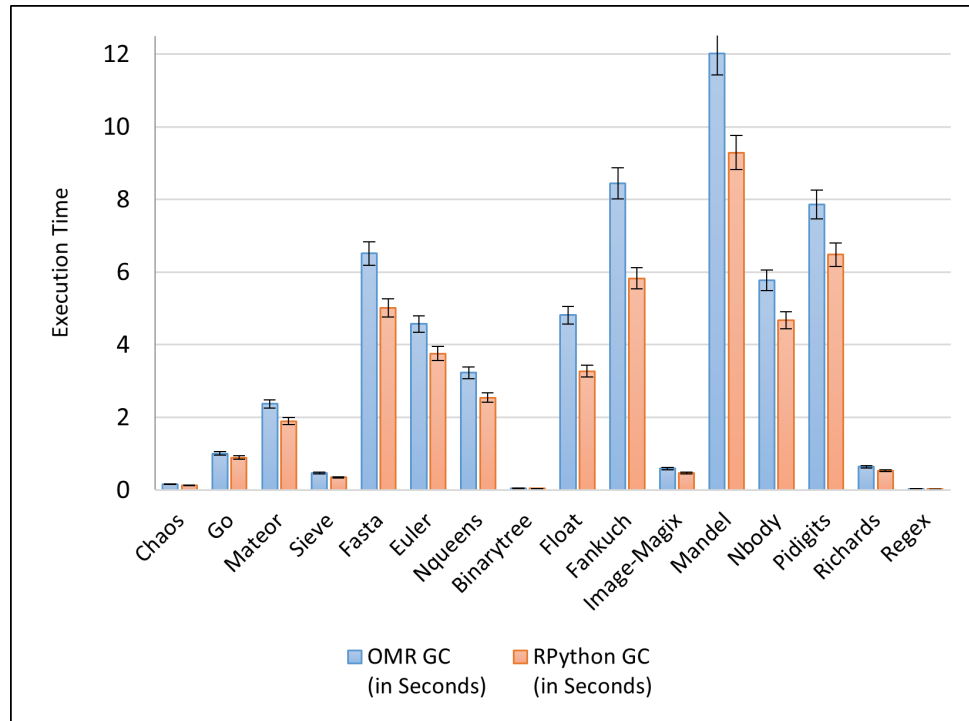


Figure 3.6: **OMR-based GCs Vs. RPython GCs in PyPy** – the total execution time (GC time + Mutator time) as a geometric mean across seven garbage collection algorithms

Generational GCs have more overhead compared to the non-general mark-sweep GC in Section 3.2, because we have extra overhead from write barriers as well, we do not tease apart the barrier time but Figure 3.7 has a breakdown of performance of a semispace GC. Of all the six policies implemented, the five policies are generational so we use a representative semispace policy to highlight some differences. We use larger data inputs to the workloads for them to run longer.

The first insight is that the OMR semispace GC is consistently having more overhead in allocation and mutator time due to the cost of allocation and write barriers which can be optimized if a fast path in OMR is implemented. For GC time, OMR runs faster for most workloads except the AI workload which is random and we are not able to ascertain what exactly causes this difference. The total execution time consistently shows overhead for OMR because it is due to the allocation and barrier costs.

	OMR	RPython
Number of Methods	52	60
Number of Bytes	32,491	34,377
Lines of Code	737	757
Logical Lines of Code	494	517
Standard Lines of Code	525	549
Difficulty	12.87	13.77
Effort	46,052.57	53,002.37

Table 3.4: **OMR-based GCs Vs. RPython GCs in PyPy** – the geometric mean of software engineering metrics across the seven garbage collection algorithms

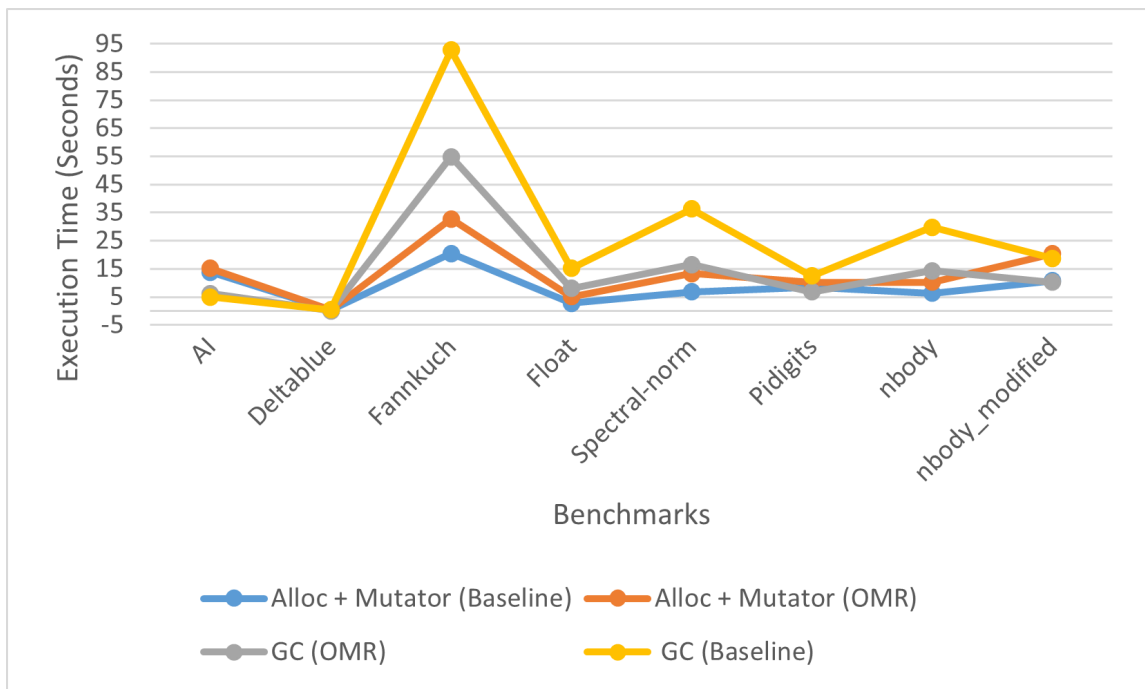


Figure 3.7: **OMR-based GCs Vs. RPython GCs in PyPy** – the allocation, garbage collection and mutator time for the semispace policy

3.6 Related Work

This work focuses on an at-scale evaluation of the use of a garbage collection framework for dynamic languages that use a tracing JIT. The UMass Language Independent GC Toolkit [74] was the first GC toolkit, published 30 years ago. Several other GC frameworks have been developed [14, 112, 52], but most of the evaluation has focused on Java. As a result, many findings and GC improvements have been made

for garbage collection specifically for Java. We also use Eclipse OMR (originally developed as part of a JVM), but instead study dynamic languages like Python and PHP.

The Boehm GC framework (`libgc`) is supported in the RPython framework, but the resulting GC is very slow [36, 20] because it is conservative. `Libgc` was also originally tailored for the C/C++ languages, making its general applicability questionable [20]. Eclipse OMR does not support reference counting, but we still choose to use it because most of the interpreters supported by the RPython toolchain use tracing garbage collection. Tracing garbage collection from this same toolchain has been linked to better performance in PyPy to some extent, but comes with incompatibility for Python extension modules [110, 117].

Chapter 4

The Garbage Collection Cost For Meta-Tracing JIT-based Dynamic Languages

We established in the previous chapter that garbage collection frameworks can have both performance and software engineering benefits for Python and other RPython-based dynamic languages. However, garbage collection in dynamic languages still exhibits surprising behaviour for different applications [111, 47, 130, 79, 69, 125, 77, 32, 98, 78], but this behaviour has not been fully studied.

In this chapter, we study the real-world cost of garbage collection while simulating production environments for dynamic languages that are developed using RPython, and use the Eclipse OMR GC framework. We use Eclipse OMR-based garbage collectors for this study based on previous evidence of applicability, software engineering and performance benefits [105].

4.1 Introduction

Inspired by at-scale garbage collection challenges faced by large companies like Instagram [144, 31], Phusion [111] and Oyster [71] using dynamic languages like Python and Ruby, we use language-independent GC workload characterization to study the cost of several garbage collection algorithms on five real-world Python applications, broadly categorized as web servers, machine learning, and gaming.

For web servers, firstly, we find that garbage collection routines slow down server startup and request processing by a magnitude of about $2\times$. It also leads to fast-growing inflated memory consumption, requiring manual interventions such as disabling GC and tuning the GC threshold for better performance. Data frames are common in popular machine learning frameworks and pressure the GC, contributing to more than 60% of the used memory for most workloads. Also, we find that it is important for developers to use less memory consuming constructs to avoid GC overhead, for instance using ordinary loops for hyper-parameter tuning, instead of grid search modules. We also find that gaming applications still have unacceptable frame drops due to GC pauses even when run with an efficient incremental generational collector.

From this same analysis, secondly, we identify that JIT tracing contributes heap allocations and thereby memory pressure that depends on the trace size. For this, we identify that an effective trace size is application-specific, and that a fixed trace limit can lead to poor performance for some applications. We also find that increasing the trace limit generally improves performance, however, a long trace can lead to memory and GC overhead, as well as JIT optimizations taking longer. We therefore propose dynamic trace sizing through trace-based profiling [67] to alleviate this problem. We evaluate this profiling technique on both PyPy and Pycket, reporting performance gain across standard benchmarks.

Therefore, the main contributions in this chapter can be summarized as follows: 1) We

implement language-independent GC event characterization tooling as part of Eclipse OMR; 2) We use this tooling to quantify the garbage collection cost for real-world Python workloads using production garbage collectors; and finally 3) Motivated by the sources of GC overhead in the study, we propose a GC-aware dynamic JIT trace sizing technique.

The rest of this chapter is organized as follows: Sections 4.2, 4.3, 4.4 and 4.5 discuss the GC cost of several Python applications and a technique for dynamic trace limit sizing; Section 4.6 contains the related work.

4.2 Language-independent GC Workload Characterization

The work in Chapter 3 enables this further experimentation. To help us quantify the GC cost, we implement a language independent workload characterization technique specific to garbage collection as part of the Eclipse OMR GC component. We then call this tooling in the RPython framework, to isolate the various GC events. The tool consists of annotations to signal the start and end of an event.

The core of the tool are `gc_begin()` and `gc_end()` annotations, which are given a string of the category of event being tracked. Using these annotations, we return time stamps, or memory stamps that can then be processed in a log tool or printed to track the time taken or memory used. We support tracking the following events for purposes of this work:

1. Object scanning
2. Allocation
3. Nursery collections
4. Memory usage

5. Major collections

We do not isolate the cost of barriers, but for any barrier operations that happen as part of these GC events, the barrier cost is accounted for in the total cost of the event. For any barrier operations outside of these GC events, it is widely assumed that barriers impose significant overhead on the mutator, so it is possible that we do not fully report the full garbage collection barrier cost [17] outside the above GC events of interest to our analysis.

PyPy has debug tools coupled with logging that emit JIT and some GC information but they are limited to the RPython implementation and cannot be used for other languages. Also, they can only track minor and major collections. OMR has some event hooks but they do not provide this level of abstraction to give us the required level of detail. This tool is sufficient for most of the evaluation and for other metrics like LLC/TLB/LL1 cache miss rate, page faults, we use the Linux perf tool and other Linux tools like `/proc/PID/smap` for shared memory.

4.3 The Garbage Collection Cost for Python Applications

Blackburn and other researchers have studied the garbage collection cost for (mostly) Java using generic benchmarks like DeCapo [14, 79], and other semi-realistic workloads. This has given us insight into general GC costs for applications but these studies lack the context for production applications like long running servers, machine learning with big data sets and games with high latency-sensitivity.

Existing work on dynamic languages has studied the impact of garbage collection in dynamic languages like Python [79, 130] mostly for the default CPython and PyPy garbage collection algorithms. Our work instead investigates the GC cost of several garbage collection algorithms on different real-world Python applications. To the

best of our knowledge, this is the first study that holistically quantifies the garbage collection overhead, while realistically emulating production environments for Python web server, machine learning and gaming applications.

This study gives insights into the GC-related performance overhead for different workloads, which will lead to ideas on garbage collection policies that can better address these problems. The applications are chosen because they are popular, and are also widely recognized as challenging with respect to GC in the Python community. For example, garbage collection was found to increase the memory footprint of server applications in both Ruby and Python, which led to companies like Instagram and Phusion disabling the Python and Ruby garbage collector respectively. As a result, these companies also patched the CPython and MRI (Matz’s Ruby Interpreter) garbage collectors to support copy-on-write to remedy the challenges [31, 111, 71]. Garbage collection in Python machine learning libraries like Pandas and H2O-Python also causes inflated footprints and poor performance [120, 61, 98, 2]. Latency-critical Python gaming applications like Pygame have struggled to completely adopt PyPy’s tracing garbage collection, citing the need for low pause times [115]. The cache and memory hierarchy is impacted for dynamic languages [78, 79], but also meta-tracing JITs seem to use more memory with garbage collection compared to ordinary tracing JITs [77]. The dissertation in this chapter provides a detailed study, not given in these mentioned sources, to verify these claims.

4.3.1 Real-world Workloads

Five workloads categorized as web servers (two), machine learning (two) and gaming (one) are implemented, emulating realistic production environments of the applications. Informed by deployments in companies like Instagram, Oyster and Phusion, we emulate the scenarios but do not run the applications to the full scale of very huge databases for example. We however use data sets and databases that are between

5GB to 10GB in size. We believe this is realistic enough to run on research lab machines in our budget. We discuss each of the categories below.

Servers: We implement a *Web.py* and *Django* based server, each having an architecture as shown in Figure 4.1. The servers use an Nginx load balancer that receives requests from a process that uses the *requests* library to send requests to both the Django and Web.py servers. The Django server uses *Gunicorn* that forks three worker nodes and a master node that serves requests. The Web.py architecture uses three isolated server processes that also serve requests from the load balancer. We acquired a 10GB database from the stack exchange open access data repository [35]. On startup about 6GB is cached in Python dictionaries. This is a common optimization used by most server applications to avoid frequent database hits. Each server or worker node receives 445 requests, which are also heavy queries from the cached data.

Machine Learning: We exercise the *Pandas* and *H2O-python* machine learning libraries. We train models on a 3GB data file. The resulting data frames can be as big as 12GB. We also evaluate specific overhead when using structures like ordinary loops vs. grid search.

Gaming: We modify the Pygame *testsprites* benchmark from the Pygame open source repository on GitHub [33].

4.3.2 Experiment Setup

By default unless otherwise stated, the analysis uses an Intel(R) Xeon(R) Gold 6248 CPU @ 2.50 GHz machine, running 64-bit Debian 11.3, using not more than 40 cores. Due to graphical requirements for the PyGame benchmark, we use the following machine specifications for it, MacBook Air, MacBookAir9, Quad-Core Intel Core i7, 1.2 GHz, 4 cores, 16 GB RAM, macOS Catalina 10.15.7 and 6 MB LLC.

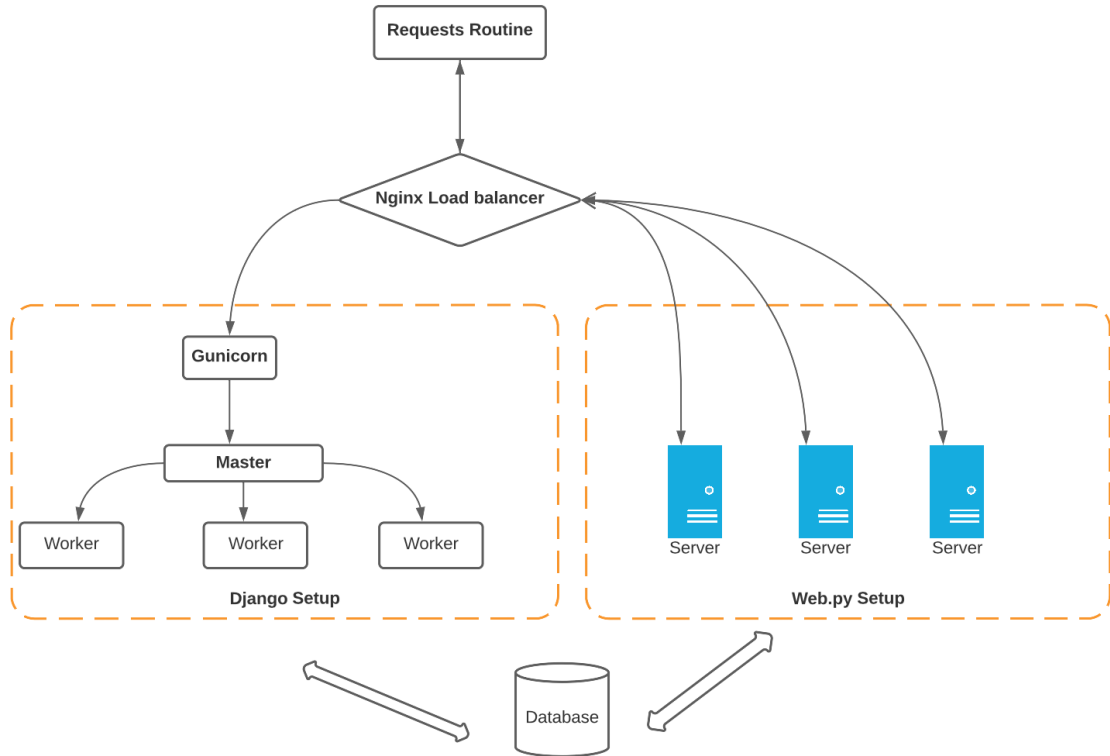


Figure 4.1: Architecture of the Server Benchmarks

4.3.3 Application-specific Analysis

There are five key areas we consider when evaluating garbage collectors, namely *heap size*, *GC policy*, *cache miss rate*, *effectiveness of the nursery* and *JIT tracing*. The first four are more common in existing studies, but the latter does not have as many studies, much less a GC-specific overhead investigation.

We find that garbage collection algorithms are application specific but for the memory hungry workloads we study here, the Minimark and Incminimark, which are generational and incremental generational GCs respectively, offer acceptable performance to facilitate frequent experimentation. The rest are relatively slow and consume more memory. This is partly due to design limitations in how the other policies were implemented. However, the whole-heap collectors like Mark-and-Sweep and Mark-compact use less memory compared to the region-based garbage collectors, semispace and generational as empirically observed in the PyPy micro benchmarks.

Due to scope limitations the experimental results are omitted in the dissertation. Across all benchmarks, increasing the heap size generally reduces GC overhead because collection is done less frequently, which in turn leads to better performance at the expense of using more memory. In the rest of the analysis, we discuss each category of benchmarks separately in relation to performance, memory consumption, cache miss rate and nursery sizing. For the rest of the discussion in this section, we use the default incminimark GC because its implementation is more complete and JIT integration is robust.

Servers: Due to the high volumes of requests processed by servers and data caching, especially on startup, for both most production Django and Web.py use cases, garbage collection impacts the performance and memory usage mainly for server startup and request processing.

In terms of performance due to the high volume of objects cached during server startup, garbage collection runs more frequently. Minor collection does not have significant overhead but for even the incremental incminimark GC, major collection is the significant contributor to the GC cost. From our experiments, major collection runs about three times by mean at startup for each of the three Web.py servers, and Django worker processes. This is shown from the very high spikes during server startup as shown in Figure 4.2. The spikes continue in smaller magnitudes as the server processes more requests due to major GC pauses.

For similar reasons, the time of execution of requests increases over time, costing between 10% and 30% of the mean execution time when GC is used. This is also shown from the high spikes during program execution. This lowers requests-per-second, worst case by 2×. In fact because major collection is Stop-the-World (STW), we noticed about 1.5% of the 445 requests sent to the server and worker processes timed out. This is a relatively small percentage but for large deployments with millions of requests, the impact can be significant. This GC cost depending on

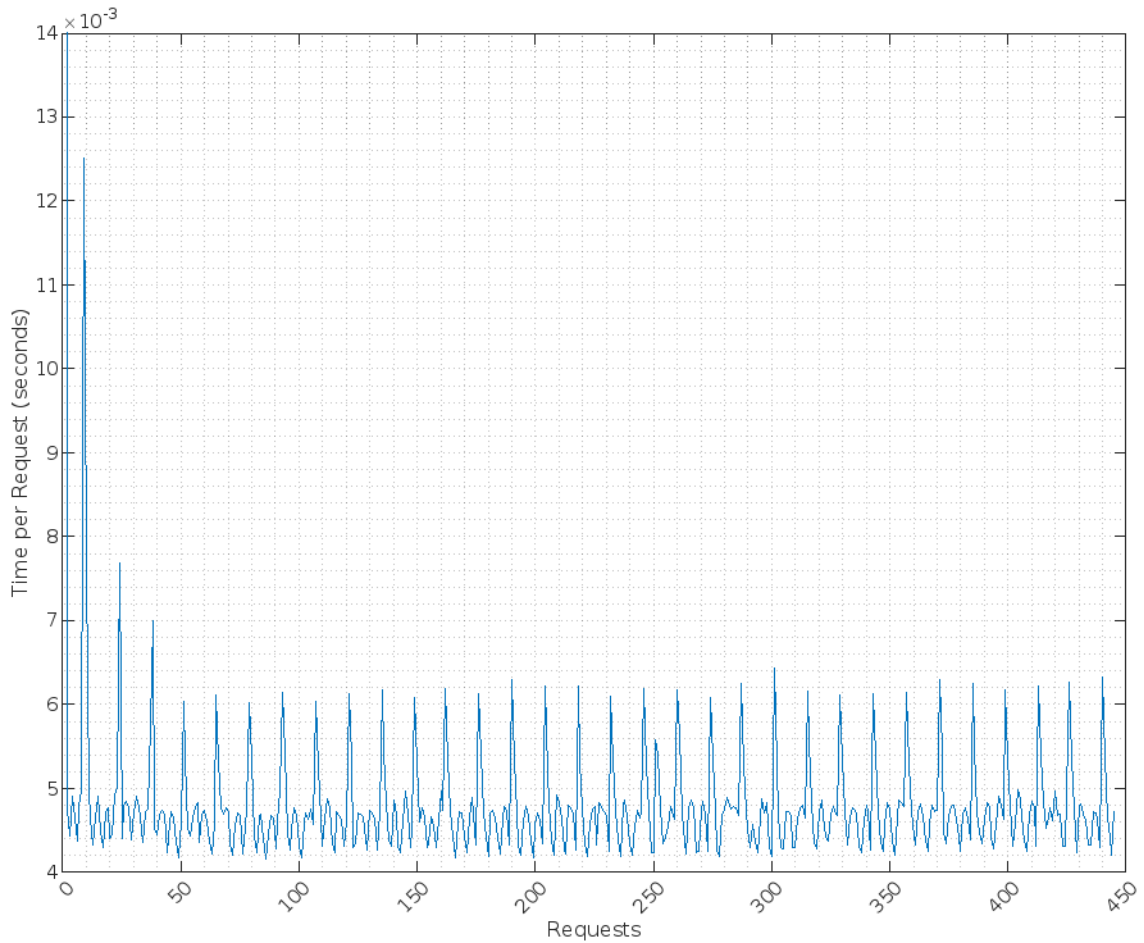


Figure 4.2: The Mean Time Taken to Process 445 Requests on the Django and Webpy Server Processes

settings/parameters, requires that we give users some control for GC tuning but also for manually starting and disabling collection in GC design, however, disabling collection dramatically increases memory consumption. For example, during server startup, GC can be disabled to reduce startup time because most collections are pointless at startup. Also to avoid GC-related timeouts, when GC is about to start, determined by the threshold, a given server process can be momentarily stopped to perform collection, while requests are routed to other nodes, since load balancers automatically direct traffic only to listening processes. If this is intended to avoid latency spikes, then there is need to ensure that a request is complete before pausing a server.

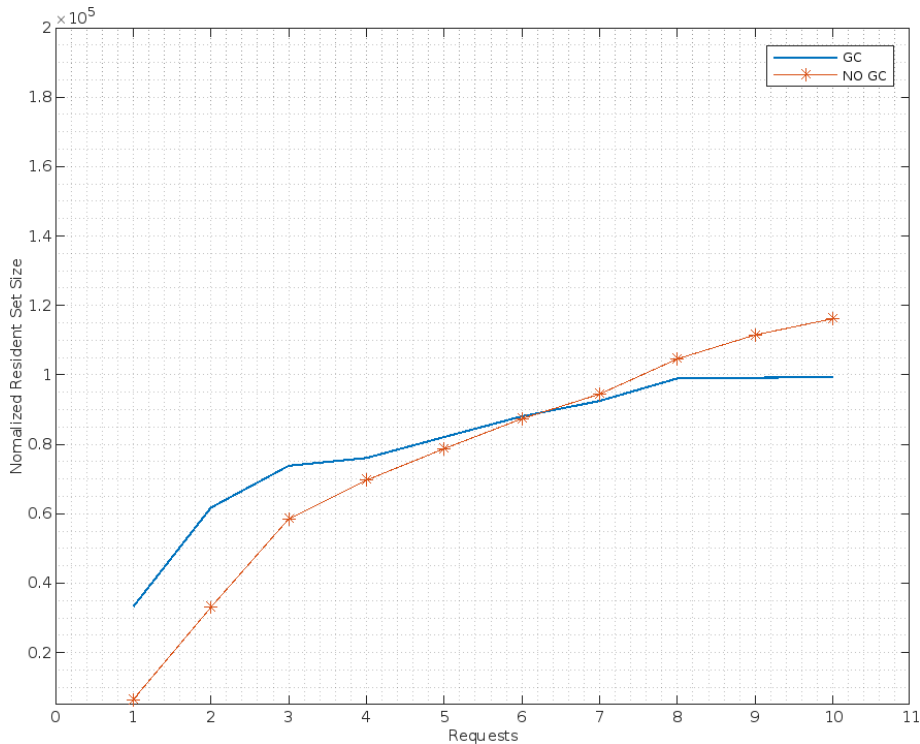


Figure 4.3: The Mean Memory Used for Both Django and Web.py

From Figure 4.3, initially memory consumption is 10% higher due to garbage collection and increases dramatically. From results we do not show but were generated and observed as part of the analysis in Figure 4.6, shared memory reduces by about 40%, and the cache miss rate is about 10% higher and performance can be 10% better when GC is disabled. However disabling GC is not a solution, because with time as shown in Figure 4.3, without garbage collection, memory usage increases dramatically leading to a higher resident set size than when GC is used. The main insight here is that if you prioritize the long run effects, then garbage collection has more benefits, but at startup for example, garbage collection introduces more overhead. In Linux environments, garbage collection causes more page faults as seen from using the perf tool from our experiments, and page faults are usually related to unnecessary copy-on-write operations [144, 71, 31]. For GC implementation, there is need to design for the effects of copy-on-write to avoid undesired memory overhead.

	Grid Search	Ordinary Loops
Execution Time (Minutes)	78.90	46.27
Working Set Size (Bytes)	1,508,164	1,796,588

Table 4.1: Hyper-parameter Tuning with Grid Search Vs. Looping

Machine Learning: For both the Pandas and H2O-Python libraries, because of the large data sets of machine learning, most of the GC overhead is as a result of loading and processing data frames. For example the workloads read a data set from a *csv* file that is 3GB in size on disk, but the resulting data frame is larger, about 12GB. This is without counting any intermediate data frames that may be a result of operations, like slicing. What this means is that to effectively manage memory, the GC threshold should be smaller, but this leads to more frequent collections hence GC overhead.

Also, certain constructs in machine learning libraries use more memory and hence cause GC overhead. These constructs can range from custom data types, to particular features supported by these libraries. As an example we experimented with performing hyper-parameter tuning using *Scikitlearn* and *GridSearch* for Pandas and H2O-python respectively, and comparing it to using ordinary loops. The latter is efficient performance-wise but uses more memory than the former, as shown in Table 4.1, by almost 50%.

Like the server applications, increasing the nursery size does not reduce the GC overhead because the highest overhead is due to major generation collection, therefore both execution time and the working set size increases but later reduces when the nursery size is 32MB. The LLC, TLB and L1 miss rate reduces relatively, with not much improvement at higher nursery sizes as shown in Figure 4.6.

Gaming: Games are latency critical applications, requiring sessions to be relatively fast. The performance of a game is determined by the amount of time it takes to

process a frame, translated to a metric called frames per second (FPS). The mean game should run at an acceptable FPS rate that is between 30–60. To know the real impact of the GC on the PyGame library, we modified the *sprites* game, implemented as part of the PyGame examples, making it complex enough to stress the garbage collectors.

Figures 4.4 and 4.5 show the time taken to process 1200 frames when major collection is enabled and disabled respectively. We find that even for the efficient `incminimark` GC, the application drops frames to as low as 20 FPS when GC is enabled. We did not explore any custom configurations of the `incminimark` GC like `PYPY_GC_INCREMENT_STEP` which can affect these frame drops. The dropping of frames is shown with the spikes in the graphs. The graph where major GC is enabled shows more spikes compared to the one where major GC is disabled. These spikes and drop of frames are due to the STW activities required to reclaim memory. Increasing the threshold does not achieve any benefits, rather we noticed a change only when major collection is disabled.

We do not concretely know the cause of large spikes when major GC is turned off in this experiment, and in the context of our analysis we can conclude that they have nothing to do with garbage collection, since we do not find their occurrence mapping to minor collections either.

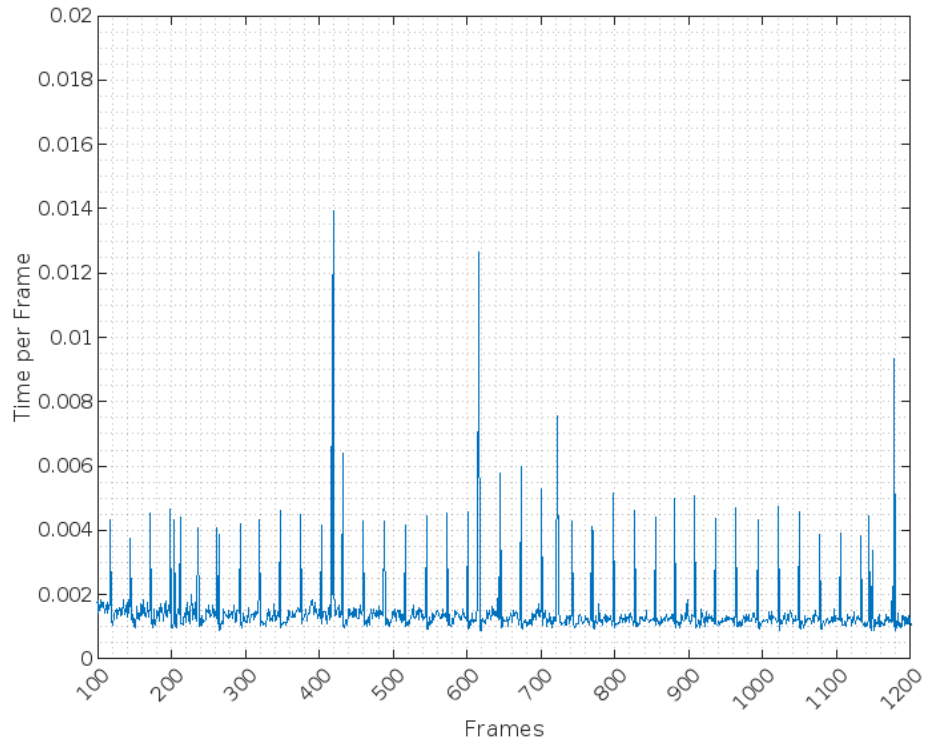


Figure 4.4: Time Per Frame With Major GC

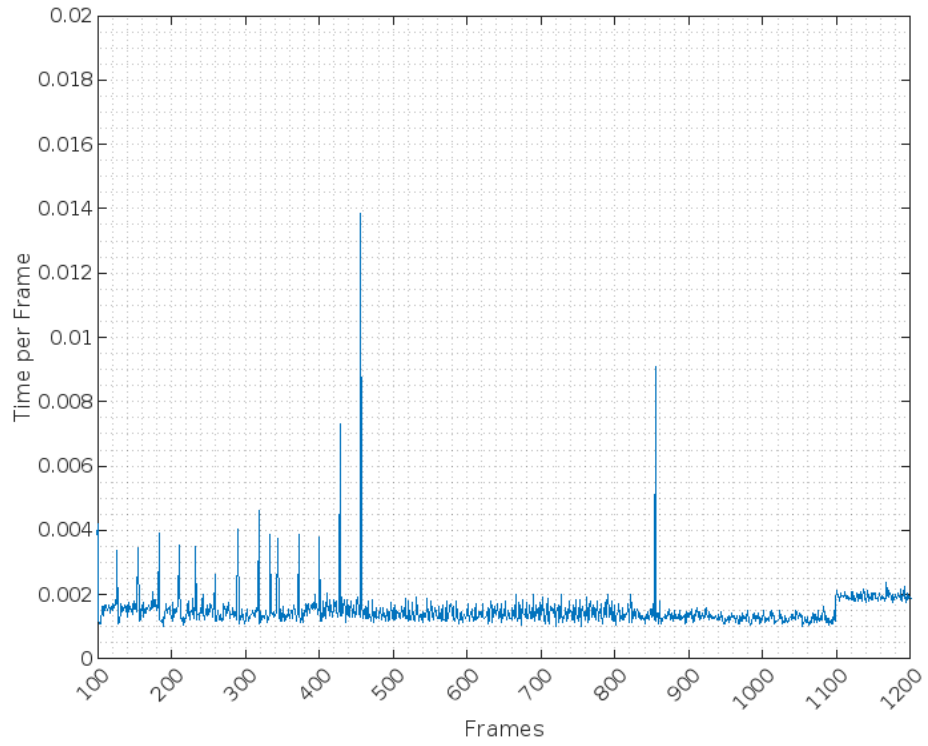


Figure 4.5: Time Per frame Without Major GC

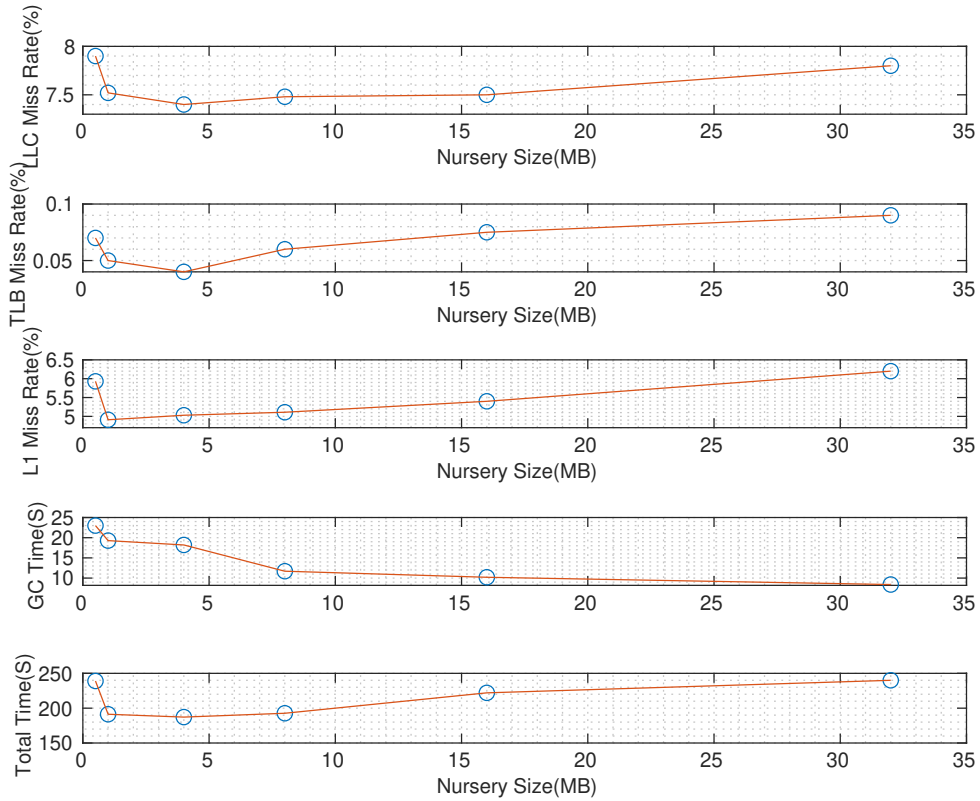


Figure 4.6: The Relationship Between the Cache and Execution Time at Varying Nursery Sizes Across all Workloads

4.3.4 Nursery Sizing and Cache Performance

We use an 8 LLC cache for this experiment. Across all benchmarks, we observe that increasing the nursery size degrades performance in general due to poor cache performance as shown in Figure 4.6. A large nursery size reduces the GC overhead which corresponds to lower GC times but when the nursery does not fit in the cache, the total execution time increases due to poor performance of the cache. In the Figure 4.6 example, the best nursery size for most workloads is a size that is half the cache size, but as pointed out by Ismail et al. [78], some workloads can perform better with a nursery size that is less/higher than half the cache size.

Increasing the tenure region threshold can reduce the frequency of major collections and improve performance in general.

We used a simpler version of the *sprites* game to test nursery sizing. Like other benchmarks, increasing the nursery size from the default 0.5 worsens FPS as shown in Table 4.2. The FPS starts at over 500 FPS when the nursery size is 0.5 MB, and drops to as low as 26 at 4 MB nursery size. It then raises between 34 - 46 at nursery sizes 16 and 32 respectively, making smaller nurseries generally favourable for most gaming applications, of course with some exceptions depending on the game workload.

Nursery Size (MB)	FPS
0.5	569.16
4	26.99
16	34.56
32	46.00

Table 4.2: Nursery Sizing and FPS

4.4 JIT Tracing and Garbage Collection

Other than the general application-specific cost already discussed in Section 4.3, we also find due to the collection of a sequence of instructions during trace formation for tracing JITs, the contents of a trace make allocations that contribute to garbage collection pressure. Existing work [77, 22] discusses this with no clear quantification of the actual cost. In this section we discuss our findings on the trade-off between JIT tracing and garbage collection.

4.4.1 JIT Tracing Overview

Tracing JIT compilers are usually used in a hybrid mode consisting of both an interpreter and a compiler. The interpreter executes a program, and conducts profiling to identify the frequently executed paths, also known as *hot paths* of the program. These hot paths are then optimized by the JIT compiler, where every

```

1 def ftn(n, m):
2     res = 1
3     i = 0
4     while i < n:
5         res *= m
6         i += 1
7     return res > 100

```

Listing 4.1: A Hypothetical RPython Program

```

1 i3 = int_lt(i1, i2)
2 guard_true(i3)
3 # inside the loop
4 i4 = int_mul(p0, i2) # res *= m
5 i5 = int_add(p1, 1) # i += 1
6 i6 = int_gt(i4, 100)
7 guard_true(i6)
8 jump(p0, i25, i2)

```

Listing 4.2: A Simplified Trace of the Program in Listing 4.1

instruction in the execution path is recorded to form a *trace*, then the trace is optimized and converted to machine code.

A *trace* in tracing JIT compilers is therefore a sequence of instructions, created at runtime corresponding to the actual execution behavior of a program [67]. Traditional tracing JITs form traces for the execution of application code for a given program, which correspond to the byte codes executed of the target program. As an example, Listings 4.1 and 4.2 show a hypothetical program and its simplified trace.

The simplified trace in Listing 4.2 represents a single iteration of the loop on line 4 of Listing 4.1 in the function `ftn`. The registers p_0 and p_1 are for the variables `res` and `i` in the RPython program. The guards in the trace have any needed information should a return to the interpreter be required in case of a guard failure.

Figure 4.7 shows the different phases taken by a typical tracing JIT. Interpretation of the target program is the first phase, where general execution and profiling is performed to detect any hot loops. The identified loops are traced by the interpreter, forming a sequence of all the operations encountered during execution of the hot loop, which forms the trace. During compilation the hot loops are compiled by the JIT

before execution with optional calls to functions compiled ahead-of-time (AOT). JIT compilation makes callbacks to VM libraries and certain VM components like the garbage collector and thereby experiences a garbage collection phase before the black hole interpreter performs any required deoptimization.

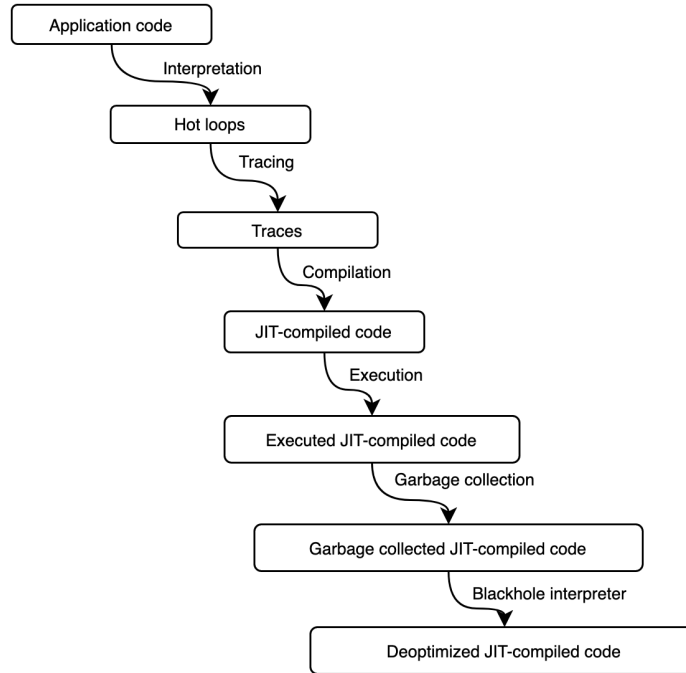


Figure 4.7: Phases of a Tracing JIT

A recent trend in tracing JITs is to trace the execution of the interpreter as it executes the application code, a technique known as *meta tracing*. As opposed to traditional tracing, meta-tracing is costly for tracing, garbage collection and even execution of the compiled code because much as traditional tracing JITs can potentially also make calls to the JIT framework from within the JIT-compiled code, there is the potential for many more calls in a meta-tracing JIT and for PyPy, the implementation we experiment on, such calls are particularly easy to use with the RPython language. This is especially true where the tracing JIT is written in a garbage collected language since most JIT implementations treat the GC and JIT as separate entities, with the GC implemented in C/C++.

4.4.2 The Garbage Collection Cost of Tracing

Each phase described in Figure 4.7 introduces overhead and impacts each application differently. Garbage collection pressure is more prominent for memory-intensive applications. Most of the GC overhead is before JIT compilation, thanks to escape analysis that removes some allocations but the compiled code also pressures the GC, which makes GC optimization necessary.

This risk of performance overhead is more likely when a trace is extremely long because not only do optimizations take a long time but this is also more work for the garbage collectors. This is because the volume of dynamic instructions¹ to be handled by garbage collection increases and directly translates to exorbitant GC execution times.

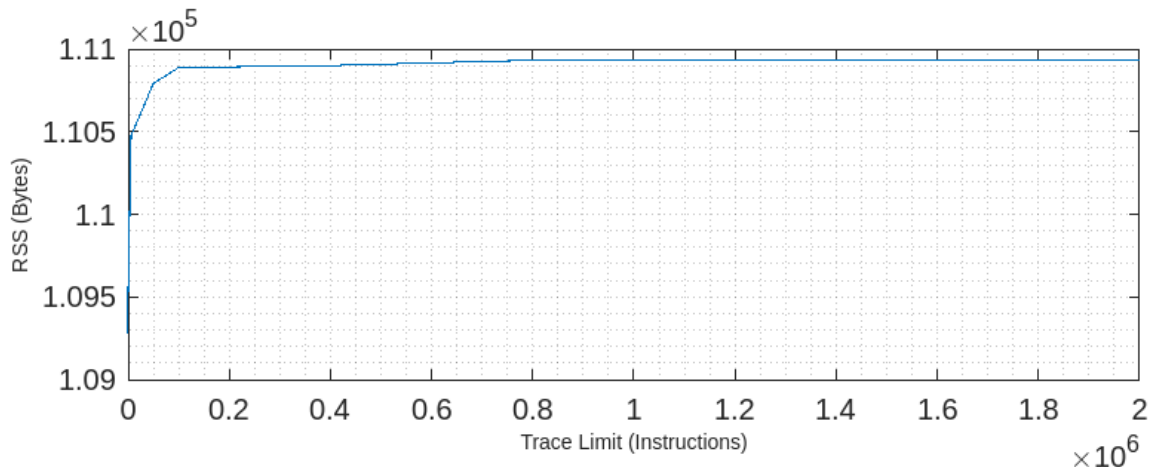


Figure 4.8: **AI benchmark** — The cost of changing the trace limit on resident set size

Tracing JIT compilers usually have a `tracelimit`, which controls how long, in terms of number of instructions, a trace can be; for example, the trace limit for the PyPy Python implementation is 6000 instructions. Figures 4.8 and 4.9 show how the resident set size (RSS) and execution, minor GC, and major GC time respectively vary for various trace limits for the *basic input/output* and *AI* benchmarks, from the

¹For a specific program execution

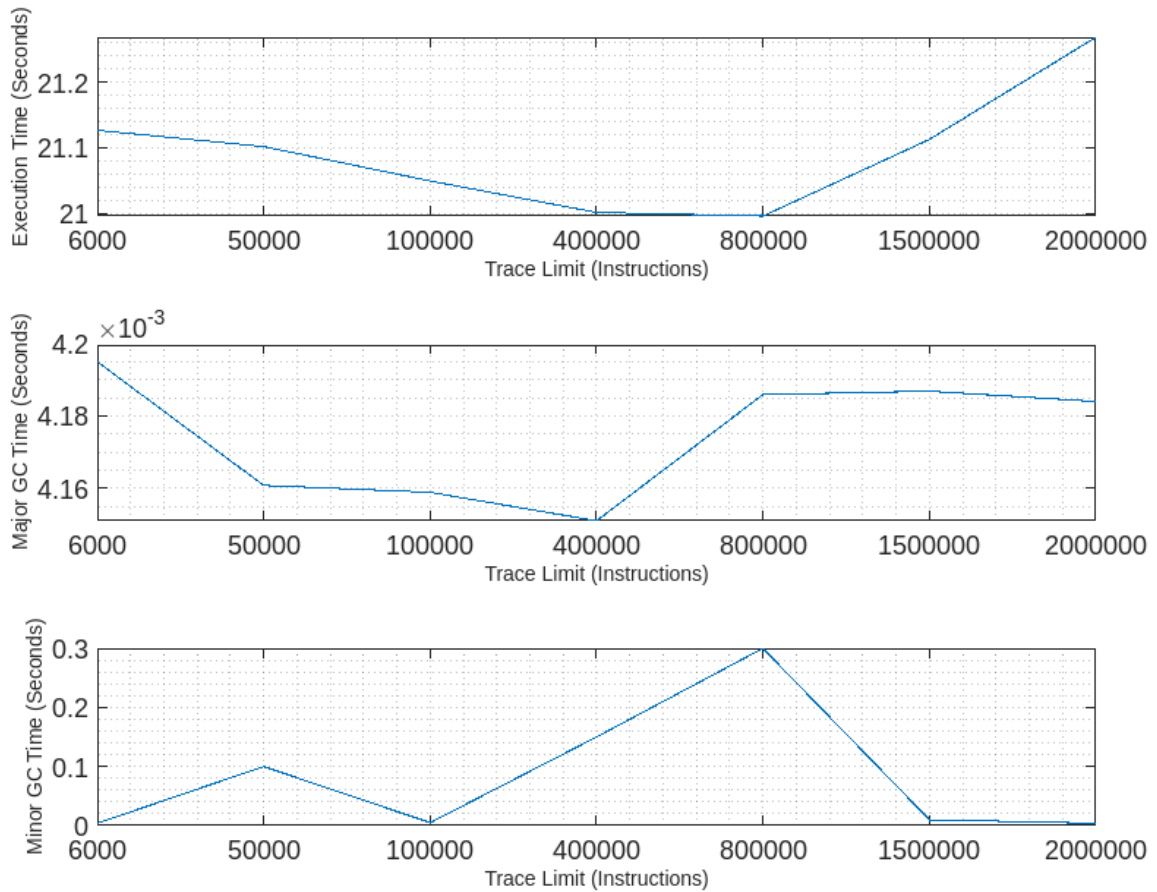


Figure 4.9: **IObasic benchmark** — The cost of changing the trace limit on execution, minor GC and major GC time in seconds

PyPerformance standard Python benchmark suite, using PyPy’s RPython JIT.

Memory-wise, our observation is that as shown in Figure 4.8, there is an increase in RSS with an increase in trace limit. It thereafter remains constant when we reach the maximum trace size of a program. We find that there are performance benefits in generally having long traces because it allows more room for the JIT to optimize the program. As shown in Figure 4.9, increasing the trace limit corresponds to improved execution time up to a point.

As pointed out earlier, the RPython JIT has a trace limit, where if reached, trace formation is aborted but it is not always effective; effective here means the best or optimal trace limit. Based on our empirical analysis, we therefore make the following hypotheses:

1. The effective trace limit is application specific.
2. Increasing the trace limit improves performance up to an application-specific point, after which memory and garbage collection pressure degrade performance.

The first hypothesis is proved from the results of varying trace limits and identifying the best trace limit for each of the applications we benchmarked. Figure 4.10 shows the best limit sizes, the dotted line shows the default trace limit for PyPy. From these results in Figure 4.10, the best trace limits for some applications is not the default 6000. For example the best trace limit (that results in the smallest execution time) for our modified AI benchmark is 400,000, after trace limit 400,000, the execution time increases. The fixed trace limit costs this benchmark about 2% overhead. At the worst trace limit, increasing the nursery size to limit the cost of major collections does not help due to the same cache impact shown in Figure 4.6, performance instead worsens. The same is true for the rest of the benchmarks in Figure 4.10.

We also observe that optimizations enabled by longer traces might not always be a win. For some applications, higher trace limits, above the default 6000, give better performance. Beyond some trace size, we also find that performance starts degrading due to garbage collection pressure as shown in Figure 4.9. The execution time reduces to some degree due to JIT optimizations but as seen, it starts increasing after trace limit 800,000, which is the best limit for this IO benchmark. The increase in execution time afterwards is related to an increase in either major and/or minor collection, which proves the second hypothesis.

These two hypotheses in the previous chapter motivate the idea of dynamically managing the size of the trace instead. The traces are compiled and an optimizer can apply profiling information during compilation, and recently historical data can also be used in the process of trace formation [22].

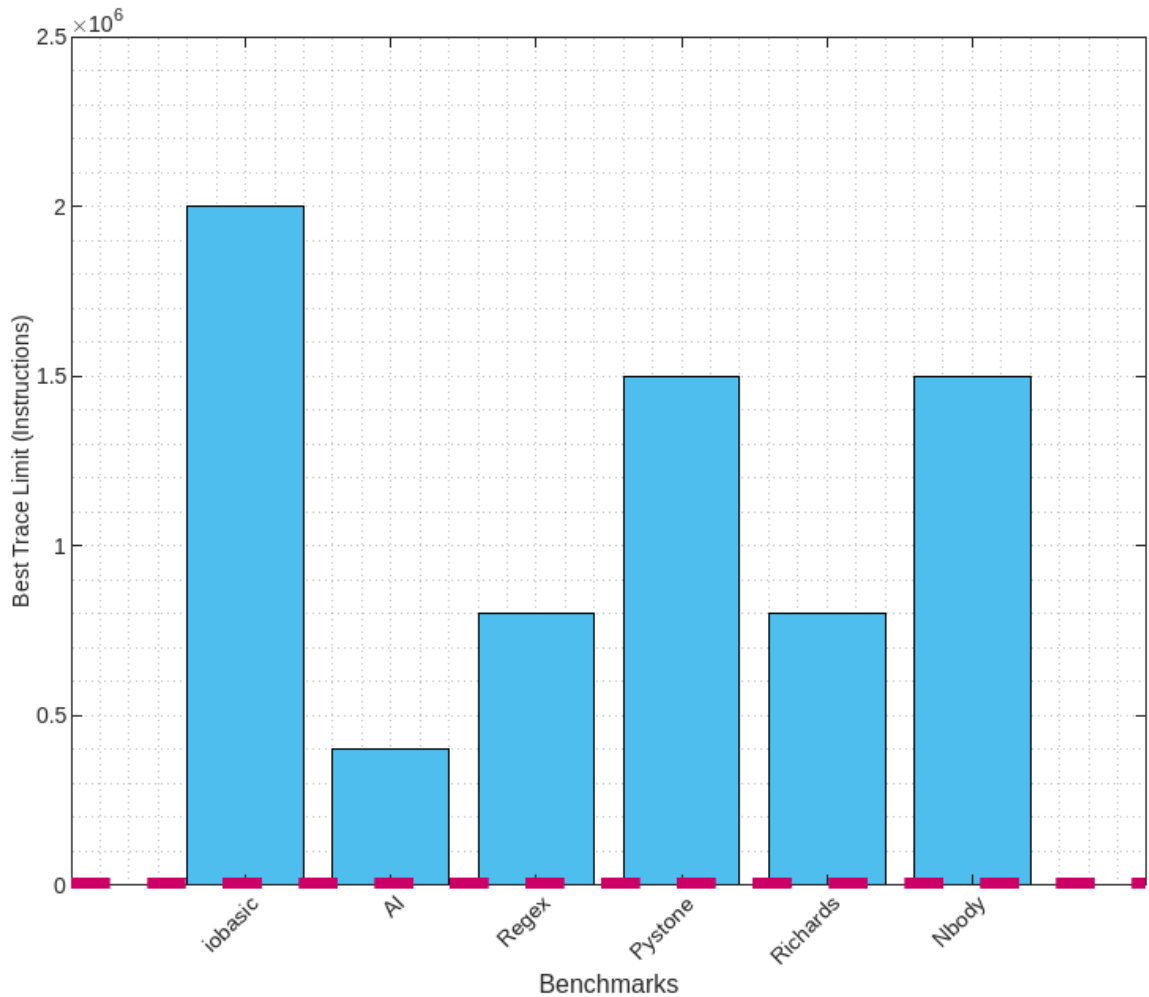


Figure 4.10: **The Effective Trace Limit is Application Specific** — blue bars show the best trace size for each of the benchmarks which is the effective trace limit while the red dotted line is the default RPython trace limit

4.5 DTS: Optimal JIT Trace Sizing for Virtual Machines

We propose a technique, Dynamic Trace Sizing (DTS), that utilizes profiling information during formation of a trace. During execution tracing, we record any useful information regarding program state, this information is used to determine when to abort trace formation in a way that the GC overhead is also minimized. This GC-aware dynamic trace sizing technique can be summarized as:

1. A new trace is not compiled immediately, and while it is in this state, frequency of access to the basic blocks in the trace is recorded, including any related information on execution time.
2. From this information, we identify hot exits of the trace, which is the effective trace size estimation phase.
3. We then estimate the total execution time for a program at this trace size.
4. The estimated total execution time at this trace size can be used to decide to either continue trace formation, or trigger a trace abort. The total execution time comprises of GC time and mutator time.
5. By triggering the trace abort we are able to dynamically size a trace at runtime.

We therefore propose efficiently sizing the trace using trace-based profiling, to set a trace size where JIT optimization opportunities are high and the GC overhead is low. The goal of the technique described in this section is to form efficient traces before compilation for reduced code sizes, compilation time, and startup time while maintaining steady-state performance.

A trace-based optimizer compiles traces at a point where static program information is not available, therefore profiling of traces makes it possible to take advantage of runtime information during compilation. The dynamic trace sizing technique described next profiles both the trace and any other runtime information to dynamically improve trace formation.

4.5.1 Overview

This technique is a GC-aware optimization for trace-based optimizers, where we efficiently generate traces that have high guarantees for low garbage collection cost

and pressure. To achieve this, we address the following intermediate technical problems:

1. An understanding of how the optimal trace size changes.
2. A run-time methodology to obtain individual program characteristics that indicate the program phase.
3. An efficient algorithm to determine an optimal trace size.

4.5.2 Trace Estimation

To address the first challenge of understanding the optimal trace size, Figure 4.11 shows the high level design of DTS where the trace is first optimized during interpretation. That is to say, when the interpreter enters the *tracing mode*, an original trace with all the basic blocks is created. However, just before this trace is committed for compilation, we profile the trace, recording profiling information about the trace and its basic blocks.

We record the frequency, f_{tr} , of entry to the trace, as well as the frequency, f_{bb} , of exit for each basic block in the trace. We determine the execution frequencies of the trace and its basic blocks from the entry frequency and exit frequency of the trace and each basic block. With this information about the trace and additional phase information, we are able to form an improved *final trace* by estimating the following:

- The *effective trace size* S_{eff} , based on the hottest exit.
- The maximum trace size, S_{max} .
- The GC time, T_{GC} .
- The mutator time, T_{mut} .

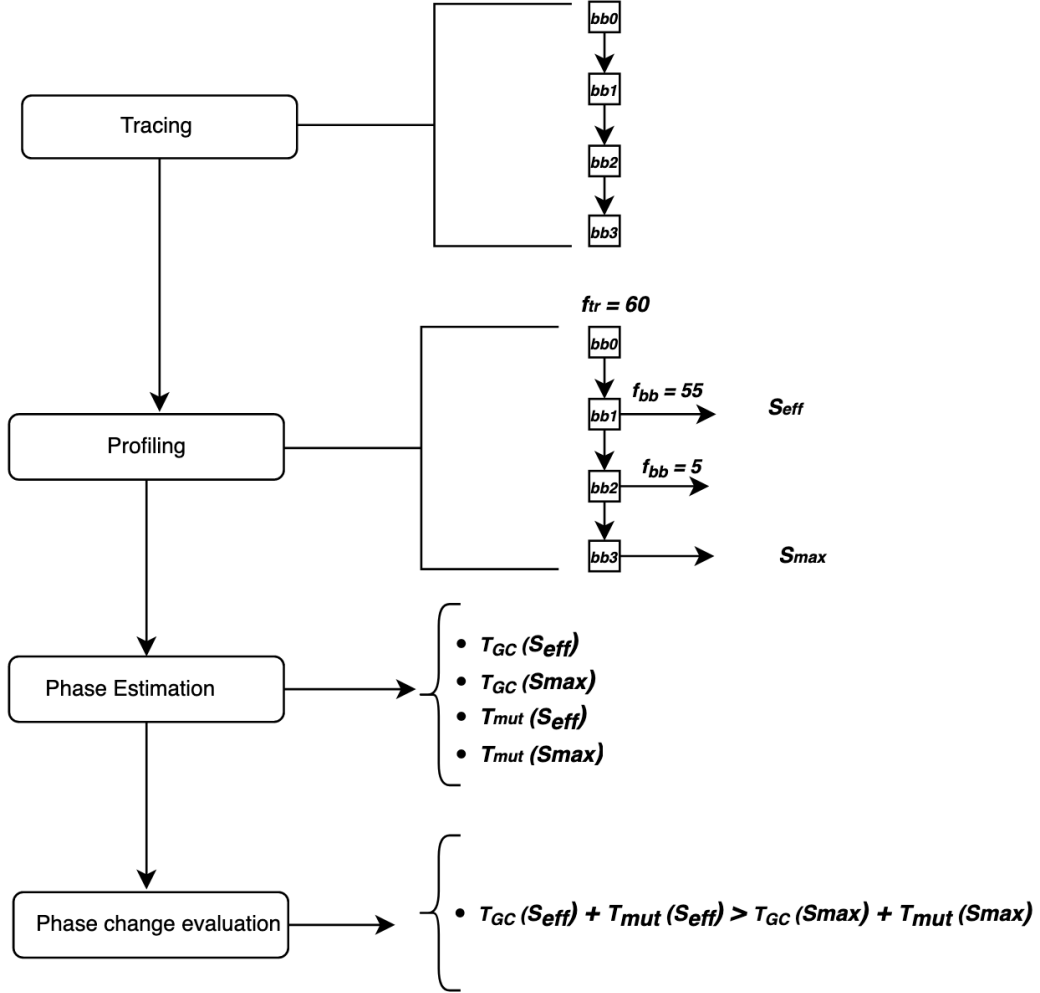


Figure 4.11: The GC-aware Dynamic Trace Sizing technique

During this profiling the trace enters an intermediate state, and does not leave this phase until its entry count exceeds a predetermined threshold, th . From empirical studies guided by maximum performance on the standard PyPy benchmarks², we set th to 80 for the current implementation. Before a trace reaches this entry threshold, we also assign the exit frequency numbers for each basic block. When f_{bb} for a basic block is below a predefined fraction, λ , of f_{tr} , which is also a predefined threshold in the algorithm, then that basic block is a cold exit:

$$f_{bb}/f_{tr} < \lambda$$

²These are similar benchmarks to those used in the evaluation

Therefore, in basic terms, at this certain point in time, T_{clock} , we have the full trace generated so far (cold), and an idea of the basic block that is likely to determine truncation at a hot exit (hot). The size of the IR of the original trace is the size of the total trace generated so far, $S_{max}(T_{clock})$ at a point in time T_{clock} . The cold exit block determines a subset of the full trace, the effective trace, which is the trace for basic blocks whose entry frequencies exceed the predetermined threshold.

This effective trace has a size, the effective trace size, $S_{eff}(T_{clock})$, also at that same point in time, T_{clock} . Therefore the basic block with the highest exit frequency determines S_{eff} , while the last block in the original trace determines, S_{max} because we compute the trace sizes at those basic blocks.

The other estimation steps use the trace sizes S_{eff} and S_{max} to estimate the GC and mutator time. The total execution overhead calculated from estimating both the GC and mutator time:

$$T(S_N) = T_{mj}(S_N) + T_{mn}(S_N) + T_{mut}(S_N) \quad (4.1)$$

$$T_{GC}(S_N) = T_{mj}(S_N) + T_{mn}(S_N) \quad (4.2)$$

Where $T(S_N)$ is the total execution time at a given trace size S_N , T_{mj} is the major GC time, T_{mn} is the minor GC time and finally T_{mut} is the mutator time. The sum of both the major and minor GC times comprises the total GC overhead T_{GC} . From empirical studies also supported by the generational hypothesis, we do not care about minor GC time, therefore GC overhead is approximately equal to the major GC time:

$$T_{GC}(S_N) \approx T_{mj}(S_N) \quad (4.3)$$

4.5.3 GC Time Estimation

In order to estimate the difference in GC time at S_{eff} and S_{max} , we use an observed GC time from a recent garbage collection run. We record several GC times from recent runs but do not necessarily take a value from an arbitrary trace size, S_N , from the most recent run because this overestimates. Instead we choose a GC time value from a rather less recent GC run where S_N is close to either S_{eff} or S_{max} . Therefore GC time at both S_{eff} and S_{max} are computed as shown in the following equations:

$$T_{GC}(S_{\text{eff}}) = T_{GC}(S_{\text{neff}}) \quad (4.4)$$

$$T_{GC}(S_{\text{max}}) = T_{GC}(S_{\text{nmax}}) \quad (4.5)$$

Where S_{neff} and S_{nmax} are the trace sizes close to S_{eff} and S_{max} respectively. There is a trade-off between taking the GC time from the recent run since it is more accurate but it represents the current program phase and one from a less recent run but at a more realistic trace size.

4.5.4 Mutator Time Estimation

Mutator time is the rest of the time spent in the application program but in practice some GC-related activities like write barriers are executed by the mutator and are difficult to isolate. Therefore, similar to GC time estimation, we also use an observed mutator time from a recent run of the application; estimating the difference in mutator time, we have to make an estimate at a trace size of either S_{eff} or S_{max} . Therefore both S_{eff} and S_{max} are computed as shown in the following equations:

$$T_{mut}(S_{eff}) = T_{mut}(S_{neff}) \quad (4.6)$$

$$T_{mut}(S_{max}) = T_{mut}(S_{nmax}) \quad (4.7)$$

Where S_{neff} and S_{nmax} are the trace sizes close to S_{eff} and S_{max} respectively. We incur the cost of profiling for the first run of the application but improve performance during subsequent runs using the profiling information. In another alternate implementation, we can assume that at the effective trace size, S_{eff} , we will experience the most realistic and feasible profiling overhead.

We thereby can measure the mutator time at S_{bf} and another at S_{af} , which correspond to sizes just above and below the effective trace size respectively:

$$S_{bf} = S_N < S_{eff} \quad (4.8)$$

$$S_{af} = S_N > S_{eff} \quad (4.9)$$

$$T_{mut}(S_{eff}) = T_{mut}(S_{bf}) \quad (4.10)$$

$$T_{mut}(S_{max}) = T_{mut}(S_{af}) \quad (4.11)$$

4.5.5 Choosing the Optimal Trace Size

After the above estimations, a decision has to be made between running at either S_{eff} or S_{max} , depending on the tradeoffs by computing the difference of the estimates:

$$X(S_{max}) = T_{GC}(S_{max}) + T_{mut}(S_{max}) \quad (4.12)$$

$$X(S_{eff}) = T_{GC}(S_{eff}) + T_{mut}(S_{eff}) \quad (4.13)$$

$$\Delta X = X(S_{max}) - X(S_{eff}) \quad (4.14)$$

4.5.6 Implementation

From Algorithm 1, we start by recording mutator time on line 1, which is measured between trace-associated allocation calls until the maximum trace size is reached. This new trace is created but not committed for compilation, instead it is profiled on line 6.

During profiling, blocks in the trace are monitored and updated for frequency in exits to determine the hot exits. If an exit is hot, the trace is truncated and the size of the trace is estimated to be the effective trace size. We record both the effective trace size (S_{eff}) and the maximum trace size (S_{max}).

The change in mutator and GC time is estimated to decide if the new trace limit should be the maximum trace limit or the effective trace limit. The decision is done by evaluating the condition in Equation 4.14. Therefore, we evaluate the expression $\Delta T_{GC} + \Delta T_{mut}$ to determine if we should set the new trace limit at either S_{max} or S_{eff} .

Algorithm 1: GCAwareTraceSizing(*cf*)

Data: Input: Let *cf* be the program control flow, and *buffer* be the recording buffer

Result: An effective trace size

```
1 RecordMutatorTimeData();
2 while !StopTrace(cf) do
3   | buffer.append(cf)
4 end
5 /*trace profiling*/;
6 if trace != null then
7   | updateExitFreq(trace, pos);
8   | if checkIfHot(trace, pos) then
9     | abortTraceFormation();
10    | recordMaxTraceSize();
11    | recordEffTraceSize();
12   | end
13 end
14 /*Evaluate change in time*/;
15 if changeInGCTime + changeInMutatorTime > 0 then
16   | updateMutatorTime();
17   | triggerTraceAbort();
18   | setTraceLimit();
19   | recordGCData();
20 end
21 else
22   | no trace abort;
23   | doTracing();
24 end
```

When a decision is made to perform a trace abort, thereby setting the required trace limit, we record and update the mutator time records and GC time details to be used in estimating the Δ GC data. We integrate the profiling technique in RPython and evaluate it against two RPython-based dynamic languages PyPy and Pycket.

4.5.7 Evaluation

We evaluate the profiling technique on two RPython-based dynamic languages, namely PyPy and Pycket, which are Python and Racket implementations respectively. We use 10 benchmarks for each of the languages. The PyPy benchmarks are from the standard PyPy benchmark suite [37] while the Racket benchmarks used to evaluate Pycket are borrowed from the computer language benchmarks game (CLBG) [5]. Table 7.1 has a description for each of them.

We compare the execution times in running PyPy and Pycket using the default trace limit of 6000 with the dynamic trace sizing from the technique. For both PyPy and Racket, using dynamic trace sizing improves performance from 0.6% to as high as 12% as shown in Figures 4.12 and 4.13.

The benefits of dynamic trace sizing can be more than this. In real-world workloads that have had challenges with the default trace limit for RPython, like Aheui [59], setting an effective trace limit increased performance by $20\times$ [77] and yet this was done manually by guesswork without any profiling. Pydgin, a Python DSL for generating instruction set simulators, also benefited from manual trace size adjustments [77], which improved performance by $6\times$ in some workloads also by guesswork, therefore dynamic trace sizing has the potential to benefit very memory intensive workloads that pressure garbage collection.

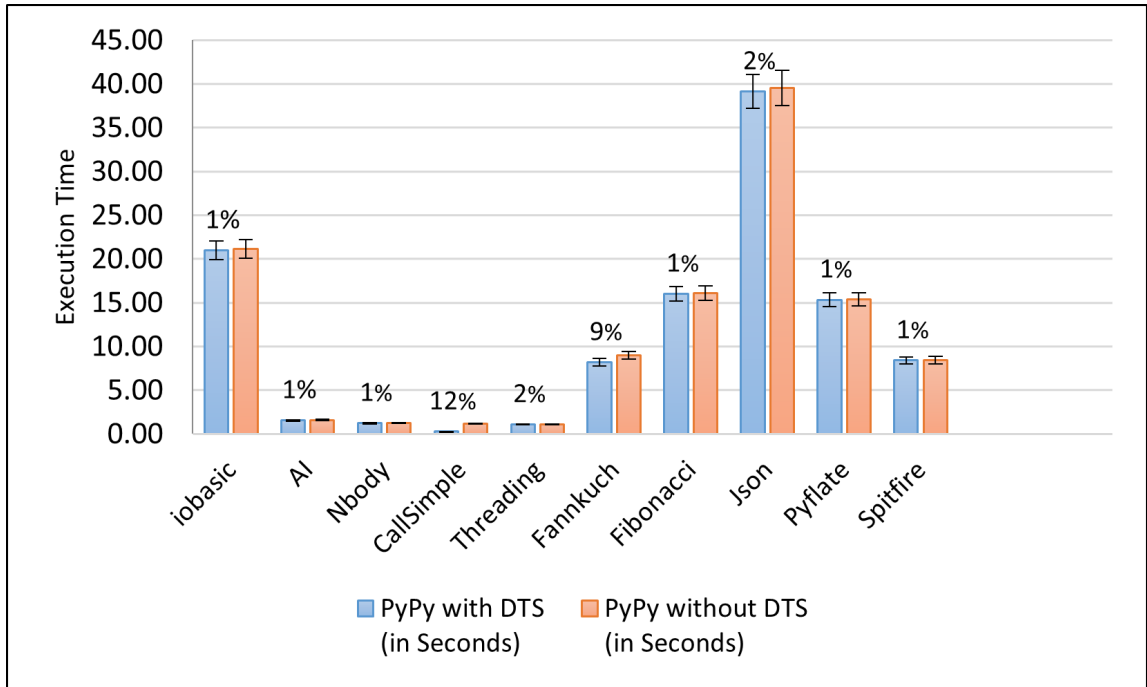


Figure 4.12: **PyPy Performance** — The execution time is measured in seconds. Dynamic trace sizing improves performance for Python applications to as high as 12% across the subset of PyPerformance benchmarks we evaluated

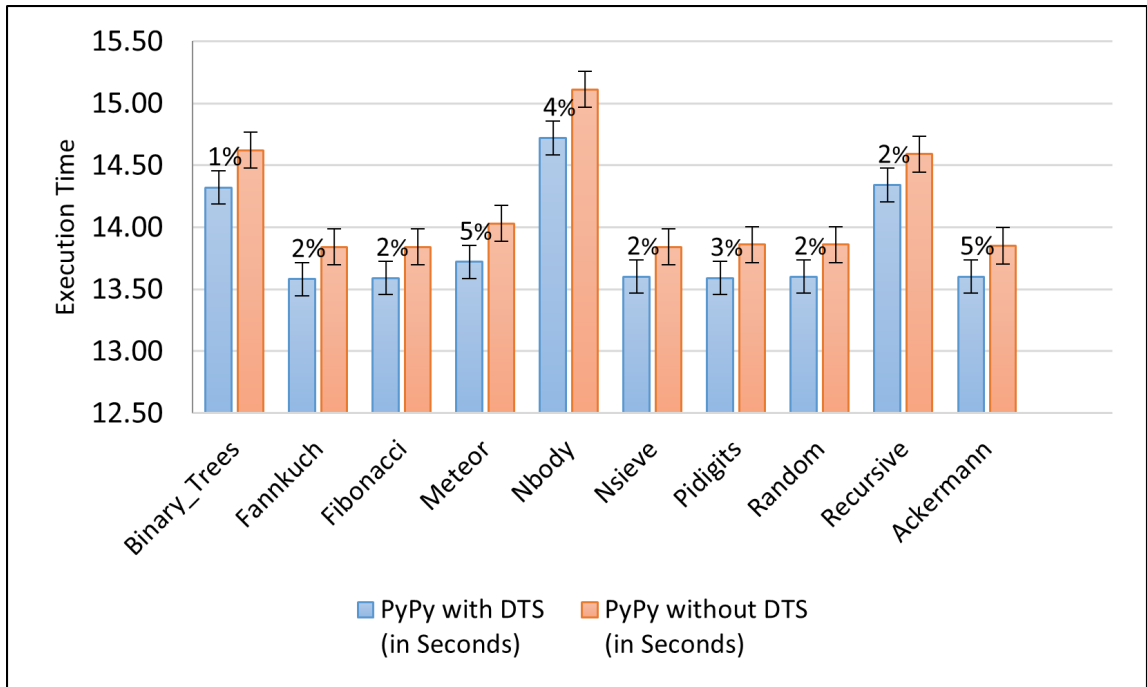


Figure 4.13: **Pycket Performance** — The execution time is measured in seconds. Dynamic trace sizing improves performance for Racket applications to as high as 5% across the Racket benchmarks we evaluated

Benchmark	Description
PyPy	
Json	Python Json processing
Spitfire	Python templating
Pyflate	Pure-Python DEFLATE (gzip) and bzip2 decoder/decompressor
Nbody	The n-body simulation
Threading	Threading support
Fibonacci	Recursive fibonacci
Fannkuch	Indexed-access to tiny integer-sequence
AI	A simple benchmark exercising machine learning
IObasic	File input and output
Call_Simple	Simple function calls that are not methods
Pycket	
Binary Tree	Binary trees implementation
Meteor	The meteor contest
Nsieve	Calculate numbers in a certain range
Recursive	Basic recursion
Fannkuch	Indexed-access to tiny integer-sequence
Nbody	The n-body simulation
Fibonacci	Recursive fibonacci
Pidigits	Streaming arbitrary-precision arithmetic
Random	Random numbers
Ackermann	The Ackermann function

Table 4.3: PyPy and Racket Benchmarks

4.6 Related Work

This work touches one key area: trace optimizations. Optimizing allocations in JIT traces was examined by Boltz et al [22], where escape analysis was used to remove any unwanted allocations in the generated traces for RPython-based dynamic languages. This optimization was merged in RPython but our work identifies that GC overhead is still a bottleneck for tracing JITs and proposes a technique to solve it.

Pape, Hayashizaki et al. studied adaptive JIT class optimization using profiling [107] but these optimizations do not consider the GC overhead during JIT tracing.

Hayashizaki et al. patented trace path profiling [67, 29, 51], which we build upon to create GC-aware trace profiling to reduce GC overhead that has significant effects in reducing execution time.

Our profiling technique is also similar to the nursery sizing technique described by Ismail et al [78]. but we focus on sizing the trace instead of the nursery. The technique described in their work can complement our work to help in cache-aware contexts.

Chapter 5

Type-based Stores for Collections in Dynamic Languages

Dynamically typed languages as noted in Chapter 1 are popular because they allow for rapid prototyping for their users but have performance and memory overhead due to several reasons [12, 70, 108, 56, 140, 141]. The focus of our work in this chapter is the inefficient low-level representation of data objects and as a result, unoptimized operation of these objects [60, 21, 9].

5.1 Introduction

The nature of dynamic languages makes it hard to statically determine the type of an object, as the type may change at runtime, which is a form of polymorphism [9]. One area where this polymorphism is heavily used is in the handling of collections. This is desired since collections in dynamic and many static languages can store values of multiple types. We use the terms *heterogeneous* and *homogeneous* for collections in different contexts for this work.

Heterogeneous Collections are data structures where a single container can have members of dissimilar types. For instance, consider the pseudocode for a Python

function in Listing 5.1, that computes the sum of select items of a list, `lst_het`, and defines another list, `lst_hom`.

```
1 def process_lst():
2     lst_het = [1, "foo", 3, "bar", 5, 6]
3     lst_het[0] + lst_het[2]
4     lst_het[0] + lst_het[1] # fails
5     str(lst_het[0]) + lst_het[1]
6     lst_hom = [1, 2, 3]
```

Listing 5.1: An Example of Data Structure Polymorphism

Heterogeneous collections as the one shown on line 2 of Listing 5.1 are typically represented uniformly, with a general object type, using a technique known as boxing or wrapping. When an operation is performed on an item, the language has to unwrap or unbox the value, to check its concrete type as a way of verifying whether the operation is acceptable for that type of value or not. In this example, lines 3 and 5 will execute correctly while line 4 will throw an exception, as it is forbidden to perform an arithmetic operation between a string and an integer.

Homogeneous Collections on the other hand contain items of the same type, for instance, the list `lst_hom` on line 6 of Listing 5.1. The literature addresses the challenges regarding collection boxing of values with optimizations like *storage strategies* and *element kinds*, both of which only handle homogeneous data structures [24, 34]. PyPy supports storage strategies for collections with a mix of floats and integers but does not do so for any other combination of types.

PyPy's storage strategies discussed in Chapter 2.4 assume that if any collections become heterogeneous, then they do so when the collection has few items. However, we show in Section 5.2 that for several Python workloads, storage strategies actually lead to worse performance. This is because, when a collection becomes heterogeneous, it is iterated, boxing any items that may have been unboxed before. Therefore storage strategies have two challenges, 1) they do not optimize heterogeneous collections and

2) they do not deal well with collections that become heterogeneous later in program execution.

Element kinds [34], also discussed in Chapter 2.4, are similar to storage strategies and minimize dehomogenization through initializing collections from a given allocation site using a strategy of a previously allocated collection from the same site [24]. This reduces the number of transitions a collection can have, but it also means a list that becomes homogeneous later will not be optimized if it already transitioned to a heterogeneous collection prior. For large collections, that transition to a general kind also incurs a boxing overhead. Therefore element kinds also share most of the same challenges as storage strategies.

We propose *type-based stores*, a new memory layout to handle heterogeneous collections through *collection splitting*. The layout introduces multiple contiguous stores to match the types of items in the collection, storing objects of the same type in each store, and can use the *strategy pattern* to manage the stores, as well as unwrap the values. We implemented the proposed mechanism in PyPy, a Python implementation, built using the RPython framework. We also demonstrate language-independent type-based stores in RPython and perform evaluation for Topaz, a Ruby implementation and Pycket, a Racket implementation, both RPython-based virtual machines. Our evaluation shows that while some individual Python collection operations see a noticeable slowdown, in PyPy benchmarks with more realistic mixes of operations, the overall performance is improved by a mean of 11.4%. The language-independent type-based stores also improve performance for Ruby and Racket applications by as high as 17% and 13% respectively while introducing overhead as high as 20% for both. The benefits are lower in Racket and Ruby due to the use of an RPython-based modular version of type-based stores.

The goal of the *type-based stores* technique is to find an efficient storage representation for collections containing values of a primitive type determined by the data

structure’s static types for dynamic languages where there is a significant possibility that collections are heterogeneous. We aim to optimize primitive types like strings in data structures where techniques like tagging are hard and complex to achieve. Tagging also has performance implications due to branch prediction. Type-based stores can be defined as contiguous storage areas created in memory, to hold items of the same type. We start with the motivation in Section 5.2 before discussing our solution and its evaluation in Sections 5.3 and 5.4, then end with the related work.

5.2 The Polymorphism Overhead

To motivate our study, we start with an analysis of the impact of storage strategies on Python applications, and also test the hypothetical case of a very large homogeneous collection becoming heterogeneous, common for applications with large collections that transition. The prevalence of strategy transitions at least suggests a potential for improvement.

We demonstrate the prevalence of heterogeneous collections in Table 5.2 and the overhead of an existing optimization, storage strategies, that only handles homogeneous data structures in Figure 5.1. The numbers in Table 5.2 are an approximation breakdown of how many collections transition from homogeneous integer, string and empty collections, to being heterogeneous. For this experiment we evaluate a subset of the standard PyPy benchmarks [37] modeled after the official PyPerformance Python benchmark suite [113].

The benchmarks are described in Table 5.1. Figure 5.1 also uses an artificial example, *list-example*, to create a homogeneous list of one million integers and transitions it to a heterogeneous collection by adding a string to it. The rest of the benchmarks are part of the standard Python benchmark suite. We reproduce the benchmark results of the storage strategies paper [24], observing two key limitations.

Benchmark	Description
Nqueens	The n-queen problem solver using different search algorithms
Pidigits-modified	A modified implementation that computes arbitrary digits of pi
Float	Heavy floating-point arithmetic
Richards	A Python implementation of the Martin Richards program
Delta Blue	Constraint solving problem
AI	An algorithm to exercise the performance of a simple AI system
Eparse	LXM parsing
Meteor-contest	An implementation of the Meteor puzzle board
Fannkuch	Indexed-access to tiny integer-sequence
Spectral-Norm	Calculating the spectral norm of an infinite matrix
Chaos	Fractals for the chaos game
Nbody	The original nbody problem solver as translated to Python
Telco	Measuring the performance of decimal calculations
Call_Simple	A trivial function call
Regex_Effbot	Working of Regex
Nbody-modified	The modified nbody problem solver as translated to Python
Unpack_Sequence	Unpacking a sequence
Fib	The fibonacci algorithm

Table 5.1: A Description of PyPy Benchmarks

	Total Number	Number of Transitions	Percentage (%)
Lists			
Integer	9,000,000	2,000,000	22.2
String	20,000,000	3,000,000	15
Empty	30,000,000	15,000,000	50
Sets			
Integer	1,500,000	20,000	1.3
String	30,000,000	30	1.5
Empty	5,000,000	200,000	4
Dictionaries (Key Types)			
Integer	120,000	50,000	41.7
String	350,000	30	0.2
Empty	2,000,000	100,000	5

Table 5.2: **Collection Strategy Transitions** – an approximate breakdown of collection transitions to the object strategy during program execution while running the PyPy benchmark suite for benchmarks described in Table 5.1 . For some key container type combinations, as much as 40%–50% of collections become heterogeneous from a specific type like integer, string or empty

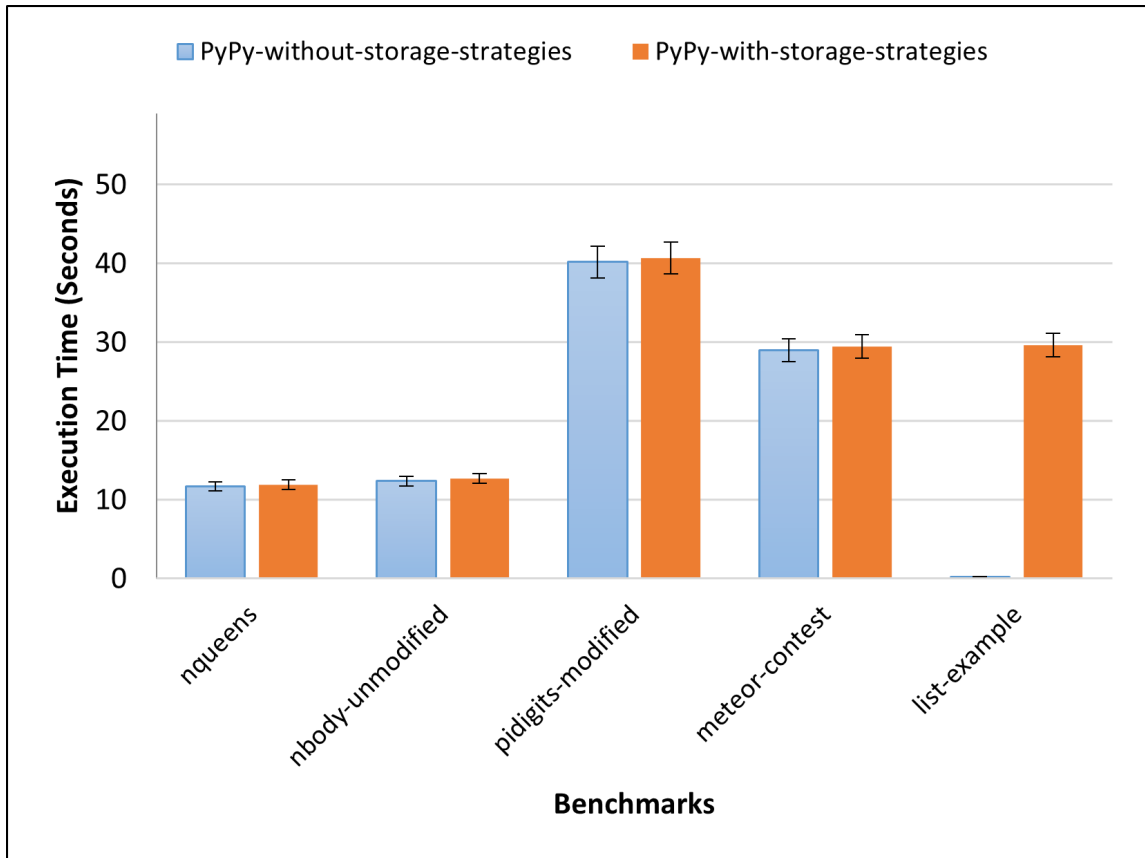


Figure 5.1: **The Overhead of Storage Strategies** – the unoptimized heterogeneous collections and the impact of strategy transitions slows down applications. This leads to an overhead of 4% by mean for the four benchmarks we ran from the PyPy benchmark suite described in Table 5.1. Transitioning a list of one million integers to the object strategy, takes the program 25 seconds with storage strategies while it takes less than five seconds without storage strategies

First, we show that about as high as 50% of collections in our benchmarks are heterogeneous and thereby remain un-optimized by storage strategies as shown in Table 5.2. We observe that list-based collections transition the most by about 50% from being empty. Equally significant are list transitions from being integer-based, which impacts about 22% of the lists. Lists are followed by dictionaries where about 42% of integer-based dictionaries become heterogeneous. Transitions from being empty are also significant at about 100,000 lists, a point capable of introducing overhead. Sets experience the least transition to the general object/wrapped state, the highest being 4% transitions from being empty. We do not track the distribution

of collection sizes when a transition happens but the speed overhead discussed next elaborates on the overhead storage strategies.

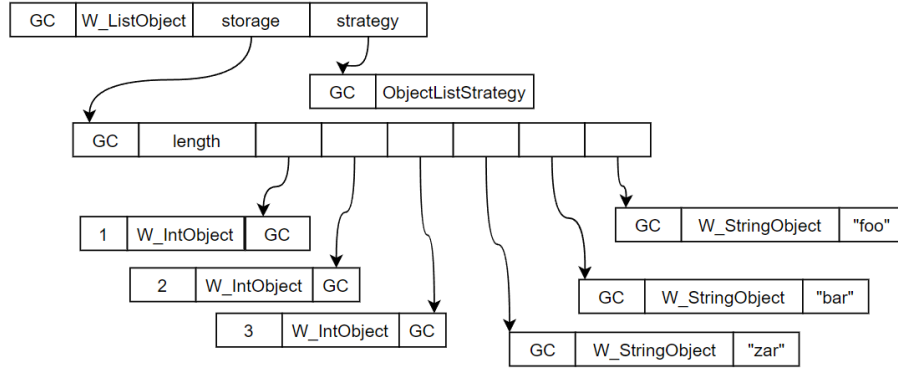
Secondly, optimizing for only homogeneous collections degrades performance for some workloads with large collection sizes due to the need to re-box items when a homogeneous collection becomes heterogeneous. When collections become heterogeneous, we observe a slowdown in speed when storage strategies are used. To our second observation, to understand this overhead, we ran the five benchmarks in Figure 5.1, against two PyPy versions, one with storage strategies turned off and the other with storage strategies turned on. We only exercise lists since they are the most used data structures but insight from the experiment is consistent with sets and dictionaries. The slow down is because every item in the list is revisited to box the items again at the point of becoming heterogeneous.

In Figure 5.1 the hypothetical example is interesting; when we transitioned a list of one million integers to the object strategy, the program took 25 seconds with storage strategies to run while it took less than five seconds without storage strategies. We also observe similar slow downs of about 4% by mean with the other four PyPy benchmarks, though not as dramatic as the hypothetical example. This shows that optimizing for heterogeneous collections is potentially beneficial for some Python applications and storage strategies instead worsen the performance of such applications.

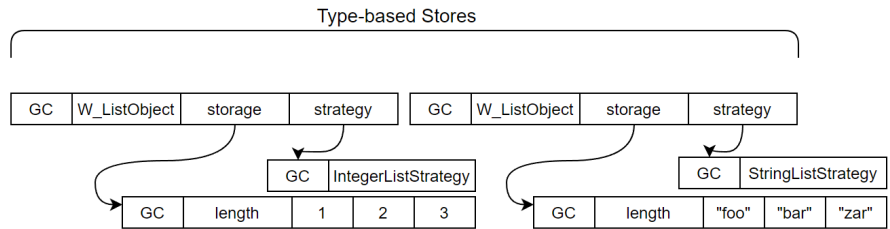
5.3 The Type-based Memory Layout

We do not attempt to forward homogeneous collections to the type-based stores technique, as these are already handled by existing optimizations. Instead, the heterogeneous data structure is broken down by categorizing its members by type creating two stores in this example: one to hold the strings and the other for integers. We use contiguous storage areas that are expandable and capable of holding both

sequential and hash-like data.



(a) Storage Strategies



(b) Type-based Stores

Figure 5.2: **Type-based Stores vs. Storage Strategies** – consider a list `lst = [1, "foo", 2, "bar", 3, "zar"]`, (a) shows the memory layout of the list after applying storage strategies; and (b) shows the memory layout of the list after applying type-based stores and storage strategies

For example, the heterogeneous collection `lst = [1, "foo", 2, "bar", 3, "zar"]`, can not be optimized with existing techniques like storage strategies. Instead, the items stay wrapped, with types `W_IntObject` for integers and `W_StringObject` for strings as shown in Figure 5.2 (a). However, with type-based stores, shown in Figure 5.2 (b), the numeric items `1`, `2`, `3` are stored in a separate contiguous area, while the textual items `foo`, `bar`, `zar` are stored in another adjacent area, each storage area managed by a storage strategy which unwraps the items.

The stores are created on demand, and there is a one-to-one mapping between a data structure and its respective internal set of stores. Therefore given a collection L , and an internal layout D , a mapping function exists, f , from L to D , for all d in D and l in L such that, $f(d) = l$. The associated stores of a collection are dynamic, to expand

and contract, matching the size of the collection they represent. Therefore, at all times, $|L| = |D|$ as discussed further in Section 5.3.2.

For sequential data structures, the indexing in the sequential storage area is implied. The size N of each storage area in the sequential parts is computed dynamically, every time the internal layout has to resize as the largest power of two such that at least half the elements in the stores will be filled. A mapping is also needed between the source collection and the type based stores.

The goal of restructuring the collection to use stores is to allow for further optimization. Therefore, when the collections have been split into the different stores by type, we take advantage of the RPython methods that are based on the `rerased` RPython library module to unbox the items in each store.

The `rerased` library provides two main functions of interest to our work, namely; `erase` and `unerase`. The `erase` method wraps the object by erasing its type, returning a generic type which is the equivalent of a void pointer in C or `Object` in many languages while `unerase` unwraps the objects in the storage area to access their actual types. We mostly use `erase` since unwrapping is core to our technique and call `unerase` to wrap objects for cases we do not support.

5.3.1 Memory Layout Analysis

Our technique determines how elements of a heterogeneous collection are allocated in memory using a restructure function that creates a set of stores for a collection, placing elements in one of the following relative positions:

$\{colocated, close\}$

Given an arbitrary collection, `lst = [X1, X2, ..., Xn]`, we define a *colocate* operation, \sim to show that two objects are in the same store. For any given heterogeneous

collection, a layout mapping exists with several stores as follows:

$$\text{layout}(\text{lst}) = \begin{cases} \text{store } 1 & X_1 \sim X_2 \sim \dots \\ \text{store } \dots & \dots \sim \dots \sim \dots \\ \text{store } k & X_{n-2} \sim X_{n-1} \sim X_n \end{cases}$$

Any of the items will be colocated in the same store, for example, $X_1 \sim X_2$, if they are of the same type and *close* (X_1 and X_{n-1}) if not in the same store but in the same set of stores belonging to a given collection.

At the point of restructuring the collection, triggered by creation or modification, every item should be allocated in one of the stores. We optimize for collections containing a mix of primitive types like strings and integers but if a collection contains non-primitive types, optimization (splitting to stores) is not supported yet, so we do nothing in this case and delegate to wrapping. At all times, two collection primitive items of different types will be colocated only if they have not been mapped to any stores yet.

5.3.2 Internal Mapping after Restructure

The consequence of reorganizing the original data structure, for sequential collections, is that there has to be a way to track items in the stores and their positions in the original collection to help with access and modification by any operations. We maintain a non-trivial map, implemented efficiently as a hash, that is updated on collection creation/update and read when there is a read request to the items of the collection. This map is the same map referenced in the storage strategies paper, PyPy already also uses maps to handle certain structures, like instances.

Figure 5.3 shows how items in a list, `lst = [1, "foo", 2, "bar", 3, "zar"]`, are laid out in memory, and a map used to track positions of items in the source data

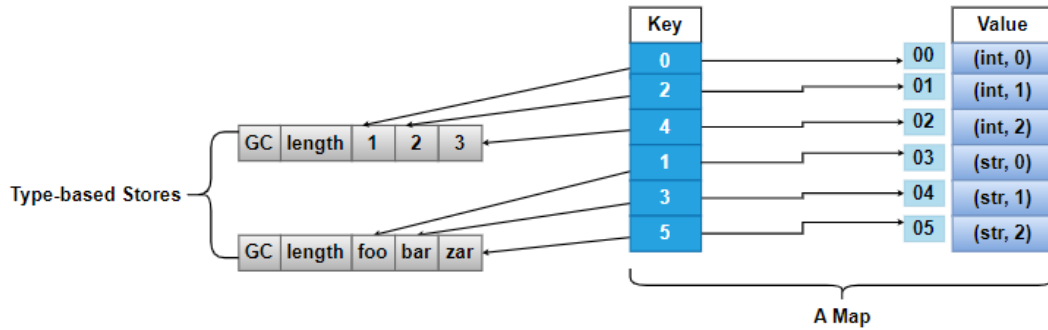


Figure 5.3: **Internal Mapping to Support Operations** – the map tracks positions of items in the source data structure to help with processing access operations. This map is useful for sequential collections

structure. For purposes of easier presentation in this figure, the map is rearranged to group values of the same type together. A typical access, `lst[i]`, first checks the map to acquire the position in the stores for an item at this index.

The values in the map, should have an indication of the type so that during the search, we can distinguish between two similar indices that may refer to different stores. This map introduces an extra step in collection access but as we show in the evaluation section, the benefits of unwrapping the items outweigh this overhead. The pseudocode below shows the steps to access `lst[i]` in a given representation.

```

1 store = lst.map[i] -> (int, 0)
2 for stor in lst.stores()
3     type = store[0]
4     if stor.type == type
5         index = store[1]
6         item = stor[index]
```

Listing 5.2: The Internal Map Access Algorithm

First, we search the internal map by index `i` as shown on line 1 of Listing 5.2, which returns a tuple containing the item type and its index in the store. On lines 2–4, we search the collection stores by the type, retrieving a store that matches the type of

item we want to access. We use the index read from the map to access the item in the retrieved store on line 6.

5.3.3 Dictionaries

Type-based stores for dictionaries can be applied for both keys and values, but we do not take this route of unwrapping each pairwise (key, value) primitive type combination. We instead optimize for hashing and comparing of keys by creating multiple stores to match the types of keys in the original data structure as shown in Figure 5.4.

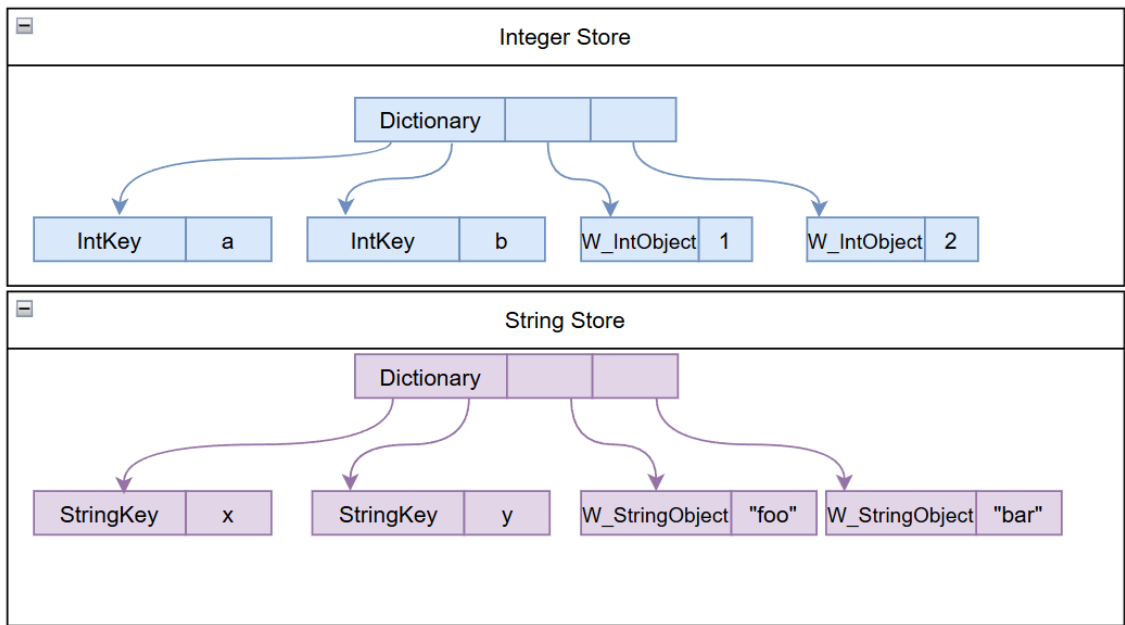


Figure 5.4: **Type-based Stores for Dictionaries** – we only optimize keys to reduce the cost of handling every possible key/value type combination when unwrapping

Consider a dictionary, $\{a: 1, b: 2, x: \text{"foo"}, y: \text{"bar"}\}$, where a and b are integer keys while x and y are string keys. The keys are unwrapped while values remain wrapped. This is a good compromise because keys are more stable in many applications than values in terms of type transition [24] but the many type combinations if we were to handle both keys and values can also easily become costly.

We also do not create a separate internal map for dictionaries, instead dictionary operations search the associated stores by key, since duplicate keys are not allowed. This reduces the bookkeeping for dictionaries.

5.3.4 Evaluation

The goal of our evaluation is to compare the PyPy version based on our technique with a default PyPy baseline version that does not optimize heterogeneous data structures, i.e., the one based solely on storage strategies.

We use this baseline for two main reasons 1) PyPy, the Python implementation used, has storage strategies in its default upstream branch, which means all PyPy users currently use storage strategies making it the feasible baseline by virtue of widespread use and 2) turning off storage strategies completely would involve a major overhaul of the interpreter because all major data structures including classes are affected, which is unrealistic and out of scope of our research.

Optimizations that reduce the boxing overhead mostly impact execution time but also reduce the memory footprint. We therefore use both metrics in our experiments and attempt to answer the following questions:

1. *How much, if at all, does the technique improve the execution speed of Python applications?*
2. *How much, if at all, does the technique reduce the memory consumption of Python applications?*
3. *How much, if at all, does the technique introduce overhead to collection operations?*

Methodology We use a subset of the standard PyPy benchmark suite, mainly the unladen swallow benchmarks, based on the standard *Pyperformance* benchmark suite.

Property	Specification
CPU	Intel(R) Xeon(R) Gold 6248
Clock Speed	2.50GHz
Operating System	64-bit Debian 10.2.1
Cores	10
GCC Version	10.2.1

Table 5.3: Hardware Setup

Pyperformance benchmarks are designed to emulate real-world Python applications. To generate results exercising all supported collection operations, we used artificial workloads. We use the hardware setup detailed in Table 5.3. We run the benchmark programs 30 times, ignoring the first 5 runs; this is not aimed at determining accurate steady-state because it is impractical, but rather a large number of iterations increases the likelihood of getting close to steady-state. The acronyms TBS and NTBS, denote, type-based stores and no type-based stores respectively.

Execution Speed Figure 5.5 shows the execution time. To answer the first question, we observe that the type-based stores technique improves performance for 11 of the 16 benchmarks. Of these, five benchmarks (*float*, *richards*, *deltablue*, *ai*, *regex-effbot*) have speedups of 20% and above and the rest are 10%–19% or slightly below (*spectral-norm*, *call-simple*, *eparse*, *meteor-contest*, *fannkuch*, *nbody-modified*). Of these 11 benchmarks, storage strategies alone could not improve performance.

We estimate the magnitude of this speed to be 11.4% for the 16 benchmarks, a number we arrived at by calculating the geometric mean increase (positive speed ups) and slow down (negative percentages) for the benchmarks. For specifics, the speedup is between 2% to 40%. The slow downs are between 2% to 60%.

The benefits of our technique can be better appreciated by comparing the boosts and slow downs in the original homogeneous-only optimization in the storage strategies work [24] with our results. Benchmarks like *fannkuch*, *meteor-contest*, *richards*, and *spectral-norm* performed worse when only homogeneous collections were optimized, but

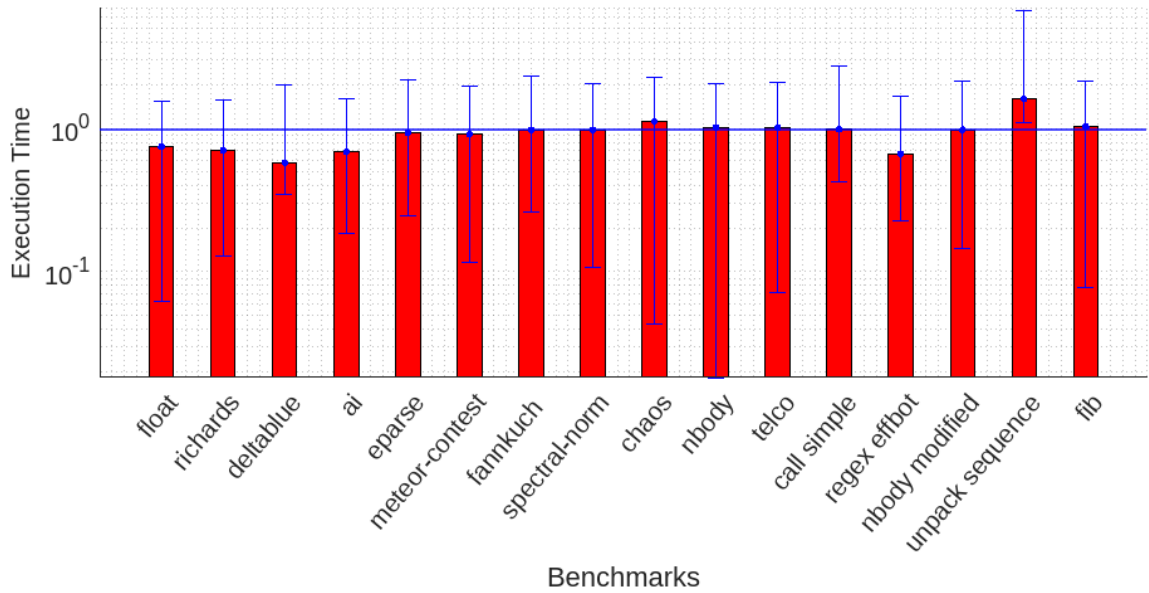


Figure 5.5: **Speedup for PyPy** – the results are normalized to the storage strategies baseline, lower is better, the type-based stores technique accelerates execution times for most of the benchmarks

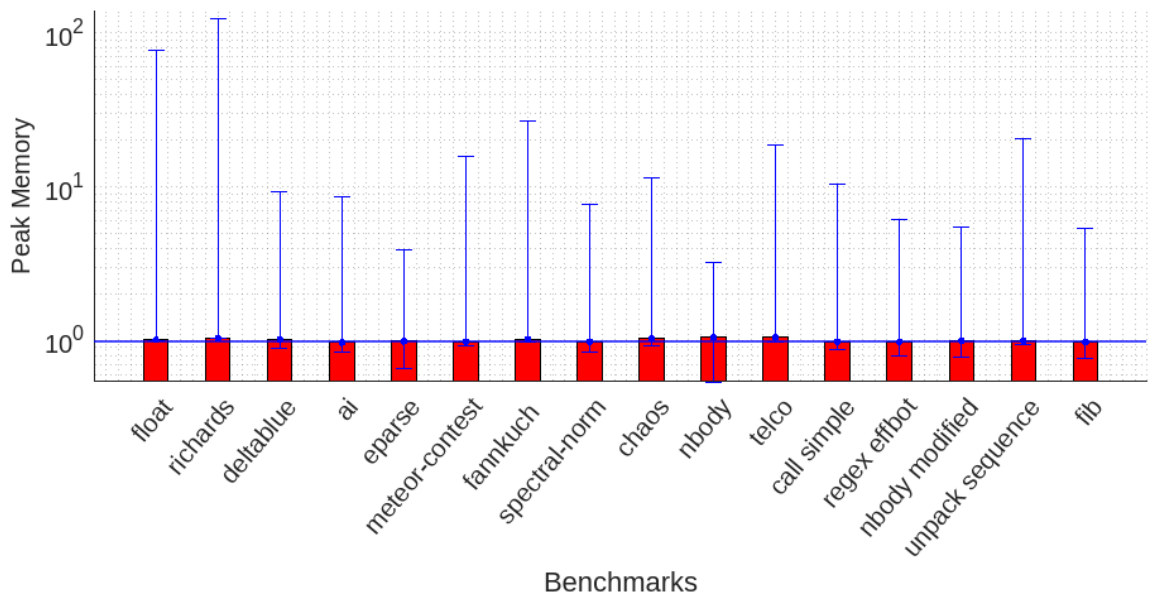


Figure 5.6: **Memory Savings for PyPy** – the results are normalized to the storage strategies baseline, lower is better, the type-based stores technique reduces the memory footprint for some benchmarks and increases the memory footprint for some others

with our technique they are faster, which means heterogeneous collection optimization is beneficial to them.

On the other hand, looking closely at the benchmarks where our technique slows down performance, the slow down is observed for benchmarks like *telco* and *nbody* that ran faster with just the homogeneous collection optimization. This is because we perform extra checking in the algorithm before splitting, even though the algorithm does not optimize homogeneous collections; these extra checks are overhead. These benchmarks are fewer, and by default dynamic languages assume polymorphism, so we can optimize for language defaults. The *chaos* and *unpack-sequence* benchmarks are an outliers as they run slower for both the homogeneous and heterogeneous optimization.

Memory Consumption Figure 5.6 shows the memory consumption. To answer the second question, these results are mixed; our technique uses less memory in some benchmarks and more memory in other benchmarks. The benchmarks *ai* and *eparse* run faster, and still use about 3% and 1% less memory respectively; however, while the *float* and *fannkuch* benchmarks run faster, they use more memory, about 1% - 2% more respectively. The benchmarks *fib* and *unpack-sequence* run slower but use less memory.

Our technique generally has to reduce the size of objects, so we expect less memory consumption, but we introduce a memory layout with a map that may use more memory in some cases. For example the *richards* benchmark runs faster but uses more memory with type-based stores. The storage strategies paper [24] also shows increased memory usage for some benchmarks, so we can expect that type-based stores will inherit the same behaviour because of using strategies in the stores.

The reason is that without storage strategies, a single object is allocated once on the heap, with multiple pointers to it from collections and elsewhere. With storage strategies, the same element can be unboxed in the storage strategy, and then, when

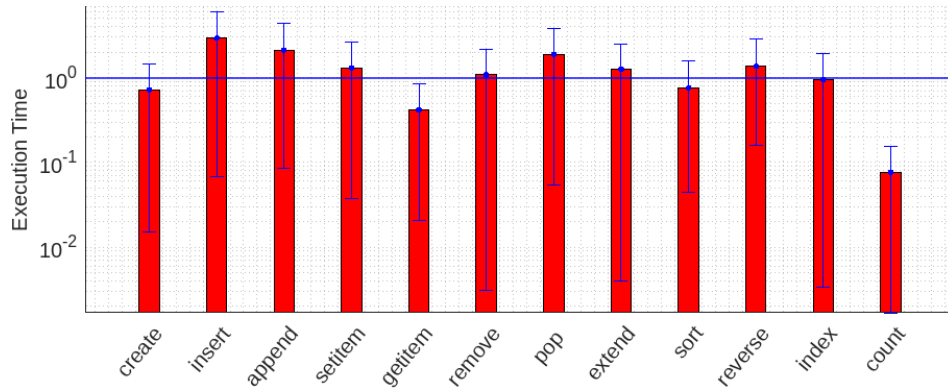
pulled out of the list, reboxed to multiple distinct objects in the heap. This is because some items that are unboxed inside a list are later used outside and reboxed.

The Overhead of Type-based Stores We base our analysis on the results shown in Figure 5.7. To answer the third question, several operations slow down due to the extra checks we do before creating or accessing the stores of a collection while others are not affected.

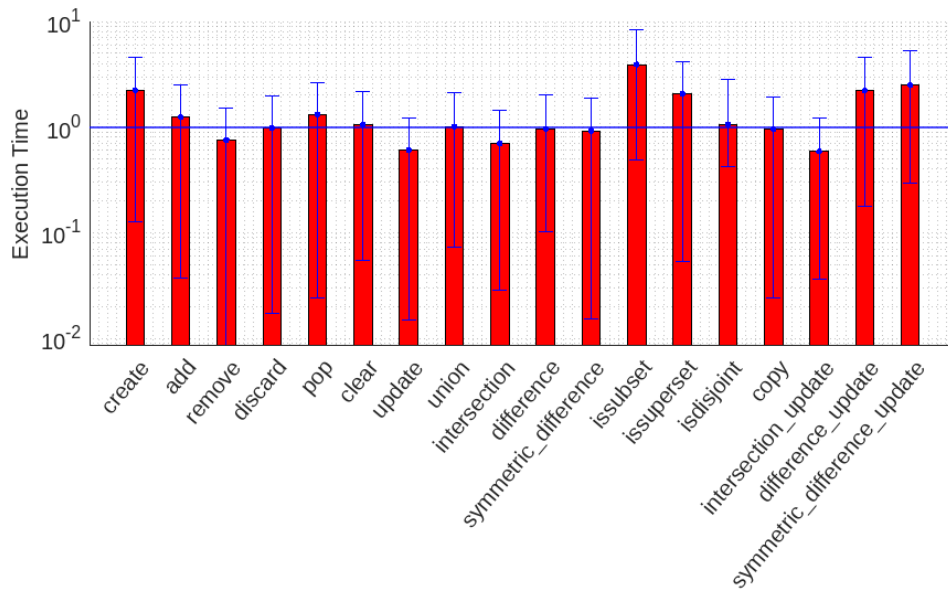
Despite some extra checking and processing of stores, some list operations are faster, specifically `getitem()`, `sort()`, `count()`, and `create()` are 23%–60% faster, `remove()` runs at almost similar speeds with overhead no more than 8%, `append()` and `insert()` operations cost about 100%–200% overhead due to the extra search and bookkeeping required for indices. Similarly the other operations like `setitem()`, `pop()`, `extend()`, and `reverse()` experience overhead between 25%–90%.

Similarly, most of the set operations run faster with our technique, for example, about 24% better for the `remove()` operation, 39% better for the `update()` operation, 30% better for the `intersection()` operation and 41% better for the `intersect_update()` operation. However, the `difference_update()`, `create()`, and `symmetric_difference_update()` operations are almost twice as slow due to extra checks and collection of items from the different stores. Sets are processed as dictionaries, so some of the overhead related to dictionaries discussed next, applies. Due to not requiring any synchronization of indices between the original dictionary and the stores using the map, dictionary operations like `fromkeys()` and `create()` are less impacted and run at approximately the same speed. However, we experience about 500% to 800% slow-downs for the `set()` and `update()` operations because of extra checking, and creation of stores. Generally, most dictionary operations have overhead with our approach.

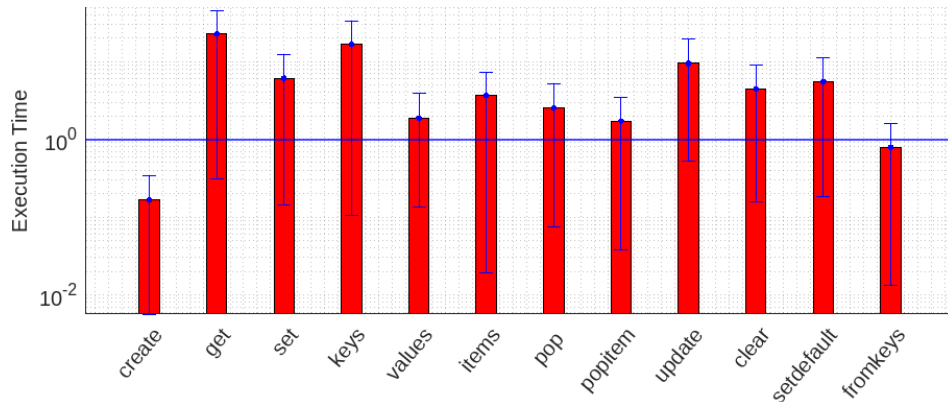
As noted above, there is an overhead observed for a few operations, where we added an extra step of creating the stores, but also where some checks are required to



(a) Lists



(b) Sets



(c) Dictionaries

Figure 5.7: **The Overhead of Collection Operations** – the results are normalized to the storage strategies baseline, lower is better

access certain items in the collection. However, as we discussed in Section 5.3.4, this overhead is insignificant and does not have much impact on the general speed for most applications, the unwrapping performance boosts outweigh the overhead in these few operations for most applications.

5.4 Language-independent Type-based Stores

Storage strategies were generalized for use by RPython virtual machines (VMs) in a library called *rstrategies* [108]. The library was ported to several languages. This work also extends this library to optimize heterogeneous collections in a language-independent way. This section discusses our algorithm for providing type-based stores in *rstrategies*, along with an integration to two RPython VMs, Topaz and Pycket.

5.4.1 Overview of Language-independent Storage Strategies

The *rstrategies* library is based on the use of metaclasses and metaprogramming concepts rather than making normal API calls, due to the complexity involved in handling collection types in the generic manner required and the need for flexibility. Integration with an RPython VM requires that a collection implementation adheres to the use of a central class structure or hierarchy. All RPython classes implemented in the VM can duplicate attributes and functionality from the library by importing `mixin` classes. The VMs we experiment with all follow this structure and the actual integration of the baseline *rstrategies* library involves three main steps.

The first step is to implement getter and setter methods in the VM collection class to expose the strategy and storage attributes required to optimize collections using storage strategies. These methods can be implemented manually but *rstrategies* also uses a method, `make_accessors(strategy= 'strategy', storage='storage')`, that can be called within the VM collection class to automate the implementation of these

getter/setter methods.

Secondly, `rstrategies` provides mixins that should be used to create strategy classes, starting with a single root strategy class, `AbstractStrategy`. The `AbstractStrategy` class is an interface that should be subclassed to specific strategy classes, to implement the methods used to interact with various storage strategies. The specific strategy classes match the known strategies for the different object types like `IntegerStrategyClass`, etc. The strategy classes can use a decorator which signals to the strategy class what strategy to switch to in case the current collection cannot be assigned to the current strategy, `@rstrategies.strategy(generalize=alist)`. This usually happens when an incoming item to the collection is not of the same type as the items in the current collection.

Then thirdly and finally, VMs have to subclass a class with some modifications (`StrategyFactory`) to any methods responsible, like `instantiate_strategy` for strategy initialization, switch actions (`switch_strategy`), etc. This allows the VM collection operations to use the storage strategies mechanisms of unboxing.

5.4.2 Extending `Rstrategies` with Type-based Stores

In extending `rstrategies` with type-based stores, we modified the mechanism responsible for switching between storage strategies. The switch that we care about for this work is where a concrete strategy is switching to the general strategy.

Therefore, inside `rstrategies`, we intercept the step that is responsible for strategy classes (step two in Section 5.4.1 above), modifying how generalization works. We define generalization as a point where the current strategy class notices an incoming item of a different type and defers to a routine that assigns the `ObjectStrategy` instead, and boxing items as required.

We specifically provide a decorator `rstores` that can be applied to strategy classes in the VM, that invokes type-based stores instead of switching to the `ObjectStrategy`

when a collection becomes heterogeneous. We therefore do not directly modify generalization, the type-based stores feature is optional, and if the `rstores` decorator is not applied, we default to assigning the `ObjectStrategy`.

The `ObjectStrategy` can also be assigned when you apply the `rstores` decorator for aspects we do not support yet, like instances. Therefore, anything the strategy class can not handle through a concrete strategy or `rstores` will default to generalization.

```
1 def generalize_for_value(self, w_self, value):
2     .....
3     try:
4         strategy_type_value = self.get_strategy_type(value)
5         create_type_based_stores(self.strategy_factory(), w_self,
6                                 strategy_type_value, value)
7         .....
8     except (ValueError, AttributeError):
9         pass
10    .....
11 def create_type_based_stores(self, collection, strategy, *args):
12     self.store_one = collection
13     list_w = collection.strategy.get_storage(collection)
14     self.update_map(self.store_one, list_w)
15     self.store_two = allocate_storage(value, size)
16     self.store_two.strategy.append(self.store_two, list(args))
17     self.update_map(self.store_two, list(args))
```

Listing 5.3: **Language-independent Type-based Stores** – the method, `generalize_for_value` is called at the point where an incoming item does not match the items in the current collection, and thereby requires generalization to the `ObjectStrategy`

Listing 5.3 contains pseudocode for the routine that handles collections that are not homogeneous. We use an example where a new object value of a different type is appended to a homogeneous collection, `w_self`. On line 1, we override the

generalization method, `generalize_for_value()`, `self` is the current instance of the collection class. We first process the type of the incoming item `value` on line 4. Then we invoke the creation of the required stores on line 5, the items in the current collection, `w_self`, the incoming item, `value`, and strategy type of this item, `strategy_type_new`, are taken as arguments.

When creating the stores, the first store (line 12) is for the existing homogeneous collection, `collection`, which was already allocated a storage strategy of appropriate type and size, all items unwrapped. The second store on the other hand is allocated a storage strategy on line 15, for the incoming item type, and line 16 stores the items, in the store in unwrapped form. For both stores, we have to update the map on lines 14 and 17, tracking positions of the items in the stores, to allow for future access of the collection items.

5.4.3 Evaluation

We applied the language-independent type-based stores, implemented in the `rstrategies` library to both Topaz and Pycket with the new `rstores` decorator. To assess the benefits of the technique, we compare versions of the implementations, using just storage strategies as a baseline, in this case through `rstrategies` with a version that has the type-based stores extension.

We use this baseline because of similar reasons as PyPy 1) The language implementations use storage strategies by default and 2) turning off storage strategies through `rstrategies` requires modifications to each language implementation, and are impacted by the collection structure of the language, which is an overhaul that is out of scope of this work.

For Ruby, we use benchmarks from the YJit project by Shopify [124] and add a few more benchmarks from the standard computer language benchmarks game. For Racket, we use benchmarks from the computer language benchmarks game [5]. All

benchmarks are run on a machine with specifications described in Table 5.3. The numbers are a geometric mean of 30 invocations for each benchmark. We do not describe each benchmark, but the links point to the relevant documentation.

Pycket			Topaz		
	Speed	Memory		Speed	Memory
nbody	1.0566 (-1.028 +1.085)	1.0119 (-1.011 +1.012)	nbody	0.9824 (-0.949 +1.016)	0.9997 (-0.996 +1.003)
meteor	0.9673 (-0.927 +1.009)	0.6892 (-0.688 +0.689)	fibonacci	0.9339 (-0.847 +1.030)	0.9997 (-0.996 +1.003)
spectral	0.8345 (-0.723 +0.963)	0.9978 (-0.994 +1.001)	binary tree	1.0052 (-0.970 +1.041)	1.0005 (-0.999 +1.002)
triangle	0.9399 (-0.911 +0.969)	1.2530 (-1.252 +1.254)	throw	1.2372 (-1.320 +1.159)	1.0054 (-1.003 +1.007)
vector	1.0319 (-1.018 +1.046)	0.9876 (-0.987 +0.989)	getvar	0.8699 (-0.776 +0.976)	0.9998 (-0.999 +1.000)
bubble	1.0246 (-0.978 +1.072)	1.271 (-1.267 +1.016)	setvar	0.9686 (-0.946 +0.991)	1.0015 (-1.000 +1.002)
nqueens	1.0159 (-1.002 +1.030)	0.9969 (-0.996 +0.998)	respond	0.9686 (-0.939 +0.999)	0.9956 (-0.991 +0.999)
puzzle	0.8768 (-0.849 +0.906)	1.2584 (-1.256 +1.260)	keywords var	0.9138 (-0.848 +0.984)	1.0000 (-0.999 +1.000)
fannkuch	1.0222 (-0.977 +1.069)	1.0014 (-0.997 +1.006)	gc array	0.9830 (-0.967 +0.999)	0.9995 (-0.998 +1.001)
hash table	1.1796 (-0.832 +1.672)	0.9984 (-0.995 +0.998)	sieve	0.9454 (-0.885 +1.009)	1.0049 (-1.002 +1.007)
CTAK	0.9787 (-0.889 +1.077)	0.9969 (-0.867 +1.146)	spectral	0.9466 (-0.791 +1.133)	1.0005 (-0.998 +1.002)
dot	1.1994 (-1.155 +1.245)	1.0043 (-1.003 +1.005)	fannkuch	1.0066 (-0.979 +1.035)	0.3110 (-0.309 +0.312)
min	0.8345	0.6892	min	0.8699	1.0009
max	1.1995	1.2610	max	1.2372	1.0054
geomean	1.0106	1.0388	geomean	0.9801	0.9958

Table 5.4: **Language-independent Benefits of Type-based Stores** – lower values are better, all results are shown as ratios normalized to the baselines. The language-independent type-based stores improve performance for both Ruby and Racket applications

The goal of our evaluation here is to assess the impact of language-independent type-based stores on Ruby and Racket benchmarks for language runtimes using the RPython meta-tracing JIT-based framework. Table 5.4 has the results for Pycket and Topaz.

Pycket For Pycket, we provide full support for vectors and partial support for hash tables. Both mutable and immutable properties of vector operations are supported.

As shown in Table 5.4, the highest speed gains are from the *spectral norm* benchmark with about 17%. The lowest gains are from the CTAK benchmark with just about 2%. We can conclude that type-based stores highly impact applications with collections for Racket and still work well for applications that may not be heavily dependent on collections.

We also observe some overhead in some benchmarks, the highest slow down is registered in the *dot* benchmark of about 20%. For benchmarks with mostly homogeneous collections, we incur overhead due to an extra check to optimize heterogeneous collections, so this is expected and minimal; the *dot* case is an outlier with the highest overhead here.

As shown in Table 5.4, the highest saving in memory is from the *meteor* benchmark, about 32%. This means that type-based stores can improve both speed and memory usage. There are cases where the optimization improves performance at the expense of using more memory, case in point, the *triangle* benchmark, which gains in speed but uses 25% more memory. This can be due to the extra book-keeping in maps for type-based stores to work.

The *dot* benchmark also uses more memory, which can be linked to the fact that compared to the storage strategies branch, it uses mostly homogeneous vectors that work well with an optimization that assumes homogeneity, therefore the extra processing for type-based stores is just overhead for this benchmark without much benefit to it.

Topaz The integration to Topaz was more complete than Pycket, with all collection data structures and operations supported. Our optimization has the most speed gains for the *getvar* benchmark, 13%, and the least gains for *GC array*, which also shows benchmarks with data structures can run with accelerated performance. We also register overhead sometimes as high as 19.2% for the *throw* benchmark.

A direct integration of type-based stores may accelerate speed gains if the overhead of calling type-based stores functionality through a library is eliminated. This applies to Pycket too but we do not quantify the overhead, because we considered it out of scope for now.

Other benchmarks that do not heavily rely on collections like *setvar* and *respond* also experience about 3% overhead. Like Pycket, memory is saved in some benchmarks, like *nbody*, and more memory is used in other cases like the *sieve* benchmark. The memory savings and overhead are all negligible in the Topaz benchmarks, all barely 1% in absolute value.

5.5 Related Work

Much work exists that uses techniques like tagging, and new frameworks [28, 12, 85, 70] to represent the types of data in dynamic languages [137], but regardless of these efforts, we have not been able, as a field, to completely avoid boxing in dynamic languages; efforts to date have targeted specific use cases. This work extends existing work to handle type representation of heterogeneous collections, which the literature has not solved.

Type optimization for collections has been researched with various levels of success, the most recent, in Graal [68, 43]; storage strategies were studied to aid list presizing, the Pharo language was modified with similar optimizations to reduce resource usage [9, 96], even parallelization algorithms exist for collections in dynamic languages [42] but none of these optimize heterogeneous collections.

Storage strategies [24] and element kinds [34] discussed in chapter 2.4.3 are similar optimizations to the ones we describe in this paper. We extend their use with modifications to support cases they do not support by efficiently handling heterogeneous collections. Storage strategies were made language-agnostic by Pape et al. [74] in

their work on the RPython translation toolchain; we also support the type-based stores mechanism in the RPython framework.

Memory restructuring is also not new; Aleksandros et al. [131] reshape the layout for objects to address locality and the Collection Skeleton technique [54] also provides flexibility in handling collections, but neither handle unboxing. D'Souza et al. solve parametric polymorphism [48] instead.

Chapter 6

Context Aware Presizing for Dynamic Languages

In dynamic languages, dynamism also means that data structures in an application are mostly designed with the potential to change size. This chapter explores an optimization for collection resizing.

6.1 Introduction

Data structure resizing allows for the expansion and shrinking of collections as shown in Listing 6.1 on lines 4 and 6. To support this flexibility, first, collections are internally over allocated, for example the lists `a` and `b` are allocated twice the slots in most virtual machines. This is a performance optimization and it is possible to naively resize by one on every insertion, but in many cases this over allocation is wasted memory for data structures that do not expand. Secondly, expansion requires expensive creation of larger internal structures and an extra copying cost to the larger structures. For example, when line 6 is executed, a larger internal list is created, copying items from the older list to the new one, as well as deleting the older list.

```

1 a = [1, 2, 3]
2 b = ["foo", "bar", "zar", "zoo", "zeh"]
3 if condition:
4     a.extend(b)
5 else:
6     a.append(4)

```

Listing 6.1: An Example of Data Structure Resizing

Presizing is a technique used to predict an optimal data structure size to avoid the overhead involved with resizing operations but can also avoid over-allocation of internal slots. Existing work uses allocation site-based techniques to achieve presizing [34], and canonical profiling [68], all of which do not account for the calling context and branches in the program.

Data structure resizing exists in both statically and dynamically typed languages for example, like Python lists, C++ vectors are a related comparison for statically typed language and have similar dynamism for some collections just like dynamic languages. The literature known to us does not mention presizing in these contexts. It is therefore possible to discuss presizing for static languages in a different context. To address the presizing limitations in existing work, we show that it is possible to predict an optimal data structure size by combining the *context-identifier* of an *allocation site*, using *call-site analysis* with *collection size* profile data created online or offline, thereby achieving allocation context-aware data structure presizing. We apply an offline version of this allocation context-aware presizing methodology to PyPy, observing as high as 30% performance improvement in the workloads we used. In the rest of this chapter, collection and data structure are used interchangeably. The novel allocation context-aware data structure presizing optimization proposed is designed to predict the *size* of a data structure, based on its *allocation calling context*. This is achieved through matching data structure sizes to their calling contexts from profiling information. The direct question we seek to answer is, *can we predict the size of a collection based on its allocation context?* We begin with a background on

presizing and motivation for our approach in the next section.

6.2 Optimal Presize Prediction Challenges

Presizing addresses the over-allocation and resizing overhead of data structures by hypothesizing that data structures from a given allocation site exhibit similar behaviour in regards to size. An observed size through allocation-site-based profiling can be used to compute the optimal size of a data structure. The key benefit for presizing is that it reduces and/or eliminates the overhead due to expansion and shrinking operations when a data structure resizes. The memory consumption is also significantly reduced because with optimal internal slot allocation, there is no need to allocate extra slots for memory that is likely not to be used. Optimal size prediction of data structures is believed to mostly be beneficial if used in hot sections of the code, and impacts large data structures [68].

The literature as noted earlier achieves presizing using allocation-site-based profiling, storing the information with garbage collection (GC) assisted approaches [34] by Clifford et al. The GC assisted approach was adopted with some modifications by Henning et al., for GraalPython due to the limited level of access to the GC [68] in this virtual machine. Presizing has been attempted in only these two papers known to us and in both, the technique is discussed with anecdotal evidence and evaluation is performed for one benchmark, *DeltaBlue*. Most significantly both implementations do not solve the key presizing challenges. Optimal data structure presize prediction in existing work does not support the following aspects, namely, branches and allocation calling context.

6.2.1 Branches

Setting sizes without accounting for the control flow of a program causes less optimal estimations, and leads to the same resizing overhead we seek to avoid with presizing. For example, in Listing 6.1, the optimal size of list `a` depends on whether the code follows the `if` path or the `else` path. Finding an optimal size with this instability is hard and remains an open question in all the existing work on presizing, because making the optimal size 8 could lead to allocation of 4 extra slots if the `else` path is taken, while making the size 4 assuming the `else` path, will lead to costly expansion operations if the `if` path is taken. A more accurate solution needs to account for the program path in addition to the list size, growth factor, frequency of growth and allocation site of a data structure.

6.2.2 Allocation Calling Context

Similar to the branching problem, the allocation calling context is part of the application call graph, important for profiling real-world applications. Allocation site profiling alone, as demonstrated in existing presizing work, is not sufficient for abstractions built on top of re-sizable data structures that provide a different interface as observed and pointed out in the GraalPython study [68]. This is because a high-level collection library for example, can be seen as having the same allocation site for data structure instantiation at different locations in the program. Profiling the allocation context of these allocation sites is therefore required to facilitate better accuracy in size estimation.

6.3 Context Aware Presizing

We use Listings 6.2 and 6.3 as a nontrivial example of collection resizing to motivate our work. Listing 6.2 defines several classes and a method, all of which use collections.

The main program Listing 6.3 contains the call sites of interest to our profiling. In Listing 6.3, allocation site based presizing will identify the five allocation sites 2, 3, 5, 6 and 13 but because sites 6 and 13 initialize the same abstraction, the allocation sites appear the same at both locations. Also presizing for allocation site 5 is difficult without understanding the context determined by the flag on line 7.

```

1 import collections
2 class AbstractCollection:
3     def allocate(self, num_slots):
4         self.slots = [None] * num_slots
5 class List(AbstractCollection):
6     def init(self):
7         super().allocate(16)
8 class Map(AbstractCollection):
9     def init(self):
10        super().allocate(2*16)
11 def initialize_map():
12    return collections.OrderedDict()

```

Listing 6.2: Collection Resizing Class and Method Definitions

For better accuracy, context aware profiling profiles call sites in addition to the allocation site. Figure 6.1 shows the calling contexts, denoted by arrows, for Listing 6.3, which are the sequence of calls associated with an allocation site and call site. We use $C_{@n}$ and $A_{@n}$ to represent call sites and allocation sites respectively.

In Figure 6.1, for example, the allocation on line 4 in Listing 6.2 can have two calling contexts, $main() \rightarrow newList() \rightarrow AbstractCollection()$ and $main() \rightarrow newMap() \rightarrow AbstractCollection()$. Similarly, `list_ex` has two contexts, $main() \rightarrow IF \rightarrow range() \rightarrow append()$ and $main() \rightarrow ELSE \rightarrow append()$.

Our approach does not store the whole call chain, we instead infer the context using the (call symbol, stack height) pair and allocation site, with the assumption that the stack height is unique for a given call path or chain of calls. The context identifier,

```

1 def main():
2     R = List()
3     K = Map()
4
5     list_ex = []
6     d1 = initialize_map()
7     if flag:
8         for i in range(0, 10000):
9             list_ex.append(i)
10    else:
11        list_ex.append(1)
12
13    d2 = initialize_map()
14
15 if __name__ == '__main__':
16    main()

```

Listing 6.3: Example Code Exercising Collections

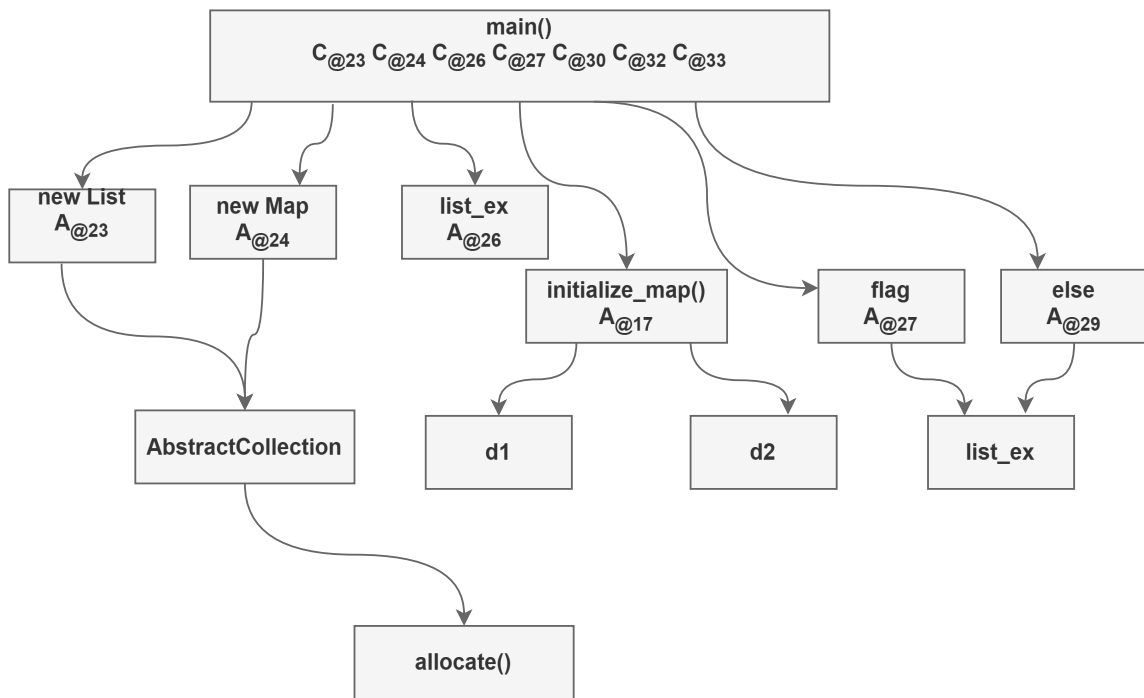


Figure 6.1: **Call Graph for Listing 6.3** – collections are indicated at the respective call sites. Arrows indicate the calling context which is a chain of calls with origins from the main method

stack height value for the current calling context and call site are computed from the stack pointer (SP) and program counter (PC). This way, even on the different paths for `list_ex`, for example, it is possible to observe a unique stack height for the different `append()` call sites.

This assumption is not always true as the a call path can have the same call site symbol and stack height, like lines 6 and 13, which can cause uncertainty. In these cases, we use existing disambiguation approaches to reduce the duplication of the mappings between call paths and the stack height. We discuss the three cases we are able to disambiguate in Section 6.4.3. For any cases we can not disambiguate, then our approach will not accurately perform presizing but as acknowledged in existing work [102] and our findings in Section 6.4.3, this assumption holds for most cases with disambiguation.

6.4 Approach Overview

Answering the presizing question stated at the start of this chapter is not straightforward, and like many optimizations, addressing every use case is aspirational. Figure 6.2 shows the techniques used to answer the question and the following novel profile-guided methodology is used to arrive at an ideal solution:

1. The first step is to gather profiling data about collection sizes, allocation sites and their allocation contexts. This can be done either offline or online.
2. The profiling data is initially stored in a log file but we provide some structure by storing it in a hash map, we call a context map, to guide the presizing optimization. We also install the allocation context in the object header, that is matched later. We discuss the impact in size to the object header in Section 6.5.1.

3. Before presizing, we access the object header and the map to match the size using the calling context as key.
4. Then the data structure is allocated slots using an exact size, estimated and read from the map to avoid waste.

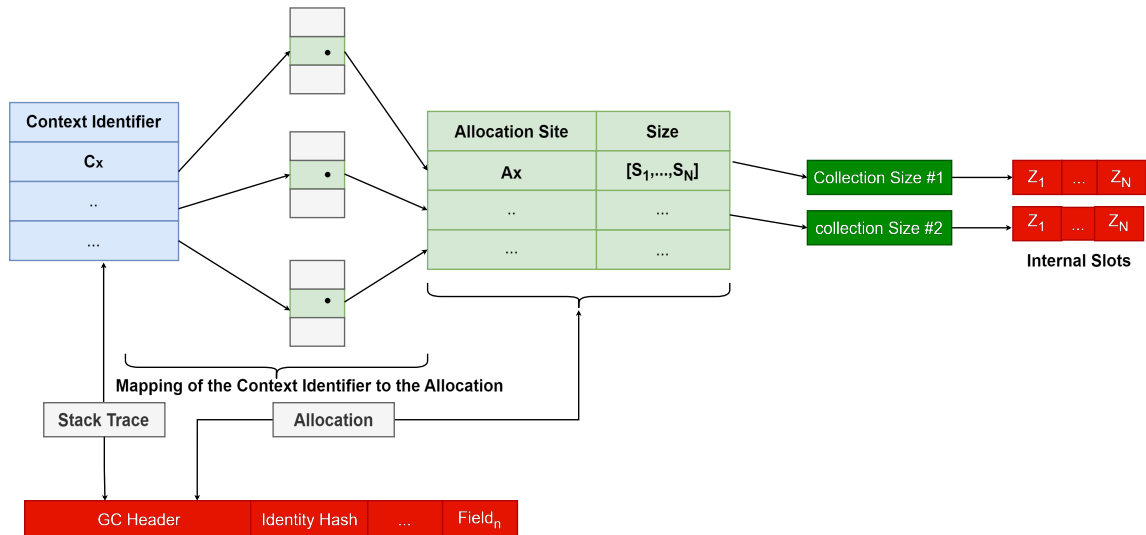


Figure 6.2: Context Aware Collection Size Profiling

We define the *size* of a data structure as the number of elements it contains. We estimate this size using profiling information that tracks the maximum, median and mean sizes of collections along with the number of times they grow and the growth factor when their allocated size is exhausted.

On the other hand, the allocation calling context is a tuple containing (1) the allocation site identifier, which is the allocation location in the source code for the collection object; and (2) the context identifier, which is a pair containing the symbolized name of the currently executing function and the stack height. The context identifier infers the stack state or trace during object allocation as discussed in the next section.

6.4.1 Object Size Profiling

The branching and calling context challenges discussed earlier are addressed by our approach by enhancing presizing to work with profiling data that is based on both allocation sites and the calling context. We infer the calling context by building on existing assumptions in the literature about the *context identifier* [94, 102]. A context identifier is a pair consisting of the *call site symbol* and *stack height* while a call path is the nested sequence of calls invoked at runtime.

We therefore complement allocation site profiling with call site analysis, and inferring the calling context through a context identifier that is based on the stack height. By using the stack height, we avoid the significant overhead of capturing the calling stack using instrumentation to track the stack epilogue and prologue. We therefore hypothesize that:

Knowing the context identifier characterized by the currently executing function, and stack height in bytes mostly identifies the calling context we need to precisely predict the size of data structures.

6.4.2 Inferring the Calling Context

Inferring the call chain using the stack height is discussed in the context of C/C++ in the literature with no evaluation for dynamic languages [102, 94]. Therefore, to use the stack height hypothesis for a dynamic language, we implemented a tool to profile Python applications, logging the program’s context identifier and call paths, first to prove the assumption, all done without annotating the program. For further profiling we do not include the call paths, we only use a context identifier that is based on the call site symbol and stack height.

Listing 6.4 shows an example of logs generated from profiling a hypothetical target program, while Table 6.1 shows profiling statistics. We capture the program log

number, file name, line number of a call site, the function name at the call site, the stack height, the call path and a tuple with information about any data structures associated with the call site. The tuple of data structure details contains the collection type, the variable name of the collection, the size and allocation site. Call sites that do not involve any data structure use, will not have the data structure fields in their log.

```

1, target.py:45, add, 4, <module> -> <module> -> add -> trace_function ,
  ([Tuple, add:4, 4])
2, target.py:6, tul, 5, <module> -> <module> -> add -> tul ->
  trace_function , ([List, tul_list:22, 4])
3, target.py:21, calculate_factorial, 6, <module> -> <module> -> add ->
  tul -> calculate_factorial
4, target.py:31, n_queens, 7, <module> -> <module> -> add -> tul ->
  calculate_factorial -> n_queens -> trace_function ,
  ([Tuple, my_tuple:35, 4], [List, item_list:36, 4])
5, target.py:7, mul, 5, <module> -> <module> -> add -> mul ->
  trace_function
6, target.py:46, sub, 4, <module> -> <module> -> sub -> trace_function ,
  ([Tuple, sub:11, 4])

```

Listing 6.4: Logs Generated from the Profiler

The stack height in this case refers to the distance between the current stack pointer and the top of stack in bytes. By combining the stack height with the current return address (i.e., local call site) and the data structure size, we are able to get a fingerprint of a particular stack trace. This corresponds to the current number of frames on the call stack in Python, calculated by inspecting code objects of a frame.

For the call paths, we build the Python interpreter with frame pointers, accessing the frames to read the calls in a file. To log collections in the body of a function call, the first challenge we encountered was that the `exec()` function produces the source code of a function in the form of text if we inspect the source lines of the frames

	stmts	stmts cov.	fns	fns cov.	calls
ai	98	6	3	3	6
call_simple	198	84	6	5	84
html5lib	59	5	5	5	5
nbody	143	8	4	3	8
pickle	350	9	6	4	9
regex_compile	92	5	2	2	5
richards	427	57	19	17	57
unpack_sequence	458	7	4	4	7
regex_effbot	147	47	4	4	47

Table 6.1: **Call Site Analysis for Python** – statistics generated from the profiling tool

instead of code objects, which makes it hard to identify any nodes of interest for our profiling. We instead through some form of static analysis parse the source code, generating an abstract syntax tree (AST), then walking the AST to identify list, set and dictionary nodes.

6.4.3 Accuracy of Context Identifiers

The logs generated are not completely accurate and thereby require disambiguation in some cases because of the lack of precision in mapping between context identifiers and call paths, i.e., multiple distinct call paths mapping to the same (function_name, stack height) pair. We used the methods of disambiguation discussed in the literature which include: *active record resizing (ARR)*, *call site wrapping (CSW)* and *function cloning (FC)* [102]. The following patterns of ambiguity are handled in our benchmarks:

$$X \rightarrow Y \rightarrow Z \qquad X \rightarrow W \rightarrow Z \qquad (6.1)$$

$$X \rightarrow Y \rightarrow Z \qquad Y \rightarrow X \rightarrow Z \qquad (6.2)$$

$$X \rightarrow Y \rightarrow X \rightarrow X \rightarrow Z \qquad X \rightarrow X \rightarrow Y \rightarrow X \rightarrow Z \qquad (6.3)$$

The first pattern involves two profile items with the same (function_name, stack height) but the call paths have a unique call, i.e., Y in the first call path and W in the other. To disambiguate this case, the intermediate operand of the instruction is modified, changing the function's active record size (ARR) to account for function's local variables on the stack, which changes the stack height of the path, making it precise.

The second pattern is harder, since the same (function_name, stack height) pair matches the same call paths, and active record resizing does not help with this. This case is managed by replacing a call on one of the edges with a wrapper function that then calls the original function using the call site wrapping technique. The wrapper function adds its own active record thereby changing the stack height.

The third case contains more duplicate calls and can not be handled by call site wrapping, instead it is handled by replacing a call to a duplicate function with a call to a copy of the function. The function clone contains disambiguation, in this example, clone $X \rightarrow X'$ so that X' wraps its call to Y. The additional disambiguation in the clone changes the stack height.

Figure 6.3 shows the degree of precision before and after disambiguation, for the three disambiguation methodologies. We use nine benchmarks from the standard PyPy benchmarks described in Table 7.1. The profiler uses these benchmarks as target programs and generates four logs in this experiment, one without disambiguation, one after applying only active record resizing, another after applying only function cloning and call site wrapping and the last log file is the one after applying all the three techniques. The hardware specifications for this experiment are detailed in Table 5.3. We use the geometric mean for all mean computations of the results.

Before disambiguation, we observe 43.3% precision by mean for the benchmarks we used. The highest level of precision is 60% seen in four benchmarks, namely *html5lib*, *regex_compile*, *unpack_sequence* and *regex_effbot*. The lowest level of precision is

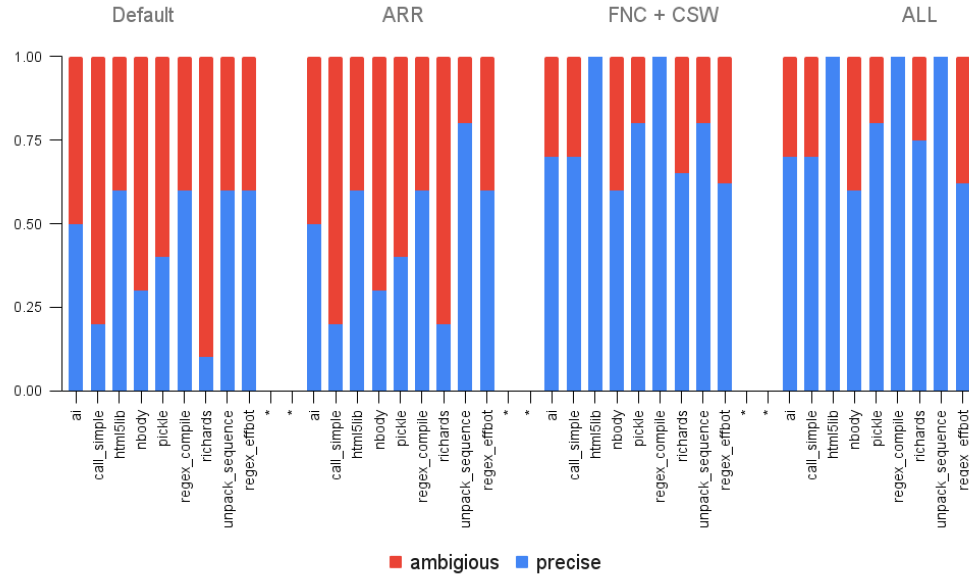


Figure 6.3: **Proving the Stack Height Hypothesis** – the profiler generates four logs in this experiment, one without disambiguation, one after applying only active record resizing, another after applying only function cloning and call site wrapping and the last log file is the one after applying all the three techniques. The configurations correspond to the disambiguation methods; *active record resizing (ARR)*: the intermediate operand of the instruction is modified, changing the function’s active record size; *call site wrapping (CSW)*: replacing a call on one of the edges with a wrapper function that then calls the original function; and *function cloning (FC)*: replacing a call to a duplicate function with a call to a copy of the function. We observe that the context identifier can uniquely identify a call path with a 79.9% accuracy level by mean.

observed for the *richards* and *call_simple* benchmarks, at a degree of 10% and 20% respectively. The general observation is, the more complex the workload, the higher the chances of ambiguity; as an example, the *richards* benchmark is more complex compared to the rest of the benchmarks.

Active record resizing was only beneficial to the *richards* and *unpack_sequence* benchmarks with improvements in precision of 10% and 20% respectively. Call site wrapping was only beneficial after function cloning, improving the precision of the rest of the benchmarks by 2%–100%. The benchmarks *html5lib* and *regex_compile* had 100% precision and the lowest degree of precision was 60% for the *nbody* benchmark

after full disambiguation.

After applying all the disambiguation techniques, we observe that by mean the context identifier (executing function and stack height) can uniquely identify a call path with a 79.9% accuracy level for the Python workloads we used. An 80% mean precision level justifies our use of the assumptions of the stack hypothesis in inferring the calling context for the presizing optimization.

6.5 Optimal Size Prediction

The profiling information gathered is stored in a globally accessible context map. The context identifier and the allocation site for each object are also installed in the object header to help with prediction (this is not needed if all prediction were to be done offline). In the RPython framework, the object layout is modified to store the allocation context. We store both the context identifier and the allocation site.

6.5.1 Installing the Context in the Object Header

The structure of the RPython object as shown in Figure 6.2 consists of one word for the GC header and vtable pointer, the identity hash field and the rest of the bits are used for static data in classes. In practice, there are a few spare bits on 64-bit machines, which we can use for the default *incminimark* garbage collector, and we mostly reuse some bits from the header to complement the spare bits. However, other times we use extra bits in the object header if we cannot reuse the hash field bits. We maintain one more bit as a flag to signal if profiling is set or not as shown in Figure 6.4.

Python objects are usually assigned a hash field in the CPython and earlier versions of PyPy interpreters, since these objects could be stored in a dictionary by the program. PyPy modified this feature later, to only create a hash field when the object's identity

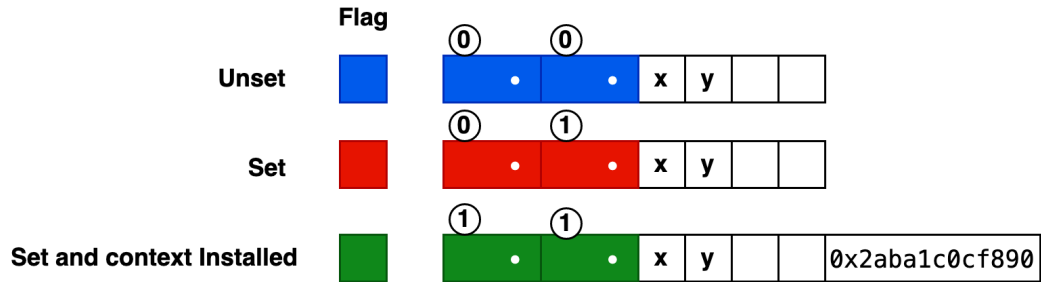


Figure 6.4: **Installing the Context in the RPython Object Header** – when a collection object is allocated, memory is allocated and reserved for the context identifier. The set and unset is for the hash field

hash is used, but once set, it can not be reset, so it is possible that some objects whose identity hash is not taken still have a hash field assigned to them.

We work on the assumption that all objects in Python are assigned a hash field as supported by the reference CPython implementation, and reuse the bits of the hash field to store the context identifier, if the object does not have its identity hash taken. This is a common case in Python where just a fraction of Python objects have their identity hash taken. When the object requires its hash field later, the profiling information is overwritten. This allows us to not enlarge the object header for most objects but for objects whose hash-field is taken, we use any spare bits and only enlarge the object header by a few bits, introducing a memory trade-off in those cases.

The garbage collector is thus modified upon allocation to install context data in the hash field if the object does not use its hash field. We maintain an extra flag as part of the GC header to signal if the object has profiling data installed. The new version of the object has its flag for profiling set, and is extended, allocating space for the profiling information that is to be written at the end of the object header as shown in Figure 6.4. Reusing bits allows us to not use more memory in the object header for most objects, but for dictionaries we may use some extra bits to store this

information.

6.5.2 Reading and Writing to the Context Map

Upon allocation, a collection object x 's data about the number of collections, allocation site, A_x , context identifier, C_x , and any observed sizes, S_N , are recorded in the context map as shown in Figure 6.2. The *context map* presented in Figure 6.2 contains the number of objects organized by allocation context and item size. For instance, if an object is allocated in the allocation context $C_x + t$ with size, S_0 , where t is a constant, the context map is updated to increment the number in the cell corresponding to row $C_x + t$ and column S_1 (one object more with size S_1). An allocation site can have several observed sizes for a given context, S_0, \dots, S_N .

6.5.3 Optimal Size Estimation

The optimal size of the collection is determined by reading the context map to access the maximum, mean or median size of the collection size that matches an object's calling context. We thus support three strategies, *max*, *mean* and *median* for the maximum, mean and median sizes respectively.

Max Strategy: This strategy uses the most recent maximum sizes of an allocation site to determine the presize of a collection. A threshold is maintained, P_{th} , where if the current presize is exceeded by this threshold, the new maximum is set as a presize. It suffers from over estimation should most collections end up being smaller. In the worst case, unused slots may be used but this strategy has the advantage of still reducing the overhead of expansion operations. In our analysis, we do not shrink the internal slots in case the collection is smaller, so there is no shrinking overhead.

Mean Strategy: Rather than naively taking the maximum value of observed sizes from the profiles, this strategy uses the geometric mean value of observed list sizes for an allocation site. The `presize` value is also likely to be high like in the max strategy, hence unused slots for smaller collections but the impact may be less. This strategy can suffer from the overhead of expansion operations for larger collections, but the impact is less than if there was no presizing at all. If anything, the performance impact of this strategy is closer to the max strategy.

Median Strategy: The `presize` using this strategy is computed from the median of the observed sizes for an allocation site. The `presize` is smaller than in the max and mean strategies hence fewer cases of unused slots in the best case scenario. However, due to underestimation, it is likely to suffer from the overhead of expansion operations for larger collections. The strategy produces the best results for most experiments discussed in Section 6.7.

6.6 Implementation

We modify the PyPy3.9 branch, which was the latest version at the time of implementation. The default RPython *incminimark* GC is modified to install profiling information on allocation, and to read from the global context map to `presize` data structures. The *incminimark* GC is a generational GC, so we naturally identify collection objects at nursery allocation when they are first allocated, setting a flag on these objects asking for the `id` or `identityhash`. This flag is different from the profiling flag which is set to signal if `presize` profiling is on or off.

The memory for these objects is initialized and reserved on object creation. The nursery objects with this flag are tracked and assigned destination locations during the closest nursery collection cycle. The *incminimark* GC supports resizing of arrays in the nursery so that they consume less memory when moved to tenure, but objects

flagged to use the `id` for our case are not resized with this feature, as we can lose pre-allocated memory in these objects.

When moving objects, we also first visit all the objects we marked as susceptible for our profiling and have the flag that signals using the `id` field, this traversal has an eminent cost quantified later in Section 6.7. We then copy over the objects while overwriting the relevant header fields with the allocation and context information. We access the global context map to allocate the optimal size. The hash value is not static, and changes the next time the object needs to move again. Our implementation relies on generational and moving collectors but it is possible to have an alternate implementation of our methodology in other contexts.

6.7 Evaluation

The evaluation discussion in this section has two main goals — we present results on execution time and peak memory usage — as a way of demonstrating the benefits of our technique. We also explore the allocation sites in the benchmarks used. Profiling has overhead, so we focus our evaluation on profile guided optimization, where we use information from a previous run, a use case that is useful for when the optimization can be used during deployment. There is overhead involved with reading the offline information to perform the actual profiling but it is not significant, we have left this overhead investigation for our future work when we have fully explored the online option.

6.7.1 Methodology

The default PyPy *Incminimark* incremental garbage collector is used and four presizing configurations are compared in this evaluation; the baseline is the main upstream PyPy branch without the presizing optimization; the *max*, *mean* and *median*

settings correspond to the heuristics for choosing the presize as either *maximum*, *mean* or *median* depending on the several sizes observed at an allocation site during profiling. Even in the baseline, we assume that all collection objects are assigned a hash field, if we used the default settings where the hash field is assigned to a few objects, then the results would be different. All benchmarks are run on a machine with specifications described in Table 5.3.

6.7.2 Workload Description

The official Python benchmark suite used in Section 6.4.1 was not sufficient for this evaluation. We instead use workloads recommended and open sourced for cross language benchmarking [97], to demonstrate the significance of the approach. The workloads described next are chosen because they use both a small and large number of collections compared to the official Python benchmark suite; of these, four benchmarks are macro while two of them are micro.

The *CD* (macro workload) benchmark is aimed at simulating airplane collision detection and was designed to evaluate real-time JVMs. We use the Python implementation of the workload with a setting of two planes and 5000 iterations. *DeltaBlue* (macro workload) is an algorithm that implements a constraint solver and is run for 300 iterations. *Havlak* (macro workload) contains a loop recognition algorithm with a number of collections that serves as a good use case for our approach, we run it for 20 iterations. The *JSON* (macro workload), *Permute* (micro workload) and *List* (micro workload) implement JSON string parsing, permutations and list operations respectively. These benchmarks are all run for 5000 iterations.

6.7.3 Observed Allocation Sites

Figure 6.5 shows a breakdown of allocation sites and the observed number of collections at those sites. To understand the plots in this figure, it is worth noting that the line

dividing the box into two represents the median value and shows that 50% of the data lies on the left hand side of the median value and 50% lies on the right hand side. The left and right edges of the box represent the lower and upper quartiles. The lower quartile shows the value at which the first 25% of the data falls up to. The upper quartile shows that 25% of the data lies to the right of the upper quartile value. The values at the ends of the horizontal lines are the upper and lower values of the data while single points on the diagram show any outliers.

The allocation sites correspond to line numbers in the source code. We show the maximum, mean and median values for the allocation site as well. We do not show the types of collections at the allocation sites, but for a breakdown, the benchmarks *DeltaBlue*, *JSON*, *Lists* and *Permute* only use lists. The *CD* benchmark uses only vectors. The *Havlak* benchmark uses a list on allocation site 318, a set on allocation site 160, a dictionary on allocation site 316 and vectors on allocation sites 315, 314 and 404.

The values without bars are mostly fixed size collections and smaller collection numbers especially for the *Lists* and *Permute* workloads. The macro benchmarks *CD*, *DeltaBlue*, *Havlak* and *JSON* contribute the highest number of collections since they are macro workloads. The different presize strategies generally benefit different workloads depending on the distribution of sizes observed at the allocation sites.

6.7.3.1 Discussion of Results

In terms of *execution time* as shown in Table 6.2, the max configuration achieves the best speed up in wall clock time of 5% for the *CD* workload followed by 5% and 1% for the mean setting and median strategies compared to the baseline. This is the only workload where choosing the maximum observed size at an allocation site has the best results in execution time. The *CD* workload is stable across call sites, mainly depending on the number of planes involved in the simulation. For the rest of the

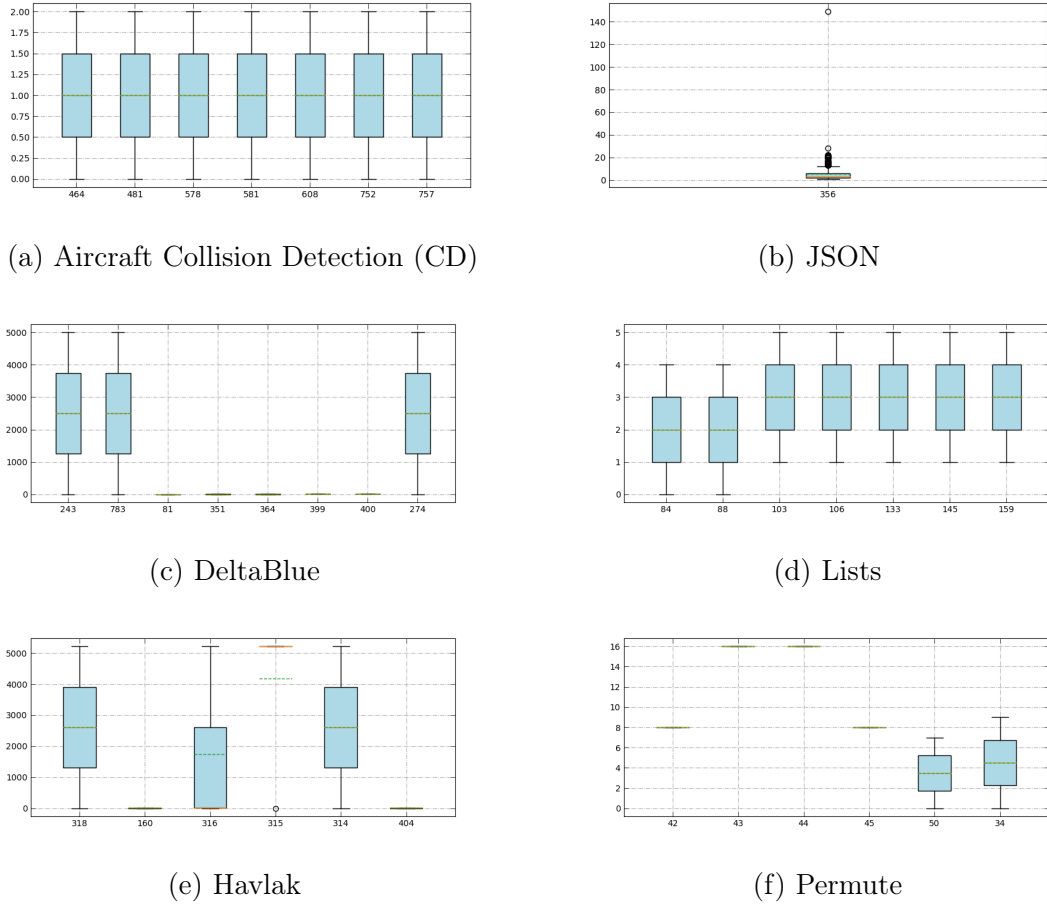


Figure 6.5: **Number of Observed List Sizes per Line Number of Allocation Sites** – the Y (collection count) and X (line number) axes show the size of collections and allocation sites respectively, while the orange and green lines show the mean and median

workloads we observe that the median setting works best due to more distribution and variation in the sizes observed at the allocation sites, which means using the median value is a good compromise. For instance the *JSON*, *Havlak*, *List* and *Permute* workloads have the best speed ups of 7%, 4%, 5% and 12% respectively compared to the baseline with the median strategy. The *DeltaBlue* benchmark experiences the highest gains of 30% overall also with the median strategy because it is collection intensive.

Memory-wise, when we reuse some header bits to store the allocation context data, our approach can reduce memory consumption due to a reduction in memory over-

	Strategy		
	Max	Mean	Median
Execution Speed			
collision detection	0.9563± 0.1745	0.9810± 0.1948	0.9810± 0.1948
json	0.9663± 0.4678	0.9662± 0.2042	0.9432± 0.3506
havlak	0.9832± 0.2376	0.9748± 0.1304	0.9791± 0.3161
deltablue	0.9076± 0.7650	0.9071± 0.6350	0.6522± 0.7650
lists	0.9867± 0.0132	0.9800± 0.0314	0.9728± 0.0171
permute	0.9779± 0.0470	0.9760± 0.0433	0.9760± 0.0433
min	0.9563	0.9662	0.9432
max	0.9837	0.9810	0.9810
geomean	0.9741	0.9758	0.9697
Peak Memory			
collision detection	0.9189± 0.1629	0.9029± 0.3025	0.9029± 0.3025
json	0.9726± 0.4665	0.9722± 0.4815	0.9459± 0.2387
havlak	0.9931± 0.5014	0.9844± 0.1167	0.9876± 0.1104
deltablue	0.9719± 0.7440	0.9432± 0.4668	0.9178± 0.7080
lists	0.9155± 0.2124	0.9099± 0.1990	0.8287± 0.1873
permute	0.9163± 0.3675	0.9016± 0.2689	0.8254± 0.2689
min	0.9155	0.9016	0.8254
max	0.9931	0.9844	0.9876
geomean	0.9480	0.9357	0.9016

Table 6.2: **Performance Gains and Memory Savings for Context Aware Presizing** – lower values are better, all results are shown as ratios normalized to the baselines and aggregated based on the geometric mean across 30 invocations. The error values correspond to the lower bounds of the geometric standard deviation. There is a 12% performance improvement and memory savings of 16% by mean for the best median strategy.

allocation from the presizing in objects. As shown in Table 6.2, the best memory savings are with the median strategy across all benchmarks. The memory results show the memory savings of the technique, without accounting for the header overhead. Profiling information does not have to be stored in the header, existing techniques like mementos can be used instead.

6.7.3.2 The Overhead of using the Profiles

Table 6.3 shows the overhead of accessing the object header and the context map while presizing; for workloads from the standard PyPy benchmark suite. We use standard benchmarks here with a mix of small and large collections. The goal is to observe the overhead of operations that access profiling data.

The numbers presented in Table 6.3 do not directly compare to the evaluation in the previous Section 6.7.3.1 due to the cost of instrumentation used but we also mostly consider installation of profiles in the object header an offline task in our evaluation. We present results for both reading and writing to the object header, as well as reading from the context map. We do not show the overhead of writing to the context map.

Our approach spends the most overhead on reading from the context map (maximum of 9.8%). Reading from the object header is the least costly (maximum overhead of 4.8%). The total overhead from all the three operations can be as high as 25% and 8% by mean. Due to our evaluation methodology, the overhead presented here may include overhead of the instrumentation used.

Our main contribution is the use of the stack height to capture the calling context required for presizing accuracy. The way the profiling information is stored can change. We do not therefore consider the overhead presented here as a bottleneck to the underlying technique as alternative approaches can be used instead of the object header and the context map.

Benchmark	Header			Map	All
	WR	RD	WRD	RD	Total
ai	1.0167± 0.2211	1.0136± 0.1000	1.0303± 0.4723	1.0104± 0.2906	1.0081± 1.6546
nbody	1.0038± 0.0650	1.0031± 0.07780	1.0069± 0.1906	1.0012± 0.1041	1.0080± 0.3032
pickle	1.0177± 0.4163	1.0099± 0.2603	1.0276± 0.7572	1.0092± 0.2667	1.0969± 1.9787
richards	1.0121± 0.6289	1.0036± 0.5972	1.0157± 0.8944	1.0045± 0.2422	1.0358± 2.7052
telco	1.0021± 1.1576	1.0079± 0.8124	1.0099± 2.0099	1.0046± 0.8124	1.0243± 5.4827
spectral	1.0531±0.2000	1.0485± 0.2000	1.1016± 0.4000	1.0508± 0.2000	1.2540± 1.1662
threading	1.0086± 0.0336	1.0047± 0.0383	1.0134± 0.0694	1.0066± 0.0233	1.0334± 0.1885
unpack	1.0447± 0.1527	1.0287± 0.1333	1.0735± 0.3055	1.0447± 0.1527	1.1917± 0.8307
pystone	1.0265± 0.0849	1.0153± 0.2956	1.0418± 0.3055	1.0153± 0.2872	1.0989± 0.8307
binary tree	1.0129± 0.0313	1.0088± 0.0274	1.0222± 0.0383	1.0100± 0.0172	1.0536± 0.0887
chaos	1.0116± 0.0077	1.0087± 0.0048	1.0203± 0.0097	1.0052± 0.0052	1.0514± 0.0289
delta	1.0182± 0.0359	1.0116± 0.0149	1.0298± 0.0389	1.0051± 0.0200	1.0647± 0.1231
float	1.0359± 0.0687	1.0175± 0.0111	1.0533± 0.0824	1.0350± 0.0784	1.1416± 0.2015
go	1.0180± 0.0027	1.0168± 0.0045	1.0349± 0.0064	1.0984± 0.2014	1.1682± 0.2049
meteor	1.0105± 0.1193	1.0104± 0.0986	1.0209± 0.1994	1.0196± 0.1107	1.0615± 0.6384
pidits	1.0107± 0.0103	1.0008± 0.0055	1.0105± 0.0015	1.0146± 0.0051	1.0354± 0.0155
callsimple	1.0319± 0.0051	1.0248± 0.0019	1.0567± 0.0117	1.0231± 0.0053	1.0909± 0.0468
min	1.0021	1.0008	1.0069	1.0012	1.0080
max	1.0531	1.0485	1.1016	1.0984	1.2540
geomean	1.0196	1.01664	1.0332	1.0208	1.0814

Table 6.3: **The Overhead of using the Profiles** – lower values are better, values presented as ratios relative to the PyPy baseline within the given error margins. The columns (WR) and (RD) refer to write and read respectively. (WRD) refers to both read and write. We also present the total overhead for all operations, all values are normalized to the baseline, aggregated across 30 invocations

6.8 The Call Stack and Garbage Collection

We extend the call stack analysis in Section 6.4 by running experiments with the goal of pinpointing the benefits of phase monitoring based on the stack height for further garbage collection related optimizations.

We start with identifying the stack-height-based phases in Python applications, investigating the relationship between stack height and the object live size and allocation rate, then close with a discussion of potential optimizations that can improve the garbage collection performance.

The literature has experiments on opportunistic garbage collection [143, 44] that is based on phases from the stack depth just like the stack height discussed in this section, but there is no mention of how the stack depth informed the design of the opportunistic garbage collectors in those papers. Kaleba et al. discuss phase-based optimizations for dynamic languages and propose monitoring the stack depth for better runtime GC performance as a potential extension of their work [83, 82].

The work in this section also considers the stack-depth-based GC optimizations including opportunistic GC, but first seeks to find a relationship between stack depth, live size and allocation rate.

6.8.1 Phase Analysis

The methodology in Section 6.4.1 is adopted to monitor the stack-height-based phases of the nine benchmarks in Figure 6.3. Figure 6.6 shows the phases of the Python applications.

The interesting trend to monitor is the point where programs move from a low stack height. The *CallSimple* and *Richards* benchmarks follow a consistent pattern where the stack height increases dramatically after call sites 45 and 23 respectively. The *AI* and *NBody* benchmarks also follow a similar pattern at call sites 7 and 7 respectively.

The presence of phases in some workloads motivates us to consider phase-based GC optimizations.

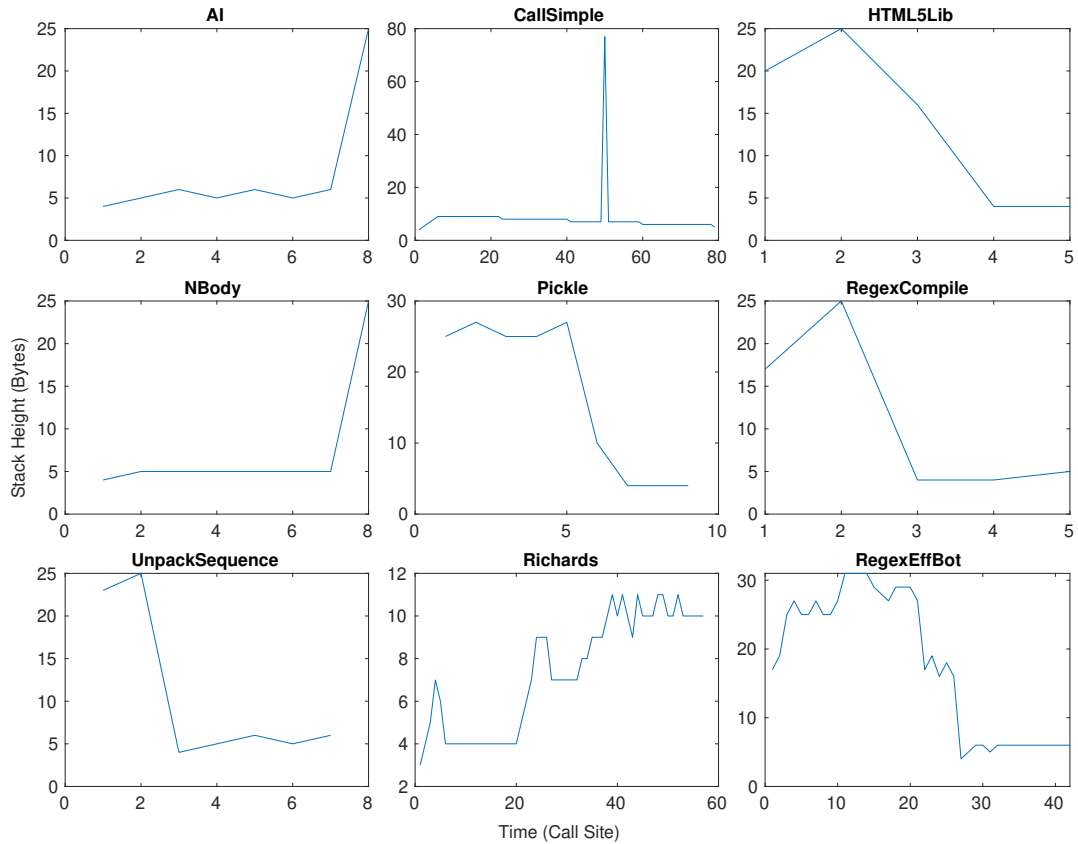


Figure 6.6: **Phase Monitoring for Python applications** — based on the stack height. A point on the X-axis maps to a call-site in the program

6.8.2 Live Size and Allocation Rate

To strategically trigger garbage collection for improved run-time performance, we need to do so at a low allocation rate and live memory size. Figure 6.7 shows the relationship between live size and allocation rate to guide any optimizations.

The stack height is measured with the same profiler discussed in Section 6.4.1. We use a counter to simulate a *heart beat* to log the allocation rate by exposing and invoking a GC function that is called at the call site of interest in the benchmark program. We similarly also expose a function to log the live memory size. The live size is computed after a garbage collection cycle.

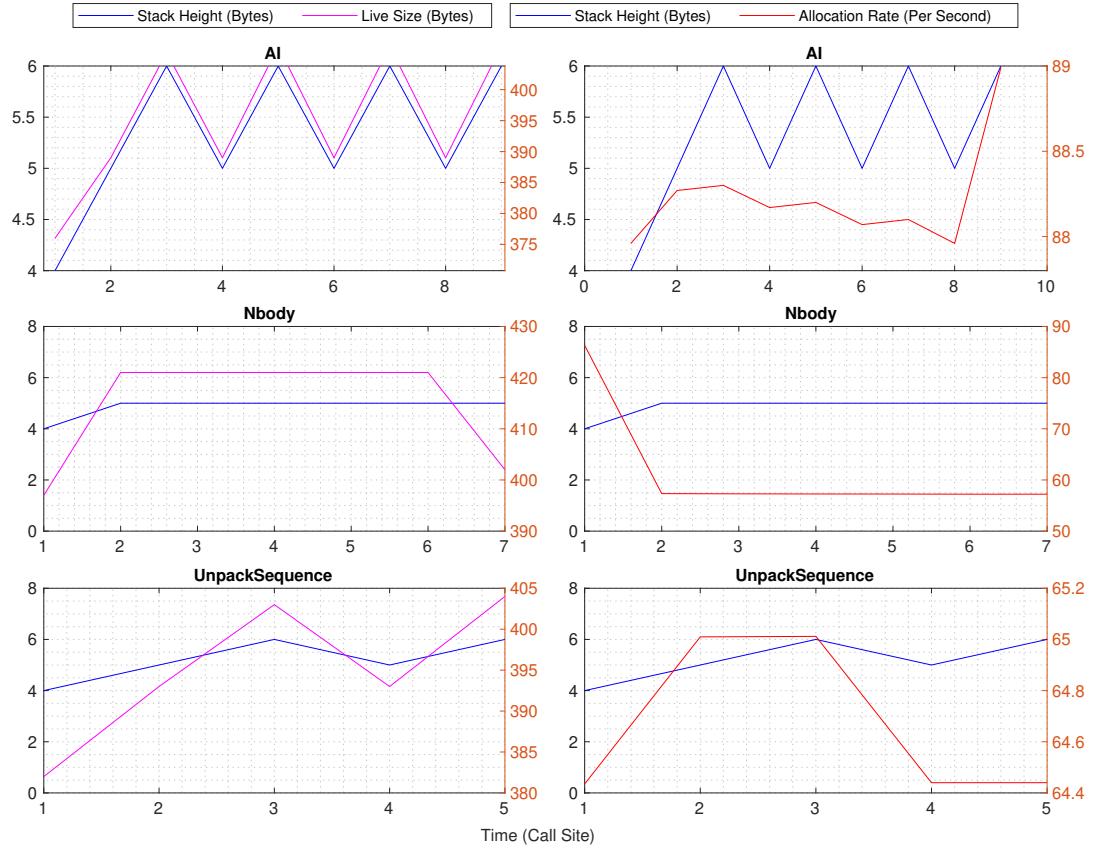


Figure 6.7: Stack Height vs. Live Memory Size and Allocation Rate

Preliminary experiments with 3 of the benchmarks above suggest a strong correlation between stack height and live size, but no clear correlation between stack height and allocation rate. Our key insight from this analysis is that optimizations that measure the live size to guide optimizations like the heap limit and opportunistic GC can leverage the stack depth as a proxy for the live size. This avoids the complexity around measuring the live size pointed out in the literature, which is that the live memory size can only be accurately measured after garbage collection [87, 132]. This is because the stack height value does not depend on garbage collection.

In relation to the allocation rate, triggering GC at the lowest stack height before transitioning to a high depth is not guaranteed to have impact due to the lack of correlation between the stack height and allocation rate. The GC can be triggered at a point where both allocation rate and stack height are low. We do not perform

the experiments for this opportunistic GC scenario because a productive study for this would require programs like a web application that are more realistic than the macro application benchmarks. Realistic workloads allow for a feasible observation of aspects like the pause times, cache miss rate etc, unlike the workloads used in this work.

6.9 Related Work

Clifford et al., use allocation-site-based profiling, storing the information with garbage collection assisted approaches in one paper [34] where a temporary object called a *memento* is allocated close to the object being profiled, created at the object's allocation site, and survives long enough just to survive the next GC cycle. The concept of mementos was demonstrated in V8 but has no further indication of being used upstream.

Mementos were adopted by Henning et al., with some modifications in GraalPython due to the limited level of access to the GC [68] in the runtime. We solve the limitations acknowledged in these papers by inferring the calling context based on the stack height in this work.

Chapter 7

Towards Correct Memory

Management for Python Native

Extensions

As introduced earlier in Section 2.6, mainstream languages often also provide foreign function interfaces but challenges on memory safety and correctness (not in the sense of formal verification) across the VM and C boundary remain prevalent. We identify the following main challenges to existing FFI design:

Pointer Stability To allocate memory passed into foreign functions, VMs/languages usually achieve this through foreign function calls. Each raw pointer exposed to C has to be carefully tracked to avoid double frees and to ensure pointer stability, so that each pointer remains valid while memory is in use. This burden is usually left for the programmer to invoke provided FFI calls that signal that an object is in use, a process that is error-prone.

Lifetime Complexity The level of detail a foreign API provides in terms of the heap object graph exposed to the VM introduces complexity in processing lifetimes.

Most FFIs expose much detail that makes it hard to create a feasible encapsulation that does not lead to different lifetimes for each object like structs in Rust. However, predictable lifetime semantics are essential aspects of a well-designed FFI.

Memory Model Compatibility Most times the memory model of the VM and a foreign language like C are different, for example, Rust’s type-based model versus C’s function-oriented model. Existing FFIs do not make an effort to ensure consistency for semantics like pointer aliasing, type conversion etc.; instead, programmers are forced to manually ensure this. For both Python and Rust the biggest source of safety incompatibilities happen with borrowed references, which are not easy to emulate into a C API and are the source of many unresolved bugs.

Generation and Verification Writing FFI bindings by hand is error-prone, therefore for any reliability, automation in the form of binding generation has to be thorough, or close to optimal. Existing binding generators are very inconsistent, forcing programmers to resort to manually written bindings that are error prone too and inconsistent. Also as a compromise, many efforts are made to limit the use of unsafe semantics for specific components of a program, like JIT compilers and operating systems. However, there are no tools to even verify that the safe parts of their code are indeed safe, except with local, manual knowledge by inspection of code blocks by domain experts.

The work in this chapter focuses on an improved implementation of a Python C API, with concepts that can be applied to FFI design for other languages.

7.1 Introduction

The Python C API provides an interface for C extensions to interact with and access the Python interpreter. The interface has C header files with functionality that gives

access to Python objects, invokes functions, performs garbage collection, and many other features [138]. This allows application developers to write C extensions for programs where performance is a primary goal of the application.

The API was designed to be as simple as making header files public and maintaining a way to dynamically load native extension modules. It remains one of the most powerful and largest C APIs, having supported and served popular native extensions like NumPy for over 20 years, with superior performance for the reference Python implementation, CPython. However, over the years, this same Python C API has grown to be a challenge for the evolution of the Python ecosystem as a whole. In fact C API authors for newer languages like Lua reference its design as an example of how not to implement a C API [101].

The main shortcomings in the current design of the CPython C API include: 1) enforcing a fixed-address object mechanism where objects can not be moved, which inhibits GCs that require reorganization; 2) exposing too many implementation details of the language; 3) enforcing a specific garbage collection policy in the API; and 4) supporting the concept of borrowed references, as implemented in Python, which has led to detrimental and unresolved bugs on object ownership. Section 7.2 discusses these challenges in detail.

For the CPython reference implementation, the API design is blocking significant optimizations to the language. The most noticeable has been the Gilectomy project [50], which aims to remove the global interpreter lock (GIL) as proposed by Hasting. Pitrou experimented with atomic increments and decrements for the C API, that are hidden behind macros, but this led to 30% performance degradation [50].

In terms of garbage collection, which is the focus of this work, as earlier noted, the current API dictates a fixed address object model where objects cannot be moved and a reference counting algorithm by design, hindering any evolution to tracing garbage collection. Reference counting has served Python well for years but its inability to

handle cycles still remains a concern. A hybrid tracing GC exists to alleviate this with moderate levels of success because supporting finalizers is still problematic [50]. Tracing garbage collection has shown to be more efficient than reference counting [81] for many applications and as such is an ideal GC algorithm for languages like Python but it is not used in CPython because support for it breaks many internals of the language and in particular, the C API.

In the Python ecosystem as a whole, alternative Python implementations like PyPy, Jython, IronPython etc., cannot efficiently support the Python C API due to its unfortunate design choices like exposing internal details of the interpreter and having an object model that these alternative implementations do not necessarily support [50]. Several solutions have been implemented by these alternative implementations but emulating the C API has always led to performance degradation [116, 135, 40].

To open up opportunities for new optimizations on garbage collection, like new GC policies, and to allow for an easier path for support by alternate implementations, a better solution would be to implement an improved C API and document a migration plan for existing extension libraries. An alternative Python C API, HPy exists as an experimental attempt to address the challenges of the current API [38]. At the core of HPy lies handles that point to the Python union type `PyObject`. This level of indirection allows for a flexible C API but, as discovered by research on handles by Kalibera and Jones, handles can make GC implementation harder, and done naively, management of handles can cause overhead and lead to fragmentation [84].

This study considers combining a *stack* and *light-weight handles* to redesign the Python C API to efficiently handle garbage collection and thereby simplify work for alternative implementations. Light-weight handles are handles that do not include the object header in the handle but because objects can move around in memory, the object has a pointer back to the handle. We further use object introspection for managing memory for FFIs like the Python C API with less programmer intervention.

We finally attempt to automate the migration of Python extensions to our new FFI. To achieve the above, we prototyped an experimental Python C API we call *CyStck* that uses a stack for communication from C to Python, and light-weight handles to an internal array, from Python to C, for each Python object to aid root access and tracking of referenced objects between the native and Python code. We used *liballocs* to attach additional GC metadata to objects to cater to corner cases of object lifetime enforcement where reachability alone is not enough to determine precise reclamation. To facilitate thorough investigation, we ported five popular Python extensions to CyStck, performing comparisons to both the current Python C API and HPy, for the time spent in Python code (Python time), time spent in interpreter and C extension module code (native time) and time spent in system routines (system time). We also analyzed the impact of migrating such large extensions to a new FFI by prototyping a tool that automatically ports Python C API extensions to CyStck.

We are the first, to the best of our knowledge, to redesign and robustly evaluate a new experimental C API for Python and provide automation for migration of existing extensions. We discuss two insights from our experience, 1) the use of even light-weight handles and complete garbage collection automation, will incur overhead especially when supporting reference counting semantics but also note that the potential gains from a moving GC outweigh these overheads, at least in an optimized implementation; and 2) the migration problem is not as complex as the Python 2 to 3 transition [103], since we demonstrate that through pattern matching and static analysis we can port most extensions automatically. The contributions of this work can therefore be summarized as follows:

1. We combine a stack and light-weight handles to prototype an experimental Python C API, we call CyStck. This design decouples GC implementation details from the API for cleaner evolution, aids in the automation of memory management and improves the object model to allow movement of objects to

support better performing moving garbage collectors.

2. We port five large, real-world Python extensions to this C API, thoroughly evaluating CyStck against the current CPython C API and HPy, profiling for both Python and native time. CyStck accelerates the execution of some of the benchmarks (KiwSolver) by 12% in native time, 13% in Python time. We observe at most 4× and 3× overhead in the other benchmarks, while introducing an acceptable overhead as low as 20% in system time for some benchmarks. CyStck also copies the fewest bytes 1%—40% between C and Python for all benchmarks.
3. We also use both pattern matching and static analysis to prototype a tool to automatically migrate extensions from the Python C API to CyStck. The technique behind this tool can apply to general migration of Python native extensions to any new C API, as well as between C API releases. We achieved a success rate of 60%—90% when we used this tool to migrate the five extensions described in Table 7.1.

7.2 The Python C API

Python exposes `python.h` for developers to access different aspects and details of the interpreter in *C*. Python objects are represented as a C struct called `PyObject`. When a block of memory is allocated on the heap, it is initialized and cast to `PyObject`. Listing 7.1 is an example of a function from the Python C API. The function takes two arguments, an object `v` and an attribute name `name`, both of type `PyObject`. The goal of this function is to retrieve the attribute `name` from object `v`. An attribute value of type `PyObject` is returned on success, and, `NULL` on failure.

The address of this object does not change throughout the object’s lifetime. `PyObject` points to an internal *base struct*. The address of `PyObject` can be freely passed to

```

1 PyObject * PyObject_GetAttr(PyObject *v, PyObject *name)
2 {
3     PyTypeObject *tp = Py_TYPE(v);
4     ....
5     PyObject* result = NULL;
6     if (tp->tp_getattro != NULL) {
7         result = (*tp->tp_getattro)(v, name);
8     }
9     else if (tp->tp_getattr != NULL) {
10        const char *name_str = PyUnicode_AsUTF8(name);
11        if (name_str == NULL) {...}
12        result = (*tp->tp_getattr)(v, (char *)name_str);
13    }
14    else {...}
15    if (result == NULL) {...}
16    return result;
17 }

```

Listing 7.1: A Python C API Function for Getting an Attribute of a PyObject

C code in different operations like storage in containers and so forth [135]. A new object is created using: `PyObject_New()`, so the initialization and allocation described above is done in this function. Due to reference counting, when an object reference is created or discarded, one has to increment and decrement the reference count with the following respectively:

```

Py_Incref(object);
Py-Decref(object);

```

These reference counting increment and decrement statements are used frequently in Python's C API and any code which uses the API. The C API was intended to have a simple design, but over time it became a maintenance burden as many third-party modules rely on it and any break to it breaks modules, which is also a compatibility burden. It is estimated [50] that for every Python release, some third party code base has to fix something due to changes introduced in the C API. This *thin layer* on top of the CPython internals and its compatibility constraints has blocked the implementation of several optimizations in CPython including a moving, tracing garbage collector (GC) due to several shortcomings. We discuss some of them that

are relevant to garbage collection below, namely, non-opaque `PyObject*` structs, a fixed-address object model, i.e., `PyObject`, tight coupling of a GC algorithm to the API and support for borrowed references.

7.2.1 Fixed-Address Object Model

As pointed out earlier, the C API relies on the idea that objects, i.e., `PyObject*`, have a fixed address and therefore do not move. To gain most of the benefits associated with tracing garbage collection, GCs such as the ones supported by PyPy assume that objects move in memory. Moving garbage collection algorithms are preferred to non-moving ones due to the ability to allow compaction and maintain heap partitions that require copying objects between the regions. Moving GCs are more efficient in both memory utilization and execution speed [81].

7.2.2 Non-Opaque PyObject Structs

The `PyObject*` object model is not completely opaque and exposes user-specific and many low-level concrete C struct details, some of which do not make sense to be exposed. Python implementations like PyPy hide these concrete struct details and thereby have an incompatible object model to the one used by Python C extensions. The public C API function `PyObject_GetBuffer()` in Listing 7.2, as an example, accesses and exposes `PyObject*` fields from its structure using the macro `Py_TYPE`. The details `tp_name` and `tp_as_buffer` from the `PyTypeObject` definition are exposed to the users of the API. PyPy for example has a different and opaque layout of its objects, which has made supporting this C API design difficult [135]. The ideal way is to make `PyTypeObject` completely opaque and providing any details through methods if required.

```

1 int PyObject_GetBuffer(PyObject *obj, Py_buffer *view, int flags)
2 {
3     PyBufferProcs *pb = Py_TYPE(obj)->tp_as_buffer;
4     if (pb == NULL || pb->bf_getbuffer == NULL) {.....}
5     return (*pb->bf_getbuffer)(obj, view, flags);
6 }

```

Listing 7.2: A Python C API Function for Getting an Attribute of a PyObject

7.2.3 Exposing GC Implementation Details

As noted earlier, other than exposing concrete `PyObject` struct details, the Python C API was also designed with implementation details specific to the reference implementation. One of the major ones is that the reference counting GC algorithm is assumed and enforced for garbage collection by the API dictated with the use of `Py_INCREF` and `Py_DECREF`. This is also a challenge to alternate implementations that use a different GC policy, which are forced to emulate reference counting for the C API.

7.2.4 Borrowed References

Some API functions like `PyList_GetItem()` do not `inccref` or `deccref` an object since the returned item is temporary and instead as an optimization use *borrowed references* to avoid calling `Py_INCREF` and `Py_DECREF`. When a function passes no ownership to its caller, the caller is said to *borrow a reference*. Therefore, a borrowed reference is a pointer that has a temporary reference. A borrowed reference becomes a dangling pointer when its associated object is destroyed since the freed memory may be used by a new object. Borrowed references have led to many unresolved bugs and crashes especially when there is no object reference to borrow in CPython and this behaviour has also proved difficult to emulate efficiently by alternate implementations like PyPy.

7.3 CyStck: A Stack-based C API for Python

Following the success of using a stack to aid garbage collection in previous work [76, 127] and the need to still provide compatibility of the Python C API by keeping the `PyObject` union type, we combine the use of a *stack* in FFI [75] and *handles* [38, 84], to prototype a memory management friendly API for Python, which we call *CyStck*. The stack and light-weight handles not only facilitate garbage collection but also act as a means of communication from C to Python and Python back to C.

7.3.1 Design

The current Python API uses a type, `PyObject`, that represents Python values in C. Any data passed to a C function is accessible using corresponding API functions. By design, `PyObject` is a struct, exposing details like the reference count, concrete type etc.

Figure 7.1 shows the simplified design of *CyStck*. The first change *CyStck* introduces to the current Python native extension API is the representation of Python objects by C programs. For backward compatibility, we avoid major changes to the CPython internals. Rather than completely eliminating the `PyObject` structure, we change its semantics by creating a handle, `Cystck_Object`, that points to the actual `PyObject` that lives in an internal array. These handles are light-weight which means that they point to the object, and any access to the object's header and fields is indirect. This is a trade off we make to allow us to reuse handle slots for several types of objects. The internal array is used to store all values that are passed to C functions from the VM. This representation allows us to use this indirection to track Python objects accessed by C, because the handles act as GC roots and permit the movement of objects without worrying about breaking C references. There is therefore an internal array for every C function invocation, so that when the function returns, the array is

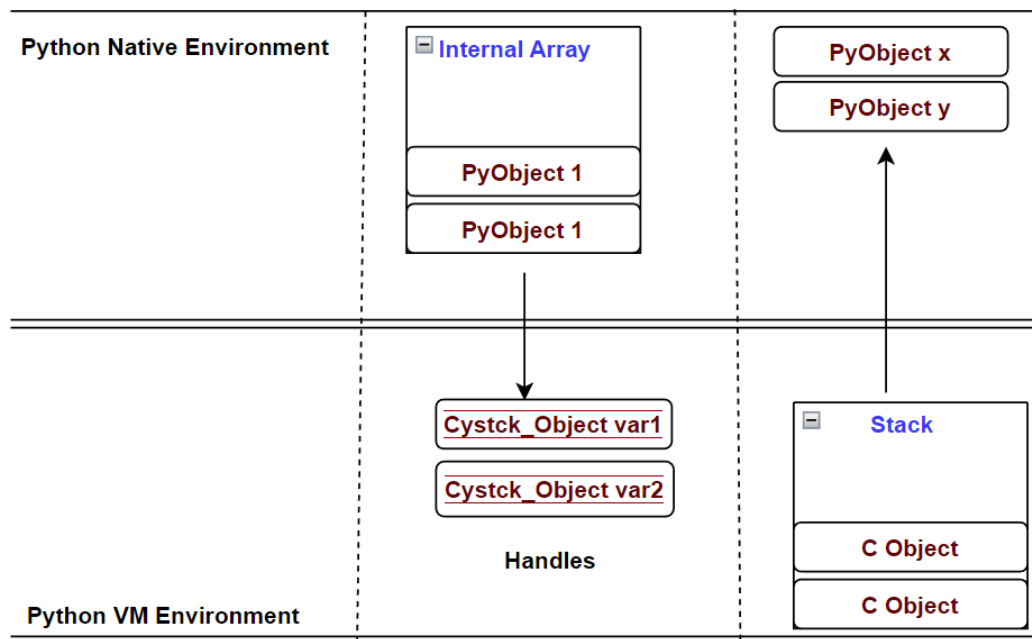


Figure 7.1: **The Design of the CyStck Prototype** — we show the two spaces that correspond to the native and the VM environments, including the stack and array data structures used to communicate between C and Python, described in Section 7.3.1

emptied. At this point, the Python values used by the C function can be collected by the garbage collector. The array contains arguments when Python calls C and return values when C calls Python. In practice when Python code imports and invokes a method of a C extension module, the parameters to the function are passed to C through the internal array. Section 7.4 has more details about this array and how it relates to garbage collection.

Other than creating an indirection to a `PyObject`, we also introduce a stack for communication from C to Python. A stack in `CyStck` is a data structure that is used to move values from C to Python. Therefore, every Python type like `String`, `Integer`, and `Float`, etc., has functions to push and retrieve values from the stack. Classes and collections, can be registered and passed as *userdata*, associated with a *metatable*. A metatable can be created with the class name and push member

functions, a constructor function has to be registered to return the class as userdata. The stack is also independent for a given C function invocation, which means that if a C function returns, all values are returned to Python as results of the function. The stack works in the opposite direction of the internal array. It contains arguments when C calls Python and return values when Python calls C. Similar to the array, as an example, when Python code imports and invokes a method of the C extension module, the C function and the return value is passed to the Python VM through the stack. The use of the stack then necessitates that we check for stack overflows. For this, developers should ensure that the stack does not overflow, an API function exists to check the status of the stack.

7.3.2 Implementation

CyStck is developed as an alternate and experimental implementation of the Python C API. The implementation can easily be upstreamed to the main CPython project. To appreciate the details of CyStck, Listing 7.4 shows a fictional example of a Python C extension that uses the new C API to extend Python while Listing 7.3 shows the equivalent extension using the Python C API. In both Listings, the module `c_module` implements the square of a number `num`.

```
1 PyObject square(PyObject *args)
2 {
3     double num;
4     PyArg_ParseTuple(args, "i", &num);
5     double result = num * num;
6     return result;
7 }
8 Py_METH_DEF(square_method, "square", Py_Squared,
9 Py_METH_VARARGS, "square a number" );
10 PyMethodDef *module_methods[] =
11 {
```

```

12     &square_method ,
13     NULL
14 };
15 struct PyStckModuleDef py_module =
16 {
17     PyModuleDef_HEAD_INIT,
18     "py_module" ,
19     "Module description" ,
20     -1,
21     module_methods
22 };
23 PyMODINIT_FUNC (py_module)
24 PyInit_module(Py_State *s)
25 {
26     return Mod_Create(s, &Py_module);
27 }

```

Listing 7.3: Example Extension Written With the Python C API

```

1 Cystck_Object square(Py_State *s, Cystck_Object *args)
2 {
3     double num;
4     CystckArg_ParseTuple(args, "i", &num);
5     double result = num * num;
6     Cystck_pushnumber(result);
7     return 1;
8 }
9 Cystck_METH_DEF(square_method, "square", Cystck_Squared,
10 Cystck_METH_VARARGS, "square a number" );
11 CyStckMethodDef *module_methods[] =
12 {
13     &square_method,
14     NULL
15 };

```

```

16 struct CyStckModuleDef cystck_module =
17 {
18     CyStckModuleDef_HEAD_INIT,
19     "cystck_module",
20     "Module description",
21     -1,
22     module_methods
23 };
24 CyMODINIT_FUNC (cystck_module)
25 CyStckInit_module(Py_State *s)
26 {
27     return Mod_Create(s, &Cystck_module);
28 }

```

Listing 7.4: Example Extension Written With CyStck

CyStck code has to include the `Cystck.h` header file exposed to access the features of the CyStck API instead of `Python.h`. As shown in Listing 7.4, Lines 1—8 implement the extension module. Instead of returning and expecting `PyObject` arguments, the extension returns a result and takes arguments of type `Cystck_Object`. As mentioned earlier, we do not eliminate the concept of `PyObject`, instead `Cystck_Object` is an indirection to the location of a `PyObject` at an integer index. `Cystck_Object` is therefore implemented as a pointer to the location in an internal data structure (array), and the location points to the `PyObject`:

```
typedef unsigned int Cystck_Object;
```

On Line 4, of Listing 7.4 still, we modify CPython's `PyArg_ParseTuple` to pass arguments to the C function using the internal array. The argument `args` is passed from Python to C in this array. It is then processed on Line 5 to get the square. The result, `result`, is passed back to Python by a push to the stack on Line 6 because communication from C to Python is through the stack. Line 7 then simply returns

the item/items on top of the stack, which is the value that this function returns. Lines 9—10 then register and initialize the extension implemented in Lines 1—8. On line 10 we introduce a macro with a caveat that it is an equivalent of a function and does some evaluation, and adds an extra slot to the extension we are registering on Line 22. This macro does data conversion, processing the function signatures. Then Lines 24—28 initialize and create the actual extension.

Most of the new C extension workflow is similar to the current Python extensions workflow except that we introduce new wrapper methods to manage the differences in semantics of returning and expecting the `Cystck_Object` type. The methods that change are `PyArg_ParseTuple`, and `PyModule_create` but also the properties of module methods have additional fields to manage the changes in the type of the result returned after invoking the C extension. The `PyState` argument, `s`, implements the supported Python state. Almost all API functions require an explicit Python state to be passed as first argument. CPython has gone through some cleanups to avoid global state, therefore to emulate Python state, API functions require passing explicit state as a first argument. The API also allows switching between Python states. This state is canonically a struct, and we capture state for the stack, array, exceptions, constants and types. Memory management for `CyStck` involves mainly emptying the internal array and stack when the C function returns, thereby removing any references to the objects, and proper handling of the reference technique. These processes are discussed next.

7.4 Garbage Collection

We support the default reference counting policy from CPython (the reference implementation) and provide a better path for moving garbage collection algorithms. Automatic memory management, when native extensions are involved, poses unique

constraints, most of which boil down to two main questions; 1) what is a root? and 2) how do we identify references among objects at the C and VM boundary? The most common design of language APIs like Python forces the programmer to consider the disparate memory management models between C and the VM when native code has any references to objects in the VM environment. Done naively, the native program may free objects referenced by the code in the VM or the VM may cause collection of objects prematurely.

In a typical garbage collection process for native extensions, it should be automatic but comes with much complexity, the biggest being the garbage collector does not manage or have knowledge of references from native programs or code. Yet correct collection dictates that a referenced object should never be freed. Native code has to therefore keep any referenced objects by signaling to the VM to avoid collection *when* moving objects across the boundary but also *how* to de-allocate the objects, by for example, explicit free.

7.4.1 Memory Reclamation

CPython uses a reference counting policy for managing memory, so CyStck provides garbage collection automation and functionality through the stack and an internal array without requiring the VM or native code to explicitly assist it in any way to find references between objects. We are able to achieve this because CyStck never returns explicit pointers to Python objects from native extensions. Instead Python objects are manipulated by processing an indirection, which is an index to the array, thereby allowing the Python VM to have complete knowledge of all objects referenced by C at any point in time.

The API typically tracks all `PyObject`s passed to native extensions, handling the reference counting semantics accordingly to avoid manually calling reference-counting functionality from the API. In a reference-counted garbage collector, a referenced

`PyObject` should have a reference count that is greater than zero to avoid its collection. This can be achieved by the reference-count functions in the `CyStck` C API for corner cases discussed later. Since `CyStck_Object` is an index to an array, when a Python object is in that array, its reference counts are incremented to avoid collection by the Python VM.

`CyStck` also provides for a finalization mechanism by providing an option for objects to be deallocated, by additionally invoking functions that relate to C code whose purpose is to perform finalization. This is useful for example in Python file I/O operations to close file descriptors. We are also able to configure an allocation routine to be executed by the Python VM for objects passed to C. The reference counting semantics in the Python VM, and handling of a cyclic object graph have to some extent limited the degree to which we could fully automate garbage collection but we believe that with a tracing GC, there can be an easier path.

7.4.2 Overflowing the Array

When an array overflows due to a C function using many values that were put onto the array, instead of running out of memory, the array functionality dictates a limit to the size of the array that tracks the `PyObject` objects. To work around this limitation, we provide functions in the C API that act as scope gates, to signal beginning and end of scope. This way objects in a given scope are erased from the array when out of scope creating room for more objects that are still in scope. These scope gates are introduced and managed by the programmer by calling the API functions and the programmer has to also make sure the stack does not overflow. In our experience, this scope management is only necessary for a few programs and is a corner case for C functions that generate many objects that correspond to many values in the internal array. The scenarios are extremely rare and most programs will work without this much programmer intervention. In particular, we did not need to apply any scope

gates in the large extensions we ported in Section 7.5.

7.4.3 Object Lifetime

Every C function invocation has a corresponding array and the array is emptied, decrementing the reference counts of the objects in it, when the function returns. In the case of tracing and moving garbage collectors, objects can be made to move in memory without worrying about losing track of said objects. The `PyObject` lifetime is therefore tied to the existence of the C function that created or uses it and is guaranteed not to be collected as long as the C function is alive. This lifetime management makes it impossible to access pointers or references from the Python VM for C structs and variables, or global variables.

This definition and processing of lifetimes is mostly correct and works for most cases but sometimes a `PyObject` can out-live the C function invocation that created it. In this case the API provides functions that return a reference to the object (`CyStck_ref`) and destroys a reference (`CyStck_unref`). These functions also implicitly update the reference counts accordingly. This reference technique is also used to process weak references. The Python VM or code has no access and can not modify these references or the values in the corresponding locations, only native code can erase these objects. This reference is therefore a light-weight handle that associates and disassociates the object making its location available or free for use.

We also explored the use of third-party object lifetime mediation using a tool called `liballocs`. Through its process inspection and knowledge of allocators, we are able to attach lifetime policies to objects. To complement the manual reference mechanism described earlier, we use the policy meta-data feature to determine when to reclaim memory. Algorithm 2 shows how object lifetime policies are attached while the algorithm for determining when to free object memory is detailed in Algorithm 3. In Algorithm 2, we use the `liballocs` API to determine when an object is allocated

using `malloc()` on Line 4. Liballocs only supports metadata policy mediation for `malloc()`-allocated objects [11]. Any object regardless of type (Python or native) is assigned a policy depending on where it was created. If it was created in Python, then only the reference counting policy applies to it (Lines 6–8). In contrast an object not created in Python is attached an explicit free-policy (Lines 9–12).

Algorithm 2: Allocation: `ObjectLifeTimeAnalysis(obj)`

Data: Input: Let `obj` be the object
Result: An accurate deallocation of `obj`

```

1 use liballocs.h;
2 initialization;
3 obj_allocator = alloc_get_allocator(obj);
4 if obj_allocator == malloc() then
5     /*Attach meta-data*/;
6     if PassedToC(obj) and CreatedFromPython(obj) then
7         attachRefCountPolicy();
8     end
9     /*Any object created by native code*/;
10    if PassedToC(obj) and !CreatedFromPython(obj) then
11        attachRefCountPolicy();
12        attachExplicitFreePolicy();
13    end
14 end

```

During reclamation of an object, as shown in Algorithm 3, objects created in the Python environment will be freed when their reference count is zero, i.e., only reachability determines when it can be freed and it is freed automatically (Lines 3–8). However, for an object that is not created from native code, it can not be freed implicitly, instead we have to check for a call to `free()`, which is a state maintained on the object, before releasing the memory (Lines 9–19). By attaching meta-data to objects, liballocs is able to avoid attempts to free an object unless any attached policies are consistent, including which allocator and reference count, delaying the deallocation until a convenient time. Any object allocated and passed to Python has reference counting and reclamation information. If reclamation is triggered before liballocs has removed any reference-counts, memory is not freed until this is true.

This *policy mediation* model described from liballocs works on the theory that native programs, or a VM, can not accurately know how or when the object memory is ready to be freed.

Algorithm 3: Deallocation: ObjectLifeTimeAnalysis(obj)

Data: Input: Let obj be the object
Result: An accurate deallocation of obj

```

1 use liballocs.h;
2 /*deallocate an object*/;
3 if CreatedFromPython(obj) then
4     if refcount == 0 then
5         detachRefCountPolicy();
6         free(obj());
7     end
8 end
9 if !CreatedFromPython(obj) then
10    if refcount == 0 then
11        detachRefCountPolicy(obj);
12    end
13    if isExplicitFreeCalled() then
14        detachExplicitFreePolicy(obj);
15    end
16    if !has_policy(obj) then
17        free(obj);
18    end
19 end

```

7.5 Evaluation

The goal of our evaluation is to gain insight into how our newly prototyped Python C API compares first to the current Python C API, but also to another alternate Python C API implementation called HPy.

7.5.1 Methodology

We ported five real-world and relatively large Python native extensions to use CyStck; these extensions are described in Table 7.1. The five substantial and widely used

modules, NumPy, Matplotlib, KiwiSolver, PicoNumPy and UltraJson are huge extension module libraries that required intensive porting from the Python C API to CyStck. Table 7.1 has the total PyPI number of downloads and lines of code (LOC) of these projects to appreciate the potential complexity of porting them to any new API.

In addition, we have implemented benchmarks for these extensions to aid with the experiments. The benchmarks are numerical computing tasks exercising solvers and array manipulation. We chose these extensions because they are popular enough in the Python community, demonstrated by their high number of downloads from the Python Package Index repository, but also because they have been ported to HPy [38], which helps with our comparisons.

All benchmarks are run on an Intel(R) Xeon(R) Gold 6248 CPU @ 2.50GHz machine, running 64-bit Debian 11.0.0 with GCC 10.2.1, using 10 cores. We use the Scalene [10] Python profiler to measure the time spent by the benchmarks in Python and native code and also the system time. We also measure the rate of objects copied across the C and Python boundary and also in both the Python and native space respectively. Scalene is a sampling profiler that uses signal delivery to estimate the execution time. We run the benchmark programs 15 times, ignoring the first 5 runs. This is not aimed at determining accurate steady-state because it is impractical, but rather a large number of iterations increases the likelihood of getting close to steady-state, and averaging the last 10 runs to come up with the evaluation insight presented in this section.

Figure 7.2 shows the memory used in terms of rate of objects copied for the five extensions. The results in Table 7.2 correspond to actual Python, native and system times calculated from the percentage results the Scalene profiler generates, in respect to the total execution time. We therefore discuss the results shown in Table 7.2 for each of these metrics next, starting with time spent in the Python VM.

Extension	Description	LOC	PyPI Downloads	Benchmark
NumPy	A package for scientific computing [4]	861,693	3,902,582,787	Laplace
Matplotlib	A plotting library [134]	244,459	990,938,563	animate_decay
Kiwi	The Cassowary constraint solver [133]	13,171	750,403	Solver

Table 7.1: **Description of Python Native Extensions Ported to CyStck** – their size of source lines of code (LOC), total number of PyPI downloads. An associated benchmark used for the evaluation is also indicated

	Native Time (seconds)		System Time (seconds)		Python Time (seconds)		Total Time (seconds)	
	C API	HPy	C API	HPy	C API	HPy	C API	HPy
UltraJson	0.35	0.32	0.35	1.28	3.91	14.4	11.6	16
PicoNumPy	0.03	0.02	0.03	0.06	0.76	0.21	0.28	0.29
NumPy	0.26	0.24	0.03	0.03	0.27	0.17	0.51	0.44
Matplotlib	8.42	9.18	0.34	4.45	6.82	14.19	17.18	27.82
Kiwisolver	0.95	1.29	22.27	23.21	27.80	7.74	31.81	32.24

Table 7.2: **Unnormalized Results** — for the native system Python and total time relative to execution time

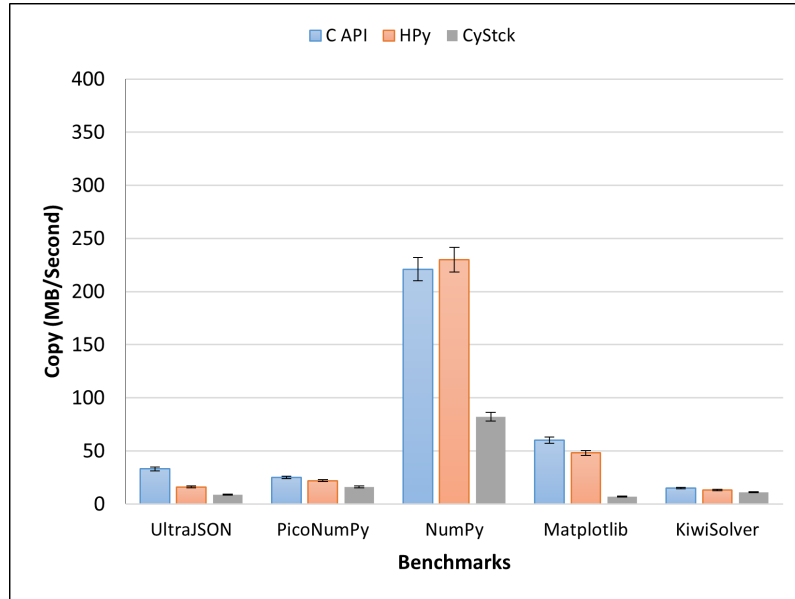


Figure 7.2: **Unnormalized Rate of Bytes** — copied generally in Python code, native code but also across the Python/C boundary

7.5.2 Discussion of Results

Python Time: To appreciate what Python time is, it is worth noting that the extensions themselves are written in C but they are built, generating a Python module that is importable in any Python script. We therefore implement a Python benchmark that uses the extension module for each of the extensions as described in Table 7.1. The *Python time* shown in Table 7.2 is the amount of time spent in the Python layer during the operations that cross between Python and C. As shown in Table 7.2 CyStck is faster by 12% and 2% compared to the C API and HPy respectively for KiwiSolver while CyStck introduces an overhead of over 90% and 50% compared to the C API and HPy respectively for the UltraJson benchmark. The same is true for NumPy with CyStck being 150% and 190% slower compared to HPy and the Python C API in actual running times. For Matplotlib, by mean CyStck is 120% slower than all the APIs we compared against for these benchmarks. Likewise CyStck is slower for the PicoNumPy benchmark, registering an overhead of about 6.9X compared to both the C API and HPy in the total running time. The

major difference here is how we communicate between Python and C, CyStck uses an internal array, while HPy and the C API use a tuple. The tuple for the C API and HPy is for only passing data, while the array in CyStck serves other purposes like memory management as well as passing data. We do the same validation checks on the data but we can attribute better or worse results to other implementation details of CyStck. The overhead we see in this category for most of the benchmarks can also be related to several things, not necessarily the CyStck implementation details but more so the application code. Python code, depending on how it is written, can run faster or slower, and optimization on this part is not out of control to the developer. This optimization is not specific to CyStck but applies Python code that calls any C API implementation. The best results are from the longest running benchmarks and even where we had overhead, it was less for longer running benchmarks.

Native Time: As pointed out earlier, Python time is the time spent in the Python VM; native time, on the other hand, is time spent in the C code. The unnormalized native time results from running the benchmarks are shown in Table 7.2. In actual unnormalized times, CyStck is faster for KiwiSolver by 24% and 44% while it is slower for UltraJson by about 50% and 70% compared to the C API and HPy. PicoNumPy is equally consistent in overhead, with a slow down of about 300% and 500% respectively. CyStck has an overhead of 50% for NumPy compared to both the C API and HPy while for Matplotlib, it is a slow down of 12.5% compared to the C API and 3.1% compared to HPy.

Native speed is significant in our experiments because this is the part of code that is immutable and out of control of the programmer, and hence cannot be optimized as it consists of interpreter code, external libraries and extension modules. What can cause poor performance is poor garbage collection in the C code but since CyStck is less hands-on, any manual steps to manage memory are not complex for programmers.

We noticed that manually calling reference counting improved performance for some extensions, which means that garbage collection for CyStck needs some help and improvement to an extent. Also the memory management automation through liballocs introduces some overhead and it is worsened with compromises required for reference counting to work. An optimized implementation with tracing GC is likely to have better performance.

System Time: Other than Python and native time, the rest of the time is spent in system time, also shown in Table 7.2. This is the part that contributes to the most CyStck overhead consistently for all the benchmarks we ran. CyStck costs more than 12% compared to HPy and the Python C API for UltraJson while for PicoNumPy, it costs more than 26% overhead. Also, for NumPy, CyStck is slower by about 50% compared to HPy and the Python C API.

Similarly for Matplotlib and KiwiSolver, CyStck runs more than 16% slower for the former, and more than 7% slower compared to HPy and CyStck respectively. Overhead in the system can also be attributed to several aspects, like I/O bottlenecks, so we cannot attribute all the overhead to just implementation details, although we are not able to reproduce a correlation for this. Extension module code contributes to system code, that is our observation from the evaluation, hence why it is different across the different C APIs that we evaluated.

Total Time: This metric is a geometric mean measurement of the wall-clock time measured 10 times. Because of the overhead seen through the Python, native and system time, CyStck introduces a slow down as shown in Table 7.2. The overhead is between 10% - 600% compared to HPy and 12% - 200% compared to the Python C API. Matplotlib is an outlier with about 30% overhead. A little overhead is acceptable when working with handles, with light-weight handles, it is less overhead. We also automate memory management tasks which can contribute to the overhead. Since

this is our first prototype of CyStck, there is therefore room for improvement in optimizing the handles. As a bigger picture, the ability to move objects unlocks better GC policies with improved throughput, which is likely to improve performance. This CyStck FFI therefore provides immediate memory safety benefits from automating memory management and a path to better performance in the future.

Bytes Copied: Shown in Figure 7.2, this metric indicates the rate of bytes copied generally in either the Python or native environment but also across the Python and C boundary. This is important to know when copying arrays as either Python or NumPy arrays for example. The metric helps us determine how much space is being consumed but also since allocation and deallocation of objects happens after the copying of bytes, large volumes can affect the performance of the application negatively.

CyStck dominates with the fewest bytes copied per second over the boundary for all benchmarks. KiwiSolver has the least margins for CyStck, the difference being just about 1% and 2% compared to HPy and the Python C API. NumPy has the highest margins still in favour of CyStck, specifically about 40% compared to the other C APIs in question for our experiments. UltraJson, PicoNumPy and Matplotlib have the most mean margins. For UltraJson, the difference is about 20%, about 12% for Matplotlib and about 4% for PicoNumPy. Due to object introspection supported by CyStck and not in the other APIs we compare against, we still see overhead in general despite these impressive results for bytes copied, but copying a lower number of bytes contributes to better performance.

7.6 Migration of Extensions

Releasing a completely new Python C API to the Python ecosystem breaks backward compatibility and affects many projects that require huge efforts in terms of person

hours to rewrite the large code bases to match the new API. Python has gone through a similar transition from Python 2 to Python 3 [103]. However as we discuss, migration automation of existing Python C extensions is less complex than the automation required for the Python 2 to 3 transition. Migration of code between releases of a language involves mapping patterns of any syntax and semantics of the previous version to the target version.

Building on *pythoncapi-compat* [129], we implemented a tool that converts the implementation of a Python C extension to an extension implemented with CyStck. The rules can easily be updated and generalized for any other C API. There are recent advancements towards large language models like CoPilot to do similar transformations but our method is cheaper and has better precision due to the specific rules that address the context required for the transformations from one C API to another.

7.6.1 Methodology

The tool makes use of pattern matching techniques of mapping strings [88, 6, 109] and regular expressions, with non-trivial static analysis. In other words, the source code is viewed as a sequence of strings, and regular expressions are used to map certain patterns in the programs, and making transformations in the system to match the syntax and semantics of the new C API, as shown in Figure 7.3. Since our problem is not as complex as transforming a program from one programming language to another, we did not find the need to extensively use higher abstractions for representing the source code in the form of abstract syntax trees or parse trees.

Patterns are matched from this stream of strings, invoking transformation routines to convert the code to use the CyStck API. The methodology for the source transformation can be summarized as follows:

1. We perform some preprocessing steps, removing any features in the program that need not to be matched and transformed, like comments, extra space, etc.

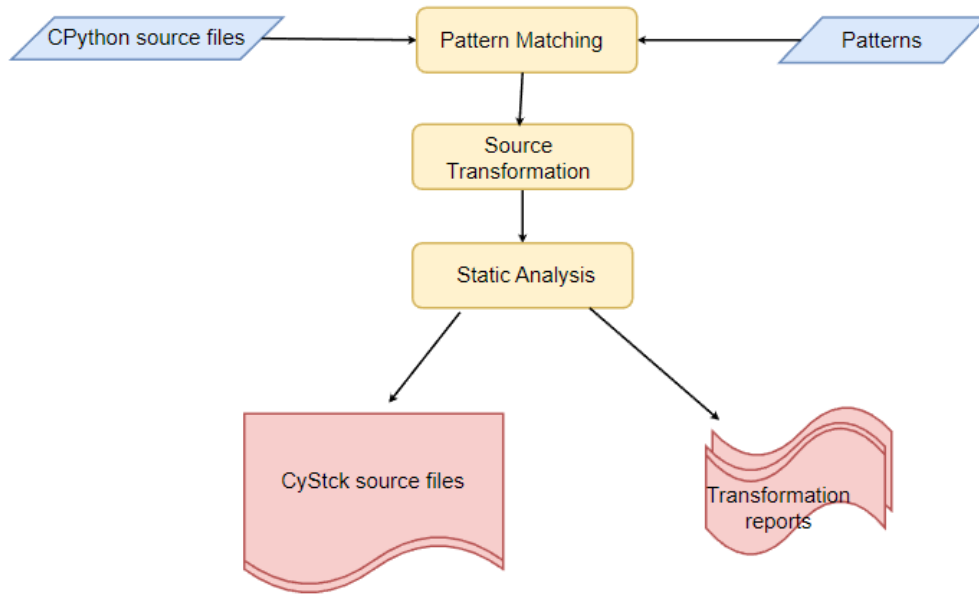


Figure 7.3: The System Flow for the Migration Tool

2. We identify certain patterns that cover the features we want to transform.
3. After identifying the patterns that need transformation, we invoke a routine to replace the matched patterns with the required CyStck-equivalent syntax and semantics.
4. We perform static analysis to check and infer the types for custom types, but also warn for other errors in the generated code.
5. We record and summarize the transformation operations performed. We generate diff information between the modified file and the file before modification.

Source Pattern Matching

When searching for syntax and semantic matches in the source code, there are three major kinds of statements that correspond and relate to the major changes CyStck introduces; `PyObject` type, API function calls and definitions, and return statements. To identify the pattern matches for the `PyObject` type, we can naively just search for

the phrase `PyObject`. Depending on semantics, this is too naive and not sufficient to inform transformation precision. We therefore identify the following patterns related to the `PyObject` type:

$$\textit{Var Declaration} \qquad \textit{PyObject Ident1}; \qquad (7.1)$$

$$\textit{Var Assignment} \qquad \textit{PyObject Ident1} = \textit{Identz}; \qquad (7.2)$$

$$\textit{Func Definition} \qquad \textit{PyObject Ident1}(\dots)\{\textit{@*}\} \qquad (7.3)$$

$$\textit{Func Signature} \qquad (\textit{PyObject Ident1}, \textit{PyObect Ident2}) \qquad (7.4)$$

These rules are variations in the formation of patterns where the `PyObject` substring can be found. Rules 7.1 and 7.2 relate to variable declaration and assignment while rules 7.3 and 7.4 describe function definition and signatures respectively. During transformation, rule 7.1 is transformed first in terms of precedence, before the rest. To avoid name collision and function pollution, C API functions for most languages follow a naming convention, `@*`, where `@` is the language abbreviation. Python C API functions follow the convention `Py*`. `CyStck` re-implements these functions to take care of state, type and stack design but also keeps the naming convention to `CyStck*`. The following rule matches the API functions:

$$\textit{C API Functions} \qquad \textit{Py@*}(\dots); \qquad (7.5)$$

This pattern applies to transformations that handle function invocations like `PyArg_ParseTupleAndKeywords` but also functions like `PyInit_c_module`, structs like `PyModuleDef` and macros like `PyMODINIT_FUNC` that are defined as part of the extension module. Lastly, since `CyStck` changes how values are returned, the `return`

statement is also matched using the following two rules:

Function $CyStckObject\ Ident1(Identx, Identity, \dots)\{@\ast\}$ (7.6)

Return Statement $return\ result;$ (7.7)

The first rule 7.6, checks to verify that the function returns a `CyStck_Object`, which is the right type, before changing the return semantics. This is because if it is a normal C function returning an `int` for example, we do not have to query the stack for the value that has to be returned by the function denoted by rule 7.7. In terms of precedence, all transformations that modify the `PyObject` type should have been applied for this rule to be accurate.

Source Transformation

For large code bases, the main aim of this tool is to make migration of extension modules to a new API a feasible alternative, with strong syntactical guarantees. Semantic guarantees are harder and thorough testing is required to verify the correctness of the generated code. For example, the tool is not yet equipped to automate this, only a human can make some of these judgements but with some advanced rules in our algorithms, we think future iterations can achieve this accuracy with `ref/unref` annotation. We achieve code transformations through a specification using *rules*, in the following format:

$[pattern : action]$ (7.8)

The first part of a rule specification is a pattern in the code. The second corresponds to the steps or procedures that must be taken when the first part matches. For example, when performing the transformation for a block of code, the operation that manages the conversion of a `Statement` is invoked. A matching procedure isolates the

Statement kind to be transformed and the relevant conversion function is invoked. As an example, consider the following hypothetical Python C API implementation of a C extension method:

```
1 PyObject *square(PyObject *args)
2 {
3     double num;
4     PyArg_ParseTuple(args, "O", &num);
5     double fact = num * num;
6     PyObject * result = PyFloat_FromDouble(fact);
7     return result;
8 }
```

Changes to:

```
1 Cystck_Object *square(Py_State *s, Cystck_Object args)
2 {
3     double num;
4     CystckArg_parseTuple(s, args, "O", &num);
5     double fact = num * num;
6     Cystck_Object result = CystckFloat_FromDouble(s, fact);
7     Cystck_pushobject(s, result);
8     return 1;
9 }
```

The source transformation for this code involves roughly five rules. First is the change of any `PyObject` patterns to `Cystck_Object`, but before we apply any transformations, we identify what kind of `PyObject` pattern it is. It can be part of a variable declaration/assignment or part of a function definition. The invoked action is different in each case. The former is a normal replacement while the latter involves additional changes to cater to the state as the first argument of a function. The `PyArg_ParseTuple` transformation is also a direct search and replace, while the return statement has to be modified, replacing it with two statements. One pushes a value

to the stack and the other returns an index to the top of the stack. Note that we choose to number indices from 1 not 0, this has no unique semantics. CyStck follows the naming convention for most API functions starting with `Cystck` instead of `Py`, except the CyStck functions expect the first argument to be the `state`. Table 7.3 has a summary of patterns and their transformations for this example code. In general keeping this non-polluting naming convention is beneficial for cleaner code but also simplifies automatic migration and avoids conflicting function names. We have implemented other non-trivial rules, like the module registration that even involves unique macros but do not discuss every rule here due to space limitations.

Match	Replacement
<code>PyObject *square(</code>	<code>Cystck_Object *square(Py_State *s</code>
<code>PyArg_ParseTuple</code>	<code>CystckArg_parseTuple</code>
<code>Py_Object result</code>	<code>Cystck_Object result</code>
<code>PyFloat_FromDouble(</code>	<code>CystckFloat_FromDouble(s</code>
<code>return @</code>	<code>Cystck_pushobject(s, @); return 1;</code>

Table 7.3: Patterns and Transformations for The Hypothetical Example in Section 7.6.1

Static Analysis

Other than normal pattern matching and transformation, we also do static analysis. In contrast to generic static analysis of C programs that we also do with CPPChecker [95], the goal for static analysis in this tool is to primarily check for errors related to the following specific implementation flaws that can appear in the transformed source code of an extension module:

1. *Custom types*: for function signatures or places in the source code where we find especially custom types other than `PyObject` or known API types, we check and infer the types for correctness.
2. *Stack and Array Overflow*: As discussed earlier, since we have set a limit to the sizes of the stack and array, we also statically scan the program to look out for

parts of the code that can overflow these data structures. We generate warning reports to guide any manual intervention.

3. *Integer Overflows*: Integer operations are a well known cause for erratic behaviour in the Python C API. Program analysis for this is aimed at uncovering integer operations that overflow, use illegitimate bit functionality and conversions that hide the integer value.
4. *Exceptions*: Exception flaws for the Python C API can manifest as either wrong handling between Python and C environments or not raising exceptions correctly. We have complete support for the latter and cover trivial scenarios to check for exception mismatches between Python/C that can cause unwanted control flow in the API code.

To identify integer overflows, we search for places in the code where arguments are passed and values are generated, checking for any sources of integer overflow. Similarly, for memory leaks, we use lexical pattern matching to scan the code for the known Python allocators, isolating any memory flaws. Exceptions in the Python C API should always be returned, therefore, through a static scan of the code, we isolate any thrown exceptions by searching for `CyStck_Err`, and checking if it is returned. Code that uses return values to carry internal errors is checked before any further execution. All APIs in `CyStck` that need to be checked for return values are inspected.

7.6.2 Case Study

This section discusses our experience in using this migration tool for the five large extensions described in Table 7.1.

Methodology

We compared the changes made through our manual port and the changes made by the transformation tool in Tables 7.4, 7.5, 7.6, 7.7 and 7.8. Another view to this evaluation would consider that results are based on several people doing the manual port, and hence multiple manual ports to compare with, but we do not go this route for this initial experimentation. We use the Git diff feature to compare the source files against the original unported extensions.

For each extension, we measure the additions and deletions and number of files modified using `git diff --stats`. We also measure the number of transformations, which are the number of routines invoked to make changes to the source code by the migration tool. We draw conclusions based on the number of files modified for the quantitative insight, addition/deletions using mostly Git diff and manual inspection for any qualitative insight.

<i>UltraJson</i>	<i>Manual Port Metrics</i>	<i>Automated Port Metrics</i>
Additions (+)	815	329
Deletions (-)	690	390
Files modified (#)	6	4
Number of transformations	-	52

Table 7.4: **Migration Metrics for UltraJson** — the automatic port covers approximately 60% of the manual port changes

Discussion of Results

For UltraJson as shown in Table 7.4, the tool is able to port about 60% of the code in terms of both additions and number of files modified. It makes 52 transformations to the original source code. The discrepancy in additions, deletions, and number of files

<i>PicoNumPy</i>	<i>Manual Port Metrics</i>	<i>Automated Port Metrics</i>
Additions (+)	128	85
Deletions (-)	184	99
Files modified (#)	1	1
Number of transformations	-	33

Table 7.5: **Migration Metrics for PicoNumPy** — the automatic port covers approximately 33% of the manual port changes

<i>NumPy</i>	<i>Manual Port Metrics</i>	<i>Automated Port Metrics</i>
Additions (+)	10162	3463
Deletions (-)	1247	3124
Files modified (#)	102	77
Number of transformations	-	57

Table 7.6: **Migration Metrics for NumPy** — the automatic port covers approximately 75% of the manual port changes

<i>Matplotlib</i>	<i>Manual Port Metrics</i>	<i>Automated Port Metrics</i>
Additions (+)	1115	1667
Deletions (-)	1504	3552
Files modified (#)	23	19
Number of transformations	-	49

Table 7.7: **Migration Metrics for Matplotlib** — the automatic port covers approximately 90% of the manual port changes

<i>KiwiSolver</i>	<i>Manual Port Metrics</i>	<i>Automated Port Metrics</i>
Additions (+)	620	1188
Deletions (-)	671	1301
Files modified (#)	11	10
Number of transformations	-	37

Table 7.8: **Migration Metrics for KiwiSolver** — the automatic port covers approximately 90% of the manual port changes

is because in the manual port, we cautiously introduce some helper code during the port. PicoNumPy shows a difference of 33% additions by the tool compared to the manual port and modifies the file completely for all matches shown in Table 7.5; the discrepancy is due to extra manual garbage collection code we add using `ref/unref` to correctly register and unregister reference counts. As pointed out earlier, the tool can not make a judgement on manual `ref/unref` and does not currently even flag these in the report but future work can make this improvement. Table 7.6 shows that for NumPy, the tool is able to make modifications to more than 75% of the number of files in the project compared to the manual port. The manual port makes more modifications and the discrepancy is due to helper functions but also some modifications from stack to heap allocations in the manual port. Table 7.8 shows that the tool also makes modifications to about 90% of the number of files for the KiwiSolver project, compared to the manual port. There is also a discrepancy in additions and deletions due to helper functions in the manual port. About 90% of the files for the Matplotlib project as shown in Table 7.7 were modified and even the semantic accuracy was equally close to the manual port from manual inspection. We do not track modifications in setup for the automated port, which contributes to fewer modifications for this extension, because we do not rename the extension in

the setup file.

7.7 Related Work

Existing work suggests using the Boehm GC [19] for C/C++ applications but the problem we address is not about automating garbage collection for C/C++ applications. It is more about reconciling garbage collection between Python or any other VM and C, regardless of what GC is used in either the VM or native Code. We address high level challenges of identifying roots and tracking references, which are the main challenges for supporting garbage collection for extension modules. Further still, the Boehm GC is a non-moving GC and only helps with the marking phase of a typical mark and sweep policy but not with the sweep phase. Therefore, even if we used the Boehm GC for the C code, the GC still needs information on reachability to accurately de-allocate objects. Our combination of the stack and light-weight handles solves these two problems. The reference mechanism described, including liballocs, helps us address the further missing pieces, which is that in some cases reachability alone is not enough to determine when an object should be deallocated. Therefore, given this context, the Boehm GC is not a direct solution to the main problems this work addresses.

Techniques for interoperability between languages and native extensions have been a subject of research for several years. Most of these have generally explored different alternatives for cooperation among languages, for example Grimmer et al. [57, 58] proposes combining interpreters on a single VM and sharing the abstract syntax trees. Barret et al. [8], discuss syntactic composition of PHP and Python with references between the languages. We chose not to invent a new interaction between Python and C because there is good traction of the current API, designed as an FFI. Rather we explore ways of improving it to address its current challenges.

Several papers have analyzed the inefficiencies [100, 130, 80, 73, 72] of the Python C API, in areas like performance and memory management among others. The results of these studies have led to the development of Python profilers that provide metrics on native code [10] but none of the projects has directly addressed or attempted to fix the problems by redesigning the API, which is the most feasible future.

Several languages have employed different designs for their C APIs in ways different from the Python C API. Ruby uses conservative stack scanning to manage memory for the C API [119], V8 uses handles [84] to also manage garbage collection, Perl still uses the union type like Python [101] while Lua uses an abstract stack to handle the same problem [76, 75]. We build on some of these techniques and report on combining a stack and light-weight handles to reliably support garbage collection.

HPy is work pioneered by the PyPy team [135] on an experimental Python C API and uses handles but due to overhead analyzed by Kalibera and Jones [84], we did not fully commit to only handles in this work, instead we combine light-weight handles and a stack. Kell et al. [86, 11] proposes a Pythonic way of writing native extensions and proposes liballocs to manage memory. We do not use this Pythonic approach for CyStck but explore how to handle object lifetimes using liballocs for an FFI like CyStck.

Chapter 8

Conclusions

This chapter summarizes the key contributions and proposes future directions for the explored topics of research.

8.1 Summary

The insights and observations discussed in this dissertation defend the thesis that designing for high performance garbage collection is generally holistic and custom in nature, depending on how other components of a language are implemented and the features supported.

8.1.1 OMR-based Garbage Collection

The results presented in Chapter 3 are our answer to research question **RQ1**:

Can the use of the Eclipse OMR garbage collection framework lead to the correct, portable, and language-independent implementation of high performing garbage collectors for the RPython Meta-tracing-JIT-based dynamic languages?

Our results on garbage collection modularity in agreeing with existing work show that a framework like OMR can still achieve high performance and the resulting GC is easier to implement and maintain.

We further observe that it is possible to use Eclipse OMR for RPython-based virtual machines but the GCs have overhead. A possible extension to this work would be to implement more sophisticated GCs for the RPython framework and make performance comparisons for different workloads to see if OMR gives any improvements, using certain GC policies. A useful exploration can try to keep constant some aspects while varying others; for example, the GC and runtime can be constant, for an analysis on different architectures [14]. This analysis is useful to understand any shortcomings in the RPython framework GCs that OMR can solve.

The overhead we observe is in allocation operations and we anticipate write barriers being another source of similar overhead for a framework like OMR. An optimization for this overhead would be writing the fast paths of allocation and write barriers in PyPy, rather than calling out to OMR's C++ code for every barrier and allocation operation. For allocation, in that case we must use a C++ data structure for the bump pointer, to explicitly read the cursor and limit from a thread-local address using special constant offsets, then store the cursor back into that data structure. Thread-local state is optional. If the comparison fails it goes to a slow path that does a call out to the OMR library. We can do something similar for the write barriers.

The other overhead that we can not optimize easily is the overhead involved with calling a library. For PyPy given it is written in Python, we have a whole Python layer calling OMR glue code exposed in a C header file. The overhead per call is difficult to reduce for OMR.

Optimization may also require OMR being as opaque as it can be with runtimes and trying to call the basic API. For example, an integration for generational policies can use routines that do not access semispaces directly. This introduces more calls than

are necessary, so if we replace these calls with generic allocation routines after policy configuration, we can reduce the calls we make to OMR. The calls on the C/C++ side may still be somewhat similar but we would avoid wrapping the individual calls in Python, instead they can be abstracted away in one C/C++ routine that we invoke. The other optimization that can help is allocating all objects on the OMR heap and doing away with the PyPy convention of externally malloced/free objects. One of the potential mistakes in the first integration was trying to completely emulate the existing PyPy/RPython GCs, which may have introduced overhead. OMR has optimization around where it allocates objects of certain sizes but we have to watch for fragmentation.

8.1.2 Dynamic Trace Sizing

The garbage collection cost analysis in Chapter 4 proves that automatic memory management operations are expensive for applications and tracing JIT compilers. For all applications, we conclude that garbage collectors should be implemented to be configurable and have the ability to be controlled by application code or use a concurrent/fully concurrent GC with sub-millisecond pauses. For example, for games and certain server operations, it is desirable to allow application developers to disable and enable garbage collection to conveniently manage the effects of long pause times. We specifically answer research question *RQ2*:

Can we attach garbage collection overhead to any specific implementation aspects of meta-tracing-JIT-based dynamic languages?

We observe that JIT tracing can pressure garbage collection, motivating the need for GC related optimizations in the design of JIT compilers. This may be unique for JIT compilers implemented in a managed language like RPython in our case since there is a certain presumption that JITs are written in C or some other language below the GC.

The profile-guided dynamic trace sizing technique and its evaluation, discussed in Section 4.4, show that sizing the trace depending on runtime behaviour of an application has performance benefits. The findings also argue that for synthetic workloads and in arguably less complex applications, a static trace size that works across applications of interest can be more efficient compared to the profiling overhead for a profile-based technique like DTS. The profiling information about the application is potentially better gathered offline to reduce the overall overhead of the DTS technique.

8.1.3 Type Optimization for Heterogeneous Collections

The *type-based stores* technique described in Chapter 5 restructures the layout of heterogeneous collections, splitting the collection into contiguous memory partitions and unboxing the items in the partitions.

Storage strategies will benefit workloads with heavy use of homogeneous collections better than our technique. This is because we incur overhead due to extra checking before and during splitting. Our technique is especially beneficial for large heterogeneous collections or where a large homogeneous collection becomes heterogeneous, which addresses research question **RQ3**:

Can heterogeneous collections in dynamic languages be stored in an unboxed form in memory without significant overhead?

As shown in the evaluation, we guarantee gains for workloads with large heterogeneous collections. Type-based stores can complement type speculation optimizations, because objects are stored in their concrete types. Type speculation benefits from our technique by being able to know the types of the objects involved.

In general, applications with heterogeneous data structures with primitives will benefit from this optimization. The number of data types may have less impact since we

support integers, floats, and strings. In fact integers and floats share a store, in which case, by default only two stores will ever be created for any heterogeneous collection; hence the number of data types is unimportant.

Storing objects in an unboxed form benefits even non-JIT interpreters like CPython in both speed and memory consumption. However, JIT-based interpreters gain more because JIT compilers benefit further from using the type information for further optimization, like type speculation mentioned in V8 earlier.

Regarding fragmentation, when objects are stored in an unboxed form, primitives are now not individually heap allocated objects; hence less GC pressure, which is why we observe speedups for most benchmarks. Other than creating several objects through splitting, our technique does not manage GC allocation; fragmentation should be handled by the underlying GC allocator like it would any other objects. Splitting in our type-based stores technique is high-level, at least in the garbage collection sense.

8.1.4 Call-stack-based Presizing and Garbage Collection

In Chapter 6, we propose a technique that optimizes data structure resizing in dynamic languages. It addresses the branching and allocation context challenges for data structure presizing, by inferring the allocation context through a profiled call site symbol and stack height, along with a profiled allocation site and list size information. This technique answers research question **RQ4**:

Can we achieve context aware collection presizing for dynamic languages with minimal overhead?

We implement an offline version of the technique, and the results do not reflect the full profiling overhead. Our main contribution is the use of stack height to infer the context required for profiling. The manner in which the profiles are stored and used for a language can vary, for example, mementos can be used for that, instead of storing the data in the object header.

We also observe that there is a relationship between the stack height and the live size, a key insight to drive GC optimizations. For example, the square-root rule algorithm [132] proposed by Kirisame et al. can use the stack height for automatically sizing the heap instead of the live size, because live size can not be measured until after a garbage collection cycle. Optimizations of this nature are best evaluated on realistic workloads to observe their impact on pause time, cache behaviour etc.

8.1.5 Memory Management for Native Extensions

The empirical study in the work discussed in Chapter 7 answers research question **RQ5**:

Can combining a stack and light-weight handles, along with object introspection, lead to correct memory management for Python native extensions?

The lessons from this research can be categorized in two areas: *garbage collection* and *migration of existing extensions*.

Garbage Collection: By promoting manual memory management, C APIs do not decouple GC details from the API, which complicates evolution and makes the API error prone. From our study we report that it is possible to mostly automate memory management for native extensions, including hiding GC details. However, there may be a need for manual configuration of, for example, GC tasks like configuring allocators and reclamation routines that should be exposed to the program. This automation is still possible while ensuring a clean design of the API.

Due to a conflict in the policies for when objects should be de-allocated, we had to support a manual reference mechanism to allow control or extend the life of certain objects that could still be required even when a C program returned. We explored *liballocs* to accurately process object life times. The key point here is that reachability

alone is insufficient to determine object life times, and a single party i.e., VM or native code, does not have enough information to allow for collection precision of an object. Techniques to handle cyclic objects that reference counting algorithms can not reclaim become onerous to design for. Handling weak references can also force commitment to using object proxies, which introduce overhead.

Migration: Automatically migrating Python native extensions is also not as hard as the Python 2 to 3 transition [103] from our experience. We demonstrate that through traditional pattern matching using text transformation and regex search, it is not complex to perform transformations on large code bases of native code, an approach that the Python 2 to 3 transition would gain very little from unless a much deeper AST introspection is explored. It is even easier if the new API enforces the convention of naming API entries in the form of `Py_*`.

Complemented by thorough automated testing, any tool for migration automation should also support non-trivial static analysis to check and validate the semantics of the ported code. With our current accuracy level for this automation, we find that most of the code can be automatically ported, leaving a few corner cases that can be handled manually, which is a relief to many developers especially for large systems.

8.2 Future Work

We observe in this dissertation that the research on garbage collection frameworks have matured enough by now and the problems are well-known. For now and the future, the focus should be on optimizing write barriers and allocation to motivate adoption in production runtimes. This may also be in the form of new GC policies or a hybrid of existing policies. OMR and MMTk should strive to demonstrate diverse integrations in runtimes as a testbed to inform adoption by mainstream language ecosystems.

Profile guided GC and memory related optimizations for collections and the JIT will still need to address the profiling overhead since they are more beneficial if they are online and dynamic. The language-independent type-based stores and storage strategies techniques can be extended to a library that is not based on the RPython toolchain for wider adoption by other languages, where adopting the toolchain is not an alternative.

The rest of this section proposes in detail three main directions for future work, namely, *optimal heap limits*, *phase-based garbage collection* and *memory safety for FFIs*.

8.2.1 Optimal Heap Limits

One direction for future work is to use the stack height profiling to estimate the live size that can guide optimizations like heap sizing and reduce the pause time for stop-the-world garbage collectors.

The heap limit algorithm [132] by Kirisame et al., can be modified to be based on the stack height instead of live size, leading to a new rule:

$$M = S + \sqrt{Sg/cs} \tag{8.1}$$

The M is the optimal heap limit, live memory replaced with stack depth S , mean allocation rate g and mean garbage collection speed cs .

This modified rule has to be experimentally evaluated, as we can not guarantee in practice that the results in the paper by Kirisame et al., can be reproduced by solely replacing live size with stack depth, even though we find a correlation. Alternatively, a factor of the stack depth can be used to size the heap instead of using the rule.

8.2.2 Phase-based Garbage Collection

Phase-based GC triggering based on the stack height can be investigated with server workloads like the ones in Chapter 4. For example, such a study can identify if phase-based opportunistic garbage collection can reduce GC pause times, and thereby avoid server timeout for requests.

8.2.3 Memory Safety of Foreign Function Interfaces

Towards solving the FFI challenges in Chapter 7, future work can empirically study the extent of memory safety for FFIs using Rust and its interaction with the C/C++ API, as a case study to practically appreciate the problem since Rust has several semantics on code safety.

The outcome of this study would be novel techniques to help ensure consistent semantics in the face of memory model disparities, pointer stability across the VM/C boundary and correct object lifetime management between the VM and FFI. Verification tools can also be built to help with validation of memory safety aspects.

Bibliography

- [1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., USA, 1986.
- [2] Biswas Apratim. Using Python’s Garbage Collector with Pandas DataFrames: Higher Efficiency and Performance for Larger Datasets. 2020. Accessed: 2021-09-10, <https://towardsdatascience.com/python-garbage-collection-article-4a530b0992e3>.
- [3] Pierre Augier. Elements Kinds in V8. <https://v8.dev/blog/elements-kinds>, 2017.
- [4] Pierre Augier. The Fundamental Package for Scientific Computing with Python. <https://github.com/numpy/numpy>, 2018.
- [5] Pierre Augier. Elements Kinds in V8. <https://sschakraborty.github.io/benchmark/index.html>, 2023.
- [6] Brenda S. Baker. Parameterized Pattern Matching by Boyer-Moore-Type Algorithms. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA ’95, page 541–550, USA, 1995. Society for Industrial and Applied Mathematics.

- [7] Gergö Barany. Python Interpreter Performance Deconstructed. In *Proceedings of the Workshop on Dynamic Languages and Applications*, Dyla'14, page 1–9, New York, NY, USA, 2014. Association for Computing Machinery.
- [8] Edd Barrett, Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. Fine-grained Language Composition: A Case Study (Artifact). In *DARTS-Dagstuhl Artifacts Series*, volume 2. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.
- [9] Alexandre Bergel, Alejandro Infante, Sergio Maass, and Juan Pablo Sandoval Alcocer. Reducing Resource Consumption of Expandable Collections: The Pharo case. *Science of Computer Programming*, 161:34–56, 2018.
- [10] Emery D Berger. Scalene: Scripting-language Aware Profiling for Python. *arXiv preprint arXiv:2006.03879*, 2020.
- [11] Guillaume Bertholon and Stephen Kell. Towards Seamless Interfacing between Dynamic Languages and Native Code. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, pages 38–47, 2019.
- [12] Peter A Bigot and Saumya K Debray. A Simple Approach to Supporting Untagged Objects in Dynamically Typed Languages. *The Journal of Logic Programming*, 32(1):25–47, 1997.
- [13] Dennis Bijlsma, Miguel Alexandre Ferreira, Bart Luitjen, and Joost Visser. Faster Issue Resolution with Higher Technical Quality of Software. *Software Quality Journal*, 20:265–285, 2012.
- [14] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *Proceedings. 26th International Conference on Software Engineering*, pages 137–146, 2004.

- [15] 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 '04/Performance '04, page 25–36, New York, NY, USA, 2004. Association for Computing Machinery.
- [16] Stephen M. Blackburn and Kathryn S. McKinley. Ulterior Reference Counting: Fast Garbage Collection without a Long Wait. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, pages 344–358. Association for Computing Machinery. event-place: Anaheim, California, USA.
- [17] Stephen M Blackburn and Kathryn S. McKinley. In or Out? Putting Write Barriers in Their Place. *SIGPLAN Not.*, 38(2 supplement):175–184, jun 2002.
- [18] Steve Blackburn. The New MMTK. Personal communication, 2021.
- [19] Hans-J Boehm, Alan J Demers, and Scott Shenker. Mostly Parallel Garbage Collection. *ACM SIGPLAN Notices*, 26(6):157–164, 1991.
- [20] Hans-J. Boehm and Mike Spertus. Garbage Collection in the next C++ Standard. In *Proceedings of the 2009 International Symposium on Memory Management*, ISMM '09, page 30–38, New York, NY, USA, 2009. Association for Computing Machinery.
- [21] Hans-Juergen Boehm and Mark Weiser. Garbage Collection in an Uncooperative Environment. *Softw. Pract. Exper.*, 18(9):807–820, September 1988.
- [22] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. Allocation Removal by Partial Evaluation in a Tracing JIT. In *Proceedings of the 20th ACM SIGPLAN Workshop*

- on Partial Evaluation and Program Manipulation*, PEPM '11, page 43–52, New York, NY, USA, 2011. Association for Computing Machinery.
- [23] 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*, ICPOOLPS '09, page 18–25, New York, NY, USA, 2009. Association for Computing Machinery.
- [24] Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. Storage Strategies for Collections in Dynamically Typed Languages. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages amp; Applications*, OOPSLA '13, page 167–182, New York, NY, USA, 2013. Association for Computing Machinery.
- [25] Carl Friedrich Bolz and Armin Rigo. How to not Write Virtual Machines for Dynamic Languages. In *3rd Workshop on Dynamic Languages and Applications*, 2007.
- [26] Rodrigo Bruno and Paulo Ferreira. A Study on Garbage Collection Algorithms for Big Data Environments. *ACM Comput. Surv.*, 51(1), January 2018.
- [27] Rodrigo Bruno, Duarte Patricio, José Simão, Luis Veiga, and Paulo Ferreira. Runtime Object Lifetime Profiler for Latency Sensitive Big Data Applications. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.
- [28] Craig Chambers, David Ungar, and Elgin Lee. An Efficient Implementation of SELF: a Dynamically-typed Object-oriented Language Based on Prototypes. *ACM Sigplan Notices*, 24(10):49–70, 1989.

- [29] Howard Chen, Wei-Chung Hsu, and Dong-yuan Chen. Dynamic Trace Selection Using Performance Monitoring Hardware Sampling. In Richard Johnson, Tom Conte, and Wen-mei W. Hwu, editors, *1st IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2003)*, 23-26 March 2003, San Francisco, CA, USA, pages 79–90. IEEE Computer Society, 2003.
- [30] Chris J Cheney. A Nonrecursive List Compacting Algorithm. *Communications of the ACM*, 13(11):677–678, 1970.
- [31] Wu Chenyang and Ni Min. Dismissing Python Garbage Collection at Instagram. 2017. Accessed: 2021-09-10, <https://instagram-engineering.com/dismissing-python-garbage-collection-at-instagram-4dca40b29172>.
- [32] Jian Chou, Lin Chen, Hui Ding, Jingxuan Tu, and Baowen Xu. A Method of Optimizing Django Based on Greedy Strategy. In *2013 10th Web Information System and Application Conference*, pages 176–179, 2013.
- [33] Clauss Christian. An experiment about Numpy and Pyhandle/HPy. <https://github.com/pygame/pygame/tree/main/examples>, 2019.
- [34] Daniel Clifford, Hannes Payer, Michael Stanton, and Ben L. Titzer. Memento Mori: Dynamic Allocation-Site-Based Optimizations. In *Proceedings of the 2015 International Symposium on Memory Management, ISMM '15*, page 105–117, New York, NY, USA, 2015. Association for Computing Machinery.
- [35] Stack Exchange Community. Stack Exchange Data Dump. <https://archive.org/details/stackexchange1>, 2014. Accessed: 2021-09-10.
- [36] PyPy Contributors. Garbage Collection in PyPy. <https://doc.pypy.org/en/latest/introduction.html>, 2011. Accessed: 2019-05-10.
- [37] PyPy Contributors. PyPy Benchmarks. https://github.com/hg-mirrors/pypy_benchmarks, 2014.

- [38] PyPy Contributors. HPy: A Better API for Python. <https://hpy.readthedocs.io/en/latest/overview.html>, 2019.
- [39] PyPy Contributors. Potential Project List: Various GCs. <https://doc.py.py.org/en/latest/project-ideas.html#various-gcs>, 2020. Accessed: 2020-05-14.
- [40] John D. Cook. Ironclad. <https://code.google.com/archive/p/ironclad/>, 2009. Accessed: 2021-07-23.
- [41] Antonio Cuni. HPy Kick-off Sprint Report. <https://morepypy.blogspot.com/2019/12/hpy-kick-off-sprint-report.html>, 2019. Accessed: 2020-12-18.
- [42] Benoit Dalozé, Arie Tal, Stefan Marr, Hanspeter Mössenböck, and Erez Petrank. Parallelization of Dynamic Languages: Synchronizing Built-in Collections. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018.
- [43] Mattias De Wael, Stefan Marr, Joeri De Koster, Jennifer B Sartor, and Wolfgang De Meuter. Just-in-time Data Structures. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, pages 61–75, 2015.
- [44] Ulan Degenbaev, Jochen Eisinger, Manfred Ernst, Ross McIlroy, and Hannes Payer. Idle Time Garbage Collection Scheduling. *SIGPLAN Not.*, 51(6):570–583, 2016.
- [45] Lukas Diekmann. Memory Optimizations for Data Types in Dynamic Languages. *Masterarbeit of Institut Fur Informatik, Softwaretechnik und Programmiersprachen, Heinrich Heine Universitat Dusseldorf*, pages 1–51, 2012.
- [46] Charles Dierbach. Python as a First Programming Language. *Journal of Computing Sciences in Colleges*, 29(3):73–73, 2014.

- [47] Edsger W Dijkstra, Leslie Lamport, Alain J. Martin, Carel S. Scholten, and Elisabeth FM Steffens. On-the-fly Garbage Collection: An Exercise in Cooperation. *Communications of the ACM*, 21(11):966–975, 1978.
- [48] Matt D’Souza, James You, Ondřej Lhoták, and Aleksandar Prokopec. TASTyTruffle: Just-in-Time Specialization of Parametric Polymorphism. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA2):1561–1588, 2023.
- [49] Eclipse. Eclipse OMR. <https://github.com/eclipse/omr>, 2016. Accessed: 2020-04-10.
- [50] Jake Edge. Making Python 3 More Attractive. <https://lwn.net/Articles/640179/>, 2015. Accessed: 2021-07-22.
- [51] J Eriksson, Pedro Ojeda-May, T Ponweiser, and T Steinreiter. Profiling and Tracing Tools for Performance Analysis of Large Scale Applications. *PRACE: Partnership for Advanced Computing in Europe*. <https://prace-ri.eu/wp-content/uploads/WP237.pdf>, 2016.
- [52] Robert Fitzgerald, Todd B Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: An Optimizing Compiler for Java. *Software: Practice and Experience*, 30(3):199–232, 2000.
- [53] Daniel Frampton. *Garbage Collection and the Case for High-level Low-level Programming*. Phd thesis, Research School of Computer Science, The Australian National University, 2010.
- [54] Björn Franke, Zhibo Li, Magnus Morton, and Michel Steuwer. Collection skeletons: Declarative abstractions for data collections. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering*,

SLE 2022, page 189–201, New York, NY, USA, 2022. Association for Computing Machinery.

- [55] Robin Garner. *The Design and Construction of High Performance Garbage Collectors*. Phd thesis, Research School of Computer Science, The Australian National University, 2012. <http://hdl.handle.net/1885/99879>.
- [56] Michael Greenberg. The Dynamic Practice and Static Theory of Gradual Typing. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *3rd Summit on Advances in Programming Languages (SNAPL 2019)*, volume 136 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:20, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
- [57] Matthias Grimmer, Roland Schatz, Chris Seaton, Thomas Würthinger, Mikel Luján, and Hanspeter Mössenböck. Cross-language Interoperability in a Multi-language Runtime. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 40(2):1–43, 2018.
- [58] Matthias Grimmer, Chris Seaton, Thomas Würthinger, and Hanspeter Mössenböck. Dynamically Composing Languages in a Modular Way: Supporting C Extensions for Dynamic Languages. In *Proceedings of the 14th International Conference on Modularity*, pages 1–13, 2015.
- [59] Aheui Working Group. PyPy/RPython tutorial - Aheui JIT Interpreter with PyPy/RPython. <https://github.com/aheui/rpaheui/blob/main/LOG.md>, 2015. Accessed: 2021-04-03.
- [60] David Gudeman. Representing type information in dynamically typed languages. Citeseer, Technical Report, TR 93-27, <https://citeseerx.ist.psu.edu>, 1995.

- [61] AI H2O. Welcome to H2O 3. <https://docs.h2o.ai/h2o/latest-stable/h2o-docs/welcome.html>, 2021. Accessed: 2021-09-10.
- [62] Josh Haberman. How do Native Extensions Manage Memory? Part 1: Ruby (MRI). <https://blog.reverberate.org/2016/06/12/native-extensions-memory-management-part1-ruby-mri.html>, 2016.
- [63] Josh Haberman. How do Native Extensions Manage Memory? Part 2: JavaScript (V8). <https://blog.reverberate.org/2016/10/17/native-extensions-memory-management-part2-javascript-v8.html>, 2016.
- [64] Boehm Hans. The Boehm-Demers-Weiser conservative C/C++ Garbage Collector. <https://github.com/ivmai/bdwgc/>, 2009. Accessed: 2021-04-03.
- [65] Boehm J. Hans. Conservative GC Algorithmic Overview. <https://www.hboehm.info/gc/gcdescr.html>, 2009. Accessed: 2021-04-03.
- [66] Boehm Hans-J. A Garbage Collector for C and C++. <https://www.hboehm.info/gc/>, 2009. Accessed: 2021-04-03.
- [67] Hiroshige Hayashizaki, Hiroshi Inoue, and Peng Wu. Integration of Trace Selection and Trace Profiling in Dynamic Optimizers, October 1 2013. US Patent 8,549,498.
- [68] Johannes Henning, Tim Felgentreff, Fabio Niephaus, and Robert Hirschfeld. Toward Presizing and Pretransitioning Strategies for GraalPython. In *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming*, <programming> '20, page 41–45, New York, NY, USA, 2020. Association for Computing Machinery.
- [69] Matthew Hertz and Emery D. Berger. Quantifying the Performance of Garbage Collection vs. Explicit Memory Management. In *Proceedings of the 20th Annual*

- ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, page 313–326, New York, NY, USA, 2005. Association for Computing Machinery.
- [70] Alex Holkner and James Harland. Evaluating the Dynamic Behaviour of Python Applications. In *Proceedings of the Thirty-Second Australasian Conference on Computer Science-Volume 91*, pages 19–28, 2009.
- [71] Ben Hoyt. An Arm Wrestle with Python’s Garbage Collector. <http://tech.oyster.com/pythons-garbage-collector/>, 2013. Accessed: 2021-09-10.
- [72] Mingzhe Hu and Yu Zhang. The Python C API: Evolution, Usage Statistics, and Bug Patterns. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 532–536. IEEE, 2020.
- [73] Mingzhe Hu, Yu Zhang, Wenchao Huang, and Yan Xiong. Static Type Inference for Foreign Functions of Python. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, pages 423–433. IEEE, 2021.
- [74] Richard L Hudson, JE Moss, Amer Diwan, and Christopher F Weight. A Language-Independent Garbage Collector Toolkit. 1991.
- [75] Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes. The Evolution of An Extension Language: A History of Lua. In *Proceedings of V Brazilian Symposium on Programming Languages*, pages B-14–B-28, 2001.
- [76] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The Evolution of Lua. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, page 12–26, New York, NY, USA, 2007. Association for Computing Machinery.
- [77] Berkin Ilbeyi. *Co-Optimizing Hardware Design and Meta-Tracing Just-in-Time Compilation*. PhD thesis, Cornell University, 2019.

- [78] Mohamed Ismail and G. Edward Suh. Efficient Nursery Sizing for Managed Languages on Multi-Core Processors with Shared Caches. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2020, page 1–15, New York, NY, USA, 2020. Association for Computing Machinery.
- [79] Ismail, Mohamed and Suh, G. Edward. Quantitative Overhead Analysis for Python. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 36–47, 2018.
- [80] Chengman Jiang, Baojian Hua, Wanrong Ouyang, Qiliang Fan, and Zhizhong Pan. PyGuard: Finding and Understanding Vulnerabilities in Python Virtual Machines. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, pages 468–475. IEEE, 2021.
- [81] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. CRC Press, 2023.
- [82] Sophie Kaleba. Avoiding Monomorphization Bottlenecks with Phase-based Splitting. In *Companion Proceedings of the 2021 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, SPLASH Companion 2021, page 16–18, New York, NY, USA, 2021. Association for Computing Machinery.
- [83] Sophie Kaleba, Octave Larose, Richard Jones, and Stefan Marr. Who You Gonna Call: Analyzing the Run-Time Call-Site Behavior of Ruby Applications. In *Proceedings of the 18th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS 2022, page 15–28, New York, NY, USA, 2022. Association for Computing Machinery.

- [84] Tomas Kalibera and Richard Jones. Handles Revisited: Optimising Performance and Memory Costs in a Real-Time Collector. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, page 89–98, New York, NY, USA, 2011. Association for Computing Machinery.
- [85] Takafumi Kataoka, Tomoharu Ugawa, and Hideya Iwasaki. A Framework for Constructing Javascript Virtual Machines with Customized Datatype Representations. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC '18, page 1238–1247, New York, NY, USA, 2018. Association for Computing Machinery.
- [86] Stephen Kell. The Inevitable Death of VMs: A Progress Report. In *Companion Proceedings of the 2nd International Conference on the Art, Science, and Engineering of Programming*, pages 61–62, 2018.
- [87] Marisa Kirisame, Pranav Shenoy, and Pavel Panchekha. Optimal Heap Limits for Reducing Browser Memory Use. *Proc. ACM Program. Lang.*, 6(OOPSLA2):986–1006, 2022.
- [88] K. Kontogiannis, J. Martin, K. Wong, R. Gregory, H. Müller, and J. Mylopoulos. Code Migration through Transformations: An Experience Report. In *CASCON First Decade High Impact Papers*, CASCON '10, page 201–213, USA, 2010. IBM Corp.
- [89] Michele Lacchia. Radon's Documentation. <https://radon.readthedocs.io>, 2012. Accessed: 2021-01-09.
- [90] Akash Lal and G. Ramalingam. Reference Count Analysis with Shallow Aliasing. volume 111, pages 57–63, 2010.
- [91] Yossi Levroni and Erez Petrank. An On-the-Fly Reference Counting Garbage Collector for Java. In *Proceedings of the 16th ACM SIGPLAN Conference on*

- Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, page 367–380, New York, NY, USA, 2001. Association for Computing Machinery.
- [92] Siliang Li and Gang Tan. Finding Reference-counting Errors in Python/C Programs with Affine Analysis. In *ECOOP 2014 - Object-Oriented Programming*, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pages 80–104, Germany, January 2014. Springer Verlag. 28th European Conference on Object-Oriented Programming, ECOOP 2014 ; Conference date: 28-07-2014 Through 01-08-2014.
- [93] Yi Lin, Stephen M. Blackburn, Antony L. Hosking, and Michael Norrish. Rust as a Language for High Performance GC Implementation. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2016, page 89–98, New York, NY, USA, 2016. Association for Computing Machinery.
- [94] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. Learning-Based Memory Allocation for C++ Server Workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 541–556, New York, NY, USA, 2020. Association for Computing Machinery.
- [95] Daniel Marjamaki. A Tool for Static C/C++ Code Analysis. <https://cppcheck.sourceforge.io/>, 2008.
- [96] Stefan Marr and Benoit Daloz. Few Versatile vs. Many Specialized Collections: How to Design a Collection Library for Exploratory Programming? In

Stefan Marr and Jennifer B. Sartor, editors, *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming, Nice, France, April 09-12, 2018*, pages 135–143. ACM, 2018.

- [97] Stephen Marr. Are We Fast Yet? Comparing Language Implementations with Objects, Closures, and Arrays. <https://github.com/smarr/are-we-fast-yet>, 2016. Accessed: 2019-06-14.
- [98] Durant Martin. Pandas and Memory Allocation. <https://martinduranta.github.io/blog/pandas-and-memory-allocation/>, 2017. Accessed: 2021-09-10.
- [99] John McCarthy. Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I. *Communications of the ACM*, 3(4):184–195, 1960.
- [100] Raphaël Monat. *Static Type and Value Analysis by Abstract Interpretation of Python Programs with Native C Libraries*. PhD thesis, Sorbonne université, 2021.
- [101] Hisham Muhammad and Roberto Ierusalimschy. C APIs in Extension and Extensible Languages. *J. Univers. Comput. Sci.*, 13(6):839–853, 2007.
- [102] Todd Mytkowicz, Devin Coughlin, and Amer Diwan. Inferred Call Path Profiling. *SIGPLAN Not.*, 44(10):175–190, 2009.
- [103] Joanna Nanjey. *Python 2 and 3 Compatibility: With Six and Python-Future Libraries*. Apress, 2017.
- [104] Joanna Nanjey. This is a Python micro-benchmark suite that exercises memory management for CPython and PyPy. <https://github.com/CAS-Atlantic/python-gc-benchmark>, 2019. Accessed: 2021-05-10.

- [105] Joannah Nanjeyke, David Bremner, and Aleksandar Micic. *Eclipse OMR Garbage Collection for Tracing JIT-Based Virtual Machines*, page 244–249. IBM Corp., USA, 2021.
- [106] Paul Oman and Jack Hagemeister. Metrics for assessing a software system’s maintainability. In *Proceedings Conference on Software Maintenance 1992*, pages 337–338. IEEE Computer Society, 1992.
- [107] Tobias Pape, Carl Friedrich Bolz, and Robert Hirschfeld. Adaptive Just-in-Time value class optimization for lowering memory consumption and improving execution time performance. *Science of Computer Programming*, 140:17–29, 2017. Object-Oriented Programming and Systems (OOPS 2015).
- [108] Tobias Pape, Tim Felgentreff, Robert Hirschfeld, Anton Gulenko, and Carl Friedrich Bolz. Language-Independent Storage Strategies for Tracing-JIT-Based Virtual Machines. *SIGPLAN Not.*, 51(2):104–113, October 2015.
- [109] S. Paul and A. Prakash. A framework for source code search using program patterns. *IEEE Trans. Softw. Eng.*, 20(6):463–475, jun 1994.
- [110] Thomas Perl. Python Garbage Collector Implementations CPython, PyPy and GaS. https://thp.io/2012/python-gc/python_gc_final_2012-01-22.pdf, 2012. Seminar. TU Wien. January 22, 2012. Accessed: 2020-05-05.
- [111] Phusion. Performance and Memory Usage Comparisons. <http://www.rubyenterpriseedition.com/comparisons.html>, 2012. Accessed: 2021-09-10.
- [112] David W. Price, Algis Rudys, and Dan S. Wallach. Garbage Collector Memory Accounting in Language-Based Systems. In *2003 IEEE Symposium on Security and Privacy (S&P 2003), 11-14 May 2003, Berkeley, CA, USA*, pages 263–274. IEEE Computer Society, 2003.

- [113] Python. Python Performance Benchmark Suite. <https://github.com/python/pyperformance>, 2016. Accessed: 2019-05-10.
- [114] Alex Reinking, Ningning Xie, Leonardo de Moura, and Daan Leijen. Perceus: Garbage Free Reference Counting with Reuse. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 96–111, 2021.
- [115] Dudfield René. Experiments with the New Low Latency PyPy garbage collector in a thread. <http://renesd.blogspot.com/2019/01/experiments-with-new-low-latency-pypy.html>, 2019. Accessed: 2021-09-10.
- [116] Stefan Richthofer. JyNI-using native CPython-extensions in Jython. *arXiv preprint arXiv:1404.6390*, 2014.
- [117] Armin Rigo and Samuele Pedroni. PyPy’s Approach to Virtual Machine Construction. In Peri L. Tarr and William R. Cook, editors, *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pages 944–953. ACM, 2006.
- [118] David Schneider and Carl Friedrich Bolz. The Efficient Handling of Guards in the Design of RPython’s tracing JIT. In *Proceedings of the sixth ACM workshop on Virtual machines and intermediate languages*, pages 3–12, 2012.
- [119] Chris Seaton. *Specialising Dynamic Techniques for Implementing the Ruby Programming Language*. The University of Manchester (United Kingdom), 2015.
- [120] Seeni, Mohamed. H2O explicit garbage collection in Python. <https://asmblogs.wordpress.com/2019/09/05/h2o-explicit-garbage-collection-in-python/>, 2019. Accessed: 2021-09-10.

- [121] Rifat Shahriyar. *High Performance Reference Counting and Conservative Garbage Collection*. Phd thesis, Research School of Computer Science, The Australian National University, 2015. <http://hdl.handle.net/1885/99879>, {DOI}=10.25911/5d690a05b02ff.
- [122] Rifat Shahriyar, Stephen M. Blackburn, and Daniel Frampton. Down for the Count? Getting Reference Counting Back in the Ring. In *Proceedings of the 2012 International Symposium on Memory Management, ISMM '12*, pages 73–84. Association for Computing Machinery. Beijing, China.
- [123] Rifat Shahriyar, Stephen Michael Blackburn, Xi Yang, and Kathryn S. McKinley. Taking off the Gloves with Reference Counting Immix. *SIGPLAN Not.*, 48(10):93–110, October 2013.
- [124] Shopify. Set of benchmarks for the YJIT CRuby JIT compiler and other Ruby implementations. <https://github.com/Shopify/yjit-bench>, 2023. Accessed: 2023-03-14.
- [125] Jeremy Singer, Gavin Brown, Ian Watson, and John Cavazos. Intelligent Selection of Application-Specific Garbage Collectors. In *Proceedings of the 6th International Symposium on Memory Management, ISMM '07*, page 91–102, New York, NY, USA, 2007. Association for Computing Machinery.
- [126] Dag I.K. Sjøberg, Bente Anda, and Audris Mockus. Questioning Software Maintenance Metrics: a Comparative Case Study. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '12*, page 107–110, New York, NY, USA, 2012. Association for Computing Machinery.
- [127] Mallku Soldevila, Beta Ziliani, and Daniel Fridlender. Understanding Lua’s Garbage Collection: Towards a Formalized Static Analyzer. In *Proceedings of*

the 22nd International Symposium on Principles and Practice of Declarative Programming, PPDP '20, New York, NY, USA, 2020. Association for Computing Machinery.

- [128] Peter Steenkiste and John Hennessy. LISP on a Reduced-Instruction-Set-Processor. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, page 192–201, New York, NY, USA, 1986. Association for Computing Machinery.
- [129] Victor Stinner. pythoncapi-compat project. <https://github.com/python/pythoncapi-compat>, 2023.
- [130] Jialiang Tan, Yu Chen, Zhenming Liu, Bin Ren, Shuaiwen Leon Song, Xipeng Shen, and Xu Liu. Toward Efficient Interactions between Python and Native Libraries. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 1117–1128, New York, NY, USA, 2021. Association for Computing Machinery.
- [131] Alexandros Tasos, Juliana Franco, Sophia Drossopoulou, Tobias Wrigstad, and Susan Eisenbach. Reshape Your Layouts, Not Your Programs: A Safe Language Extension for Better Cache Locality (SCICO Journal-first). In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, volume 166 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 31:1–31:3, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.
- [132] Sanaz Tavakolisomeh, Marina Shimchenko, Erik Österlund, Rodrigo Bruno, Paulo Ferreira, and Tobias Wrigstad. Heap Size Adjustment with CPU Control. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, MPLR 2023, page 114–128, New York, NY, USA, 2023. Association for Computing Machinery.

- [133] Kiwi team. Efficient C++ implementation of the Cassowary constraint solving algorithm. <https://github.com/nucleic/kiwi>, 2018.
- [134] Matplotlib team. Matplotlib: plotting with Python. <https://github.com/matplotlib/matplotlib>, 2018.
- [135] PyPy Team. Inside cpyext: Why emulating CPython C API is so Hard. <https://morepypy.blogspot.com/2018/09/inside-cpyext-why-emulating-cpython-c.html>.
- [136] Sebastian Ullrich and Leonardo de Moura. Counting immutable beans: Reference counting optimized for purely functional programming. *arXiv preprint arXiv:1908.05647*, 2019.
- [137] Vlad Ureche, Aggelos Biboudis, Yannis Smaragdakis, and Martin Odersky. Automating ad hoc data representation transformations. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 801–820, 2015.
- [138] Guido Van Rossum et al. Python programming language. In *USENIX annual technical conference*, volume 41, page 36, 2007.
- [139] Vicente Franco and Juliana Patricia. *Orca: Ownership and Reference Count Collection for Actors*. Phd thesis, Imperial College London, 2019. <http://hdl.handle.net/10044/1/67952>.
- [140] Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. *Design and Evaluation of Gradual Typing for Python*, pages 45–56. ACM, 2014.
- [141] Michael M. Vitousek, Jeremy G. Siek, and Avik Chaudhuri. Optimizing and Evaluating Transient Gradual Typing. In *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages, DLS 2019*, page 28–41, New York, NY, USA, 2019. Association for Computing Machinery.

- [142] Michal Wegiel and Chandra Krintz. Concurrent Collection as an Operating System Service for Cross-Runtime Cross-Language Memory Management. Technical report, 2010. http://nidhogg.cs.ucsb.edu/research/tech_reports/reports/2010-15.pdf.
- [143] P. R. Wilson and T. G. Moher. Design of the opportunistic garbage collector. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '89, page 23-35, New York, NY, USA, 1989. Association for Computing Machinery.
- [144] Li Zekun. Copy-on-write Friendly Python Garbage Collection. <https://instagram-engineering.com/copy-on-write-friendly-python-garbage-collection-ad6ed5233ddf>, 2017. Accessed: 2021-09-10.

Appendix A

Type-based Stores for Collections

A.1 Type-based Stores for PyPy

A.1.1 Implementation

The algorithm for the optimization described in this work is summarized in Algorithm 4 and impacts list, set, and dictionary data objects. We modify the PyPy3.9 branch, which is the latest version at the time of writing this dissertation. This algorithm is invoked at the point where a collection switches from homogeneous to heterogeneous. We first read all the items in the heterogeneous collection (line 3), then transfer control to the layout function to allocate the required stores (lines 6 and 10). Depending on whether the collection is a list, set or dictionary, we create a new instance of storage areas, either create sequential storage areas for lists or hash-like storage areas for dictionaries, then split the objects, sorted by type into the storage areas, and finally update the map on line 7 for lists.

Algorithm 4: TypeBasedStores: layout(ds)

1 **Input:** Let ds be the heterogeneous collection.

Result: Layout and unbox heterogeneous collections

2 initialization;

3 items = ds.getItems();

4 ds.discard();

5 **if** *type(ds) == list* **then**

6 createSequentialStores(items);

7 updateMap();

8 **end**

9 **if** *type(ds) == dict or set* **then**

10 createHashStores(items);

11 **end**

To achieve the procedures in Algorithm 4, we implement a new internal data structure in PyPy. Splitting of the heterogeneous collections involves intercepting the implementation of the collections at the points where collections currently switch to the general object strategy or at initialization after checking if the collection is heterogeneous, thereby creating stores for each collection, and passing items by type to the stores.

Listing A.1 is an extract of the implementation of the `List` data structure/collection in PyPy. Lines 5–9 consist of the method that currently switches any heterogeneous collections, while lines 10–13 have an implementation of the `length()` method for lists. For lists, the `switch_to_object_strategy` (line 5) routine and `from_storage_and_strategy` (not shown in this code), which is one of the other operations, have special conditions to handle heterogeneous collections by boxing.

```
1 class W_ListObject(W_Root):
2     strategy = None
```

```

3 storages = None
4 ....
5 def switch_to_object_strategy(self, w_item=None):
6     list_w = self.getitems()
7     self.clear(self.space)
8     if self.stores is None:
9         self.stores = create_stores(self.space, list_w)
10 def length(self):
11     if self.stores:
12         return self.stores.size
13     return self.strategy.length(self)

```

Listing A.1: **Splitting Objects to Type-based Stores** – pseudocode for the key implementation details of the type-based stores technique for the list data structure in PyPy

We therefore intercepted them to use or pass their data to the type-based stores mechanism. We also consequently modify existing functions like `length()` on line 10, etc., to instead read and/or modify the stores of the said collection (line 12). In PyPy and CPython, sets are handled as dictionaries with the actual items of the set as keys and corresponding values being `None`, so in our implementation sets are handled by the code for dictionaries.

A.2 PyPy Evaluation

A.2.1 Execution Speed

Performance Measurements		
Benchmark	PyPy-with-TBS	PyPy-without-TBS
float	1.14 \pm 0.07	1.53 \pm 0.11
richards	0.95 \pm 0.18	1.35 \pm 0.22
deltablue	0.11 \pm 0.32	0.19 \pm 0.10
ai	1.64 \pm 0.21	2.39 \pm 0.40
eparse	3.37 \pm 0.42	3.61 \pm 0.15
meteor-contest	1.44 \pm 0.42	1.58 \pm 0.08
fannkuch	5.88 \pm 0.67	6.01 \pm 0.70
spectral-norm	3.33 \pm 0.28	3.42 \pm 0.06
chaos	2.42 \pm 0.06	2.16 \pm 0.06
nbody	1.72 \pm 0.01	1.69 \pm 0.01
telco	5.27 \pm 0.16	5.21 \pm 0.17
call_simple	3.25 \pm 0.12	3.30 \pm 0.08
regex_effbot	0.04 \pm 0.01	0.06 \pm 0.01
nbody_modified	1.53 \pm 0.12	1.56 \pm 0.11
unpack_sequence	0.08 \pm 0.06	0.05 \pm 0.03
fib	4.97 \pm 0.18	4.84 \pm 0.16

Table A.1: Benchmark Speed results in seconds, lower is better

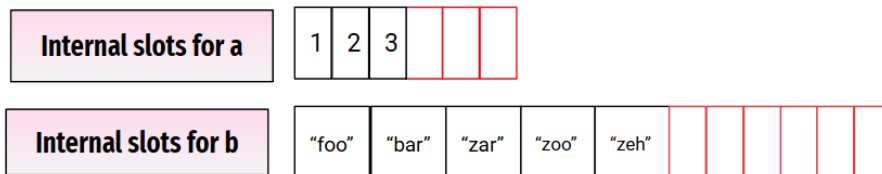
Appendix B

Context Aware Profiling

B.1 Motivation

B.1.1 Collection Over-allocation Schematic

```
1 a = [1, 2, 3]
2 b = ["foo", "bar", "zar", "zoo", "zeh"]
3 if flag:
4     a.extend(b)
5 else:
6     a.append(4)
```



B.2 Accuracy of Context Identifiers

B.2.1 Ambiguity Example for a Python Benchmark

```

1, bm_unpack_sequence.py:454, get, 23, <module> -> <module> -> _find_and_load -> _find_and_load_unlocked -> _load_unlocked ->
exec_module -> _call_with_frames_removed -> <module> -> _find_and_load -> _find_and_load_unlocked -> _load_unlocked ->
exec_module -> _call_with_frames_removed -> <module> -> TextWrapper -> compile -> _compile -> compile -> parse -> _parse_sub ->
_parse -> get
2, bm_unpack_sequence.py:419, append, 25, <module> -> <module> -> _find_and_load -> _find_and_load_unlocked -> _load_unlocked
-> exec_module -> _call_with_frames_removed -> <module> -> _find_and_load -> _find_and_load_unlocked -> _load_unlocked ->
exec_module -> _call_with_frames_removed -> <module> -> TextWrapper -> compile -> _compile -> compile -> parse -> _parse_sub ->
_parse -> _parse_sub -> _parse -> append
3, bm_unpack_sequence.py:442, test_all, 4, <module> -> <module> -> test_all
4, bm_unpack_sequence.py:436, test_tuple_unpacking, 5, <module> -> <module> -> test_all -> test_tuple_unpacking
5, bm_unpack_sequence.py:426, do_unpacking, 6, <module> -> <module> -> test_all -> test_tuple_unpacking -> do_unpacking
6, bm_unpack_sequence.py:437, test_list_unpacking, 5, <module> -> <module> -> test_all -> test_list_unpacking
7, bm_unpack_sequence.py:432, do_unpacking, 6, <module> -> <module> -> test_all -> test_list_unpacking -> do_unpacking

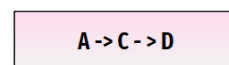
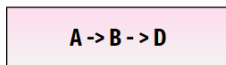
```

B.2.2 Active Record Resizing Case for a Python Benchmark

```

5, bm_unpack_sequence.py:426, do_unpacking, 7, <module> -> <module> -> test_all -> test_tuple_unpacking -> do_unpacking
6, bm_unpack_sequence.py:437, test_list_unpacking, 5, <module> -> <module> -> test_all -> test_list_unpacking
7, bm_unpack_sequence.py:432, do_unpacking, 7, <module> -> <module> -> test_all -> test_list_unpacking -> do_unpacking

```

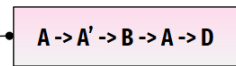
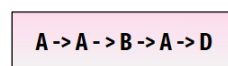
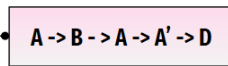
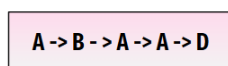


B.2.3 Call Site Wrapping Case for a Python Benchmark

```

5, bm_ai.py:80, n_queens, 5, <module> -> <module> -> test_n_queens -> n_queens
6, bm_ai.py:85, n_queens, 5, <module> -> <module> -> test_n_queens -> n_queens

```



Vita

Candidate's full name

Joannah Nanjekye

Research Areas

Garbage Collection, Programming Language Design, Optimization, Migration, Privacy and Security Programming Models.

Education

- May. 2019 **UNB/IBM CAS Canada**, NB, Canada
- APR. 2024 PhD in COMPUTER SCIENCE
Advisor: Dr. David Bremner
Topic: Memory Management Techniques for Dynamic Languages
- Jan. 2022 **Kings College London**, Somerset, London, UK
- APR. 2024 Research Associate (Visiting Research Student)
Advisor: Dr. Laurence Tratt
Research Project: Language Migration
- Jan. 2022 **Australian National University**, Canberra, Australia
- APR. 2024 Visiting Research Scholar (Visiting Research Student)
Advisor: Dr. Steve Blackburn
Research Project: Integration of MMTk to PyPy
- 2016-2018 **Kenya Aeronautical College**, Nairobi, Kenya
Diploma in Aeronautical Engineering
- 2009-2014 **Makerere University**, Kampala, Uganda
BSc. Software Engineering

Programming Languages Leadership

- SEPT. 2019 Python Core Developer
 - TODATE Maintainer of CAPI and Sub interpreters
- JUN. 2019 Director of the Python Software Foundation
- JUN. 2023 Served as vice chair in 2019 - 2020

Honors and Awards

- SEPT. 2023 IBM CAS Student of the Year Award 2023
- JUL. 2022 Google Academic Scholarship Award Winner, thesis chosen for presentation at award summit
- JUN. 2022 Rust Foundation Individual Project Award, for one year
- SEPT. 2019 NBIF STEM Social and Innovation graduate Award, for one year
- JUN. 2015 Grace Hopper Conference Faculty Award

Books

- DEC. 2017 Python 2 and 3 Compatibility
 - Joannah Nanjekye**
 - Apress, Berkeley, December 2017
 - Online ISBN: 978-1-4842-2955-2
 - Print ISBN: 978-1-4842-2954-5
 - Link : <https://www.apress.com/de/book/9781484229545>

Patents and Disclosures

- SEPT. 2023 Optimal JIT Trace Sizing for Virtual Machines
 - Authors: Joannah Nanjekye, David Bremner, Aleksandar Micic
 - Docket Number: P20220429US01
 - Status: Filed
 - File Date: 22 Sept, 2023
- NOV. 2023 Type-based Stores for Data Structure Polymorphism
 - Authors: Joannah Nanjekye, David Bremner, Aleksandar Micic
 - Invention Reference: P202305693 (Record ID 99220386)
 - Invention Publisher: IP.com
 - Invention Publication Date: March 22, 2024
 - Invention Publication Number: IPCOM000274036D
 - Publication Link: <https://priorart.ip.com/IPCOM/00274036D>
- JAN. 2024 Context Aware Presizing for Dynamic Languages
 - Authors: Joannah Nanjekye, David Bremner, Aleksandar Micic
 - Status: Rated as Publish for a defensive patent
 - Invention Ref: P202400921
 - Record ID: 99282899

Publications

- 2023 Towards Reliable Memory Management for Python Native Extensions
Joannah Nanjeyke, David Bremner, Aleksandar Micic
Proceedings of the 18th ACM International Workshop on Implementation, Compilation, Optimization of OO Languages, Programs and Systems (ICOOOLPS 2023)
- 2022 The Garbage Collection Cost For Meta-Tracing JIT-based Dynamic Languages
Joannah Nanjeyke, David Bremner, Aleksandar Micic
Proceedings of the 32nd Annual International Conference on Computer Science and Software Engineering (CASCON '22), November 2022, Pages 140–149
- 2021 Eclipse OMR garbage collection for tracing JIT-based virtual machines
Joannah Nanjeyke, David Bremner, Aleksandar Micic
Proceedings of the 31st Annual International Conference on Computer Science and Software Engineering (CASCON '21), November 2021, Pages 244–249
- 2017 Python 2 and 3 Compatibility
Joannah Nanjeyke
Apress, Berkeley, December 2017
Online ISBN: 978-1-4842-2955-2
Print ISBN: 978-1-4842-2954-5

Working Papers

- 2024 Data Structure Polymorphism and Presizing Optimizing Techniques for Dynamic Languages
Joannah Nanjeyke, David Bremner, Aleksandar Micic
Submitted to: The Journal of Object Technology (JOT 2024)
- 2024 Garbage Collection with Frameworks: Lessons Learned from OMR and MMTk
Joannah Nanjeyke, David Bremner, Aleksandar Micic, Stephen Blackburn, Carl Friedrich Boltz
- 2024 TMVs: Smoother Program Migration in the Face of Disruptive Language Changes
Joannah Nanjeyke, Laurence Tratt

Service

2024: ARTIFACT EVALUATION COMMITTEE MEMBER, PLDI 2024
2023: ARTIFACT EVALUATION COMMITTEE MEMBER, CGO 2024
2023: COMMITTEE MEMBER, PYCON US 2023
2023: CHAIR, PYCON UGANDA, 2023
2021: REVIEWER CASCON, 2021
2020: REVIEWER CASCON, 2020

Teaching

JUL - AUG 2024	University of New Brunswick <i>Instructor</i> Developed the curriculum and taught CS3853: Computer Organization and Architecture
JAN - APR 2024	University of New Brunswick <i>Graduate Teaching Assistant</i> Working on grading tasks for the course: CS4613: Programming Language Interpretation
SEPT - DEC 2023	University of New Brunswick <i>Graduate Teaching Assistant</i> Worked on grading tasks for the course: CS3113: PIntroduction to Numerical Methods
JAN - APR 2021	Faculty of computer science, University of New Brunswick (UNB) <i>Student Administrative Assistant</i> Worked on administrative tasks for the admission of graduate students
SEPT - DEC 2021	University of New Brunswick <i>Graduate Teaching Assistant</i> Worked on grading tasks for a course on Advanced software practices
JAN - APR 2021	University of New Brunswick <i>Graduate Teaching Assistant</i> Worked on grading tasks for the course: CS4613: Programming Language Interpretation

SEPT - DEC 2020	University of New Brunswick <i>Graduate Teaching Assistant</i> Worked on grading tasks for the course: CS2613: Programming Language Laboratory .
JAN - APR 2020	University of New Brunswick <i>Graduate Teaching Assistant</i> Worked on grading tasks for the course: CS1031: Discrete Structures
SEPT - DEC 2019	University of New Brunswick <i>Graduate Teaching Assistant</i> Worked on grading tasks for the course: CS2613: Programming Language Laboratory

Keynotes and Special Invited Conference Presentations

OCT 2023	Special Invited Talk, Native Profiling for Python PyCon MEA 2023, Dubai, UAE
APR 2023	Special Invited Talk, Signal-based Native Profiling for Python 17th Annual Research Exposition of the UNB Faculty of Computer Science, Fredericton, Canada
JUL 2021	Keynote presentation, Python the Bad Parts EUROPYTHON 2021, Ramnebacken, Sweden
AUG 2020	Keynote presentation, Python 4.x: What do you Expect Pycon Africa, Accra, Ghana
AUG 2017	Special Invited Talk, Ruby in Containers EuRuKo , Vienna Austria

Other Conference Presentations

- SEP 2023 A Rust-based Garbage Collector for Python
RustConf 2023, Albuquerque, NM, USA
- JUL 2023 Towards Reliable Memory Management for Python Native
Extensions
17th Workshop on Implementation, Compilation, Optimization of
Object- Oriented Languages, Programs and Systems
(ICOOOLPS 2023),Seattle, USA.
- APR 2023 Towards Native Profiling for Python, Python Language Summit
presentation, PyCon US 2023, Salt Lake City, USA
- NOV 2022 The Garbage Collection Cost For Meta-Tracing JIT-based Dynamic
Languages, WeaveSphere 2022, powered by CASCON x EVOKE,
Toronto Canada
- AUG 2022 Memory Management Techniques for Dynamic Languages
Google Award Summit 2022, Remote
- NOV 2021 Eclipse OMR Garbage Collection for Tracing JIT-based Virtual
Machines, 31st Annual International Conference on Computer
Science and Software Engineering (CASCON x EVOKE 2021),
Markham, Canada
- JUL 2021 An Eclipse OMR-based Garbage Collector for Python
16th Workshop on Implementation, Compilation, Optimization of
Object- Oriented Languages Programs and Systems
(ICOOOLPS 2021), Aarhus, Denmark
- NOV 2020 On using Eclipse OMR to Implement a GC in Python
4th AORTCC 2020 Workshop, CASCON x EVOKE 2020,
Toronto, Canada
- NOV 2019 My Python is Restricted: Implementing a Garbage Collector
in High Level
Language using the OMR GC Framework, 3rd AORTCC 2019
Workshop, CASCON x EVOKE 2019, Toronto, Canada
- SEP 2019 PEP 554: Multiple Subinterpreters in the Standard Library
CPython Core Developer Sprints, Bloomberg, London, UK

Poster Presentations

- APR 2023 Eclipse OMR Garbage Collection for Meta-tracing-JIT-based
Dynamic Languages
17th Annual Research Exposition of the UNB Faculty
of Computer Science, Fredericton, Canada
- JUL 2023 Towards Reliable Memory Management for Python Native Code
ECOOP/ISSTA 2023 conference, Seattle, Washington, United States.
- NOV 2022 Dynamic Trace Sizing for Virtual Machines
WeaveSphere 2022, powered by CASCON x EVOKE,
Toronto Canada
- MAY 2022 1031: Python on OMR
IBM CASTLE 2022
Markham, Canada
- MAY 2021 Python on OMR
IBM CASTLE 2021
Markham, Canada
- NOV 2020 A Control Flow Optimizer for CPython
30th Annual International Conference on Computer Science and
Software Engineering (CASCON x EVOKE 2020),
Markham, Canada
- MAY 2020 Python on OMR: Interpreter Integration
IBM CASTLE 2020
Markham, Canada

Work Experience

JUN - AUG 2017	CEPH <i>FOSS Outreachy Intern</i>
MAR - SEPT 2017	EAST AFRICAN CIVIL AVIATION ACADEMY <i>Aeronautical Engineering Apprentice</i>
JUL - SEPT 2016	Rails Girls Summer of Code <i>Fellow</i>
OCT 2014 - AUG 2015	FINTECH UGANDA LIMITED <i>Programmer</i>
JUN - AUG 2021	Laboremus Uganda Limited <i>Software Developer Apprentice</i>