Precision without Regret: Sound and Efficient Staged Analysis for Managed Runtimes

Submitted in partial fulfillment of the requirements
of the degree of

Doctor of Philosophy

by

Aditya Anand
(Roll No. 23D0381)

Supervisor
Prof. Manas Thakur

Department of Computer Science & Engineering
INDIAN INSTITUTE OF TECHNOLOGY BOMBAY
2026

To

my family :)

Thesis Approval

The thesis titled

Precision without Regret: Sound and Efficient Staged Analysis for Managed Runtimes

by

Aditya Anand
(Roll No. 23D0381)

is approved for the degree of

Doctor of Philosophy

Examiner	Examiner
Guide	Chairman

Date:

Place: IIT Bombay, Powai.

Declaration

I declare that this written submission represents my ideas in my own words and where others ideas or words have been included I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken where needed.

Aditya Anand
(Roll No: 23D0381)

INDIAN INSTITUTE OF TECHNOLOGY BOMBAY, INDIA

CERTIFICATE OF COURSE WORK

This is to certify that **Aditya Anand** (Roll No. 23D0381) was admitted to the candidacy of Ph.D. degree on July 2023, after successfully completing all the courses required for the Ph.D. programme. The details of the course work done are given below.

S.No	Course Code	Course Name	Credits
1	CS 787	Language Engineering for Complex Programs: A C++ Perspective	6
2	CSS801	Seminar	4
3	GC 101	Gender in the workplace	PP
4	TA 101	Teaching Assistant Skill Enhancement & Training (TASET)	PP
		Total Credits	**10**

IIT Bombay

Date: Dy. Registrar (Academic)

Acknowledgements

My PhD journey has been filled with excitement, deadlines, new experiences, countless memorable moments, and, looking back now, a truly rewarding journey. It began in the valleys of the Himalayas and concluded near the shores of the Arabian Sea. I started this journey at IIT Mandi as a Master's scholar, working in an area that has always fascinated me. Within a few months of joining the group, I decided to continue this journey further and convert to the PhD programme. A year after making that decision, I moved to IIT Bombay along with my advisor, beginning a new chapter of my PhD. Looking back, the journey from the mountains of Mandi to the shores of Mumbai has been much more than an academic transition. It has been a journey of learning, growth, challenges, and, most importantly, the people who made it meaningful.

First and foremost, I would like to express my deepest gratitude to my advisor, Prof. Manas Thakur, whom I deeply admire and look up to. At every stage of my PhD, his constant guidance and support showed me how to find the best way forward. His mentorship has shaped not only my academic journey but also the way I approach challenges and opportunities in life, and that is something I will carry with me long after my PhD. One of the most valuable lessons I have learned from him is simple yet profound: keep doing your absolute best, and the world will find a way for you.

I would also like to express my sincere gratitude to my progress committee members, Prof. Uday Khedker and Prof. Supratik Chakraborty, for their valuable and insightful feedback throughout my PhD. Their suggestions and perspectives have been instrumental in shaping my research. I am also deeply grateful to Prof. Rupesh Nasre from IIT Madras and Prof. Stefan Marr from Johannes Kepler University Linz for their thorough review of my thesis and for providing valuable feedback. I would also like to sincerely thank Prof. Vikram Gadre for kindly agreeing to officiate my PhD defense.

I would also like to take this opportunity to thank the wonderful administrative staff in the department, Vijay P. Ambre, Balasaheb Gaikwad, and Sunanda Ghadge, as well as Bhagyalakshmi Joshi from the Academic Office, for their constant support and for always going out of their way to help whenever needed.

The list of friends who have been a part of my journey, and the many wonderful moments we have shared, is truly endless. I am deeply grateful to all my friends from IIT Mandi, as well as the past and present members of the PLATO Lab at IIT Bombay, for making my PhD journey more enjoyable, memorable, and fulfilling. The countless conversations, discussions, shared meals, walks, and moments of laughter have made the lab much more than just a place of work. I am especially grateful to Arjun Harikumar and Meetesh Mehta for the many interesting discussions we had during my initial days, both during our walks and in the lab. Those conversations, ranging from research to everything beyond it, were always enjoyable and have given me many fond memories. I am thankful to Supriya Bhide for her constant willingness to help and for always being supportive. I am also grateful to Preet Soni for the many wonderful and thought-provoking discussions we shared. I am sincerely grateful to Jayasri Pa, Vedant Chavan, and Sania Khokhar for being such wonderful friends and for making the journey more enjoyable with their companionship and warmth. I would also like to thank my badminton group, particularly Anvay Shah, Darshan Prabhu, and Ramsundar Anandanarayanan, for the many enjoyable moments we spent together on the court. The doubles matches, friendly competition, and conversations before and after the games provided a much-needed break from the demands of research and brought a great deal of joy to my time at IIT Bombay.

Finally, I would like to express my deepest gratitude to my father, SP Chaudhary, my mother, Bibha Rani, and my sister, Aditi Garg, for their unwavering love, support, and encouragement throughout my journey. Their constant faith in me, their support for my decisions, and their encouragement during both the challenging and rewarding moments have been a source of strength and motivation. I am truly grateful for the sacrifices they have made and for always standing by me, no matter the circumstances. Whatever I have achieved today would not have been possible without their unconditional support and belief in me. I owe a significant part of who I am today to them, and I will always remain deeply grateful for their presence and support in my life.

Abstract

Managed runtimes for modern object-oriented languages provide abstractions for memory management, execution, and optimization, relieving developers from manual resource handling. In such runtimes (eg: Java Virtual Machines) objects are typically allocated on the heap, and reclaimed using automatic garbage collection (GC). Most such runtimes also consist of just-in-time (JIT) compilers that optimize memory access and GC times by employing *escape analysis*: an object that does not escape (outlive) its allocating method can be allocated on (and freed up with) the stack frame of the corresponding method. However, in order to minimize the time spent in JIT compilation, the scope of such useful analyses is quite limited, thereby restricting their precision significantly. On the contrary, even though it is feasible to perform precise program analyses statically, it is not possible to use their results in a managed runtime without a closed-world assumption.

To address this gap, this thesis proposes a "staged" static+dynamic scheme that allows one to harness the results of a precise static escape analysis for allocating objects on stack, while taking care of both soundness and efficiency concerns in the runtime. The approach performs *optimistic stack allocation* during JIT compilation based on static analysis results, handles the challenges associated with features that may invalidate the optimism using a novel idea of run-time checks and *dynamic heapification*. It further reduces the overheads associated with the checks using another novel notion of *stack ordering*, again supported by a static analysis. To evaluate the effectiveness, the thesis compares the proposed scheme with that of a production JVM and demonstrates much higher stack allocation. Furthermore, the enhanced stack allocation leads to a significant reduction in the number of GC cycles and brings decent performance improvements, particularly in memory-constrained environments.

Building on this foundation, the thesis further explores the complementary strengths of

static analysis and JIT compilation. While JIT compilers typically sacrifice the precision of analysis for efficiency, they are capable of performing sophisticated speculative optimizations based on run-time profiles. On the contrary, while static compilers can perform precise interprocedural analysis, they remain conservative where run-time specialization could improve precision. This thesis proposes a first-of-its-kind approach to enrich static analysis with the possibility of speculative optimization during JIT compilation, as well as its usage to perform aggressive stack allocation on a production JVM. The approach, named `CoSSJIT` involves three key contributions. First, it identifies scenarios where a static analysis is conservative but a JIT speculation could improve precision. Second, it introduces the notion of "speculative conditions" and plugs them into a static interprocedural dataflow analyzer, to generate partial results to be specialized at run-time. Finally, it extends a production JIT compiler to incorporate and refine these results using resolved speculative conditions, leading to a practical approach that efficiently combines the best of both worlds. To fully realize these benefits, this thesis also outlines few modifications and tunings to the JIT compiler that maximize the effectiveness of the static+dynamic scheme.

Despite the effectiveness of staged static+dynamic approaches, such "staged analyses" lack a formal theoretical foundation. In particular, it is unclear if one could transform a general whole-program analysis (that resolves dependencies across all program elements) to a staged one that involves evaluation of statically generated partial results later. To address this, this thesis proposes a formal model of partial analysis based on the classical theory of partial evaluation. The model characterizes partial analysis as the evaluation of program dependencies with respect to statically available information, producing intermediate results that can later be specialized. Inspired from Futamura projections, it introduces a novel notion of *Analysis Mix* (AM) projections to generate specialized evaluators that process these partial results using runtime information during JIT compilation. The thesis then uses the proposed model to establish the correctness and precision properties of the idea of staging, independent of the analysis under consideration. Its applicability is demonstrated through examples drawn from non-trivial Java program analyses, including an implementation of the pipeline for one such analysis, along with discussion of future possibilities. Overall, the thesis broadens the scope of staged analysis for languages with managed runtimes, and presents concrete applications that lead to performance improvements in real-world systems.

Publications

1. Aditya Anand, Vijay Sundaresan, Daryl Maier, and Manas Thakur. 2025. CoSSJIT: Combining Static Analysis and Speculation in JIT Compilers. Proc. ACM Program. Lang. 9, OOPSLA2, Article 371 (October 2025), 27 pages.
   DOI: https://doi.org/10.1145/3763149 [7]

2. Aditya Anand and Manas Thakur. Partial Program Analysis for Staged Compilation Systems. Formal Methods in System Design, 65(1):195–230, June 2024.
   DOI: https://doi.org/10.1007/s10703-024-00458-x [9]

3. Aditya Anand, Solai Adithya, Swapnil Rustagi, Priyam Seth, Vijay Sundaresan, Daryl Maier, V. Krishna Nandivada, and Manas Thakur. Optimistic Stack Allocation and Dynamic Heapification for Managed Runtimes. Proc. ACM Program. Lang., 8(PLDI), June 2024.
   DOI: https://doi.org/10.1145/3656389 [6]

4. Aditya Anand and Manas Thakur. Principles of Staged Static+Dynamic Partial Analysis. In Gagandeep Singh and Caterina Urban, editors, Static Analysis (SAS), pages 44–73, Cham, 2022. Springer Nature Switzerland.
   DOI: https://link.springer.com/chapter/10.1007/978-3-031-22308-2_4 [8]

Acronyms

Acronym	**Definition**
BCI	Byte Code Index
GC	Garbage Collector
HC	Heapification Checks
JIT	Just In Time
JVM	Java Virtual Machine
SCC	Strongly Connected Component
SPRE	Specialized Partial-Result Evaluator

Table 1: Lexicographical order of acronyms used in the thesis

Contents

Declaration v

Acknowledgements ix

Abstract xi

Publications xiii

Acronyms xv

List of Figures xxi

List of Tables xxv

1 Introduction 1

1.1 Optimistic Stack Allocation . . . 3
1.2 Combining Speculation and Static Analysis . . . 5
1.3 Tuning the VM for Static+Dynamic Analysis . . . 8
1.4 Formalizing Static+Dynamic Analysis . . . 9
1.5 Contributions made by the Thesis . . . 13
1.6 Organization of the Thesis . . . 15

2 Background 17

2.1 Managed Runtimes . . . 17
2.2 Eclipse OpenJ9 . . . 18
2.3 Speculation during JIT Compilation . . . 19
2.4 Deoptimization during JIT Compilation . . . 20
2.5 Escape Analysis and Stack Allocation . . . 20
2.6 Static Analysis for Managed Runtimes . . . 21
2.6.1 Dependencies and Conditional Values . . . 22
2.6.2 Generation of Conditional Values . . . 22
2.6.3 Evaluation of Conditional Values . . . 23
2.6.4 Scope and Applicability of Conditional Values . . . 24

3 Optimistic Stack Allocation **27**
3.1 Motivating Example . . . 27
3.1.1 Detecting Incorrect Stack Allocation . . . 29
3.1.2 Moving Objects and Correcting References . . . 30
3.1.3 Imparting Efficiency through Stack Ordering . . . 30
3.2 Optimistic Stack Allocation in Eclipse OpenJ9 . . . 31
3.2.1 Static Context-Sensitive Escape Analysis . . . 32
3.2.2 Dynamic Heapification . . . 35
3.3 Imparting Efficiency through Stack Ordering . . . 40
3.4 Discussion . . . 42
3.4.1 Correctness . . . 43
3.4.2 Design Choices . . . 43
3.5 Evaluation . . . 44
3.5.1 Experimental Setup . . . 45
3.5.2 Enhancement in Stack Allocation . . . 46
3.5.3 Impact of Additional Stack Allocation . . . 51

4 Combining Speculation and Static Analysis **59**
4.1 Motivating Example . . . 59
4.2 Enriching Static Analysis . . . 63
4.2.1 Handling Polymorphic Callsites . . . 64
4.2.2 Handling Conditional Branching . . . 65
4.2.3 Handling Method Inlining . . . 66
4.2.4 Discussion . . . 69
4.3 Implementation in OpenJ9 . . . 69
4.3.1 Working of OpenJ9 . . . 70
4.3.2 Speculative Stack Allocation in OpenJ9 . . . 71
4.3.3 Discussion . . . 75
4.4 Evaluation . . . 76
4.4.1 Experimental Setup and Benchmarks . . . 76
4.4.2 Increase in Stack Allocation . . . 77
4.4.3 Impact of Stack Allocation . . . 82
4.4.4 Additional Overheads . . . 85

5 Tuning the VM **91**
5.1 Inlining in OpenJ9 . . . 91
5.1.1 Enriching the Inlining Heuristics . . . 92
5.1.2 Cost-Benefit Function for Inlining Heuristics . . . 94
5.2 Adapting Static Results to Runtime Decisions . . . 95
5.3 Stack Allocation at Lower Optimization Levels . . . 97
5.4 Evaluation . . . 98

5.4.1 Static and Dynamic Number of Allocations . . . 98
5.4.2 Impact of Enriched Inlining Heuristics . . . 100
5.4.3 Impact of Adapting Static Analysis Results at Run-time . . . 101
5.4.4 Impact of Escape Analysis at Lower Optimization Levels . . . 102
5.4.5 Impact on Performance . . . 103
5.4.6 Impact on Performance at Lower Heap Size . . . 105

6 Formalizing Static+Dynamic Analysis 107
6.1 Partial Evaluation and Futamura Projections . . . 108
6.1.1 Partial Evaluation . . . 109
6.1.2 First Futamura Projection . . . 109
6.1.3 Second Futamura Projection . . . 110
6.1.4 Third Futamura Projection . . . 111
6.2 Staged Partial Analysis . . . 112
6.2.1 Partial Analysis . . . 112
6.2.2 First AM Projection . . . 114
6.2.3 Second AM Projection . . . 116
6.2.4 Third AM Projection . . . 117
6.2.5 Correctness, Precision, and Efficiency of Staging . . . 118
6.3 Implementing Specializers for Partial-Result Evaluation . . . 122
6.3.1 A Grammar for Conditional Values . . . 123
6.3.2 Conditional-Value Evaluators and Specialization . . . 124
6.4 Incorporating Callbacks from Libraries . . . 128
6.4.1 Computing Callback Dependencies During Static Analysis . . . 129
6.4.2 Staging Schema with Callback Dependencies . . . 130
6.5 Evaluation . . . 132
6.5.1 Computation Performed by Each Stage . . . 133
6.5.2 Computation Performed Statically . . . 133
6.5.3 Computation Performed Dynamically . . . 135
6.6 Directions and Connections . . . 137
6.6.1 Extending Staged Analysis with Speculation . . . 138
6.6.2 Runtime Features: Challenges and Possibilities . . . 139
6.6.3 Drawing Newer Connections . . . 141

7 Related Work 145
7.1 Escape Analysis . . . 145
7.2 Speculation and Runtime Tuning . . . 147
7.3 Imparting Efficiency to Precise JIT Analysis . . . 148
7.4 Partial Evaluation and its Applications . . . 149
7.5 Partial Program Analysis . . . 150
7.6 Staged and Modular Analysis . . . 151

7.7 Handling Dynamic Features 152

8 Conclusion and Future Work **153**

8.1 Limitations 155

8.2 Future Work 156

References **157**

List of Figures

1.1 Block diagram representing static+dynamic approach for optimistic stack allocation. Abbreviations indicate modifications done in OpenJ9 towards: (i) HC: insertion of heapification checks; (ii) DH: dynamic heapification; (iii) SOSA: stack-ordered stack allocation. 4

1.2 Block diagram of our approach `CoSSJIT`. Abbreviations indicate modifications done in OpenJ9 towards the resolution of conditions related to: (i) SMC: speculative polymorphic calls; (ii) SMI: speculative method inlining; and (iii) SBE: speculative branch execution. 6

1.3 A Java code snippet to demonstrate generation of dependencies. 10

1.4 Generation of partial-result evaluators. 11

2.1 A Java code snippet to demonstrate static+JIT analysis. Class `L` is a library class not available during the analysis of the application class `A`. . . 23

2.2 Whole-program analysis. 24

3.1 Motivating example to explain various aspects of our run-time scheme. . . 28

3.2 A snapshot of the run-time stack and the heap after dynamic classloading in Figure 3.1. Dashed edges represent heapification and green edges represent updated points-to edges post heapification. 29

3.3 The heapification check performed at store statements ($a.f = b$). lhsReg and rhsReg store the addresses of the base object (pointed-to by a) and that of the object being stored (pointed-to by b), respectively. 39

3.4 (a) Example code to explain the scenarios for stack ordering. (b) Possible stack layouts while processing store statements for the example in (a). . . . 40

3.5 Points-to graphs for methods doo and bar from Fig. 3.1 during stack ordering: (a) intraprocedural; (b) interprocedural. Nodes O_{p_1} and O_{p_2} represent the objects pointed-to by parameters p1 and p2, respectively. 42

3.6 Performance comparison (default heap size); lower the better. For DaCapo benchmarks the y-axis is time (in msec) and for SPECjvm benchmarks it is the normalized reciprocal of ops/sec. 52

3.7 Number of GC cycles with varied heap sizes (X, 2X, 3X and default heap memory); lower the better. 53

3.8 Performance comparison (minimum heap size for each benchmark); lower the better. For DaCapo benchmarks the y-axis is time (in msec) and for SPECjvm it is the normalized reciprocal of ops/sec. 54

4.1 Motivating example to illustrate various considerations made by CoSSJIT. . 60
4.2 Handling a polymorphic callsite: (a) Existing static analysis; (b) CoSSJIT's static analysis. Arrows indicate (may) points-to relationships. 64
4.3 Handling objects in branches: (a) Existing static analysis; (b) CoSSJIT's static analysis. 66
4.4 Extended example from Figure 4.1 to illustrate benefits of conditional inlining by CoSSJIT. 67
4.5 A callsite with possible inlining: (a) Existing static analysis; (b) CoSSJIT's static analysis. 68
4.6 Number of stack allocations and heapifications (in millions) at different values of ST (Speculation Threshold) for the benchmark h2. 80
4.7 Distribution of the contributions made by different speculative conditions, in terms of the number of objects marked for stack allocation during JIT compilation. 81
4.8 Performance comparison between BaseLine and CoSSJIT; lower the better. For DaCapo benchmarks the y-axis is time (in milli sec) and for SPECjvm benchmarks it is the normalized reciprocal of ops/sec. 82
4.9 Number of GC cycles with varied heap sizes (X, 2X, 3X and default heap memory); lower the better. 84
4.10 Number of heap allocations removed in HotSpot VM, OpenJ9 (BaseLine) and OpenJ9 (CoSSJIT) 88

5.1 Example to explain various aspects of our run-time tuning. 93
5.2 Callsites where better callee methods (in terms of number of stack allocatable objects) were identified and selected for inlining. 101
5.3 Additional objects allocated on the stack after adapting static-analysis results to run-time changes. 102
5.4 Performance at default heap size. 104
5.5 Performance at lower heap size. 105

6.1 Partial evaluation of program P using $\texttt{in}_1$. 109
6.2 Futamura projections in partial evaluation. Note that the three Mixes are the same specializers; we have added subscripted numbers for brevity in referencing them. 110
6.3 A reference to the notations used in rest of the sections. 113
6.4 Partial analysis: Specializing $g_m(x)$ using the set S_x of static dependencies to obtain partial result. 114
6.5 Code snippet from Figure 2.1 reproduced for better readability. 115
6.6 AM projections in partial analysis. Note that $\texttt{Mix}_{(1)}$, $\texttt{Mix}_{(2)}$ and $\texttt{Mix}_{(3)}$ are the same specializers as used in partial evaluation; also, we have added subscripted numbers for brevity in referencing them. 116
6.7 A complete staging of whole-program analysis. 119
6.8 (a) The grammar for our division prepass; (b) Extended grammar for conditional-value evaluation. 123

6.9 Schema of the partial-result evaluator emitted for $g_{\texttt{A.foo}}(O_4)$. 126
6.10 Schema for prototype implementation of staged analysis. 127
6.11 Example to demonstrate the impact of callbacks. 129
6.12 Extensions to the grammar from Figure 6.8 to handle callbacks. 129
6.13 Specializing the partial-result evaluator with $[\![D_x]\!]$ 132
6.14 Number of program elements (PE) and various kinds of dependencies. . . . 134
6.15 Time taken (in seconds) by various steps of Figure 6.10; WPA shows the Whole-Program Analysis time in hours (a '-' indicates that the analysis did not terminate in 4 hours). 136
6.16 Extending a staged static+dynamic scheme with additional dependencies such as speculative conditions. 138
6.17 A Java code snippet to demonstrate control-flow analysis. 140

List of Tables

1 Lexicographical order of acronyms used in the thesis xv

3.1 Stack allocation statistics for various benchmarks with `BaseLine` and `OPT` schemes. The first six benchmarks (biojava–zxing) are from DaCapo 23.10-chopin, the next eight (avrora–sunflow) are from DaCapo 9.12-MRI, and the last eight (compiler–signverify) are from SPECjvm 2008. 47
3.2 Stack allocation statistics with forced compilation of all the methods, for the benchmarks where a longer run leads to noticeable improvements. . . . 50
3.3 Time taken by our static analysis along with the .res file overhead. 56

4.1 Stack allocation statistics for various benchmarks with `BaseLine` and `CoSSJIT` schemes. The first nine benchmarks (avrora–zxing) are from DaCapo 23.11-chopin, the next four (eclipse–lusearch) are from DaCapo 9.12-MRI, and the last eight (compiler–signverify) are from SPECjvm 2008. 78
4.2 Static and JIT overheads of `CoSSJIT`. Improvement column is dashed out for SPECjvm benchmarks because the iteration duration for them is fixed. 86

5.1 Stack allocation statistics and compile time statistics for selected benchmarks for using `CoSSJIT` and `TUNED` schemes. The first nine benchmarks (avrora–zxing) are from DaCapo 23.11-chopin, the next benchmark luindex is from DaCapo 9.12-MRI, and the last six (aes–signverify) are from SPECjvm 2008. 99

Chapter 1

Introduction

Rapid evolution of technology has made computing an essential part of our daily lives. In today's world, software is used to carry out a variety of jobs on many kinds of computers, ranging from small computers to large-scale servers. Most software applications are developed using high-level programming languages, which provide many convenient abstractions for expressing computation. Between the program and the underlying hardware, there exists the run-time system responsible for program execution. Traditionally, programs written in languages like C or C++ place low-level run-time tasks such as memory management, safety guarantee and resource handling on the programmer, whereas modern programming languages such as Java, C#, Scala and JavaScript use managed runtimes that automate many of these tasks.

Typical managed runtimes, such as Java Virtual Machines (JVMs), not only execute (or interpret) the input program, but also employ just-in-time (JIT) compilers to dynamically specialize parts of the program during execution. Although these compilers can improve performance by speculating based on run-time profiles, they must share limited resources with the program for analysis and optimization. As a result, JIT compilers are particularly constrained by the well-known trade-off between the precision and efficiency of program analysis, and often sacrifice the former for the latter. Furthermore, features in modern languages and runtimes can render ahead-of-time optimizations unsound, thereby exposing a fundamental triad of concerns – precision, efficiency, and soundness – in managed runtimes. Let us examine these challenges through the lens of memory management in such runtimes.

Automatic memory management in managed runtimes ensures that memory is allocated and freed up without direct programmer intervention. For example, languages such as Java allocate objects on heap and deallocate them using automatic garbage collection. This unburdens programmers from making complex decisions of manual memory allocation and deallocation, and reduces the possibility of harmful memory bugs. However, allocating objects on heaps introduces certain overheads, including additional indirection while accessing objects and their fields. In addition, a garbage collector for freeing up the memory has its cost as well. As an alternative, if the memory was allocated on stack, the access becomes more efficient as objects can be accessed through simple stack pointer movement. Moreover, the memory allocated on stack gets freed up as soon as the activation record of the allocating method is popped out, thus eliminating the need for explicit garbage collection.

The ability to allocate objects or their fields on the stack instead of on the heap has long intrigued the community, with researchers coming up with different applications of *escape analysis* to enable the same. In this context, two main optimizations are used to enable stack allocation. The first, known as *scalar replacement*, is performed by the C2 just-in-time (JIT) compiler of the HotSpot VM [53]. It uses the results of escape analysis to decompose objects into scalar variables that can be allocated on the stack. Similarly, GraalVM [51] uses a partial-escape analysis [60] to enable scalar replacement in parts of a program when it cannot be performed throughout the program. Nonetheless, there are a large number of scenarios when Java objects cannot be scalarized and need to exist in entirety. The most common case is when an object is passed as an argument to a method that is too large to be inlined. To overcome these limitations, many prior works, instead of trying to decompose objects, choose to allocate them in their original shape on the stack instead of on the heap; this second optimization is referred to as *stack allocation* [15, 18, 70]. However, stack allocation in a managed runtime relies on analyses performed during JIT compilation which, like any JIT analysis, affects the execution time of the program. Consequently, to target efficiency, prior works involving JIT compilers resort to performing imprecise escape analyses that usually enable very few objects to be allocated on the stack. As an example, on average across several benchmark programs of interest, the JIT compiler of the Eclipse OpenJ9 VM [30] allocates only 0.16% of the total number of objects on the stack.

A promising alternative to imprecise JIT analyses could be to perform an aggressive escape analysis statically (ahead of time) and use its results to perform stack allocation during JIT compilation. This way, one could potentially identify a much larger number of objects to be stack allocated, without performing expensive analysis during JIT compilation. However, in presence of dynamic features supported by the language and the runtime, as well as differing libraries between the static analysis and the runtime, such a static+dynamic strategy may lead to erroneous stack allocation. As an instance, it is quite popular in the Java static-analysis community to resolve reflective calls using run-time logs [16]; if the call graph changes during execution from what was assumed statically, an object may get passed to a previously unknown method, potentially making it escape. Similarly, many managed runtimes provide features like dynamic classloading and hot-code replacement that allow new code to show up during program execution, thereby again leading to a potentially wrong stack allocation if the object escapes therein. This thesis proposes a sound and efficient approach of using static-analysis results to perform stack allocation in a Java VM, which is not marred by any dynamism that the language or the runtime may offer during program execution.

1.1 Optimistic Stack Allocation

The approach of object allocation proposed in this thesis is based on three novel ideas: *optimistic stack allocation*, *dynamic heapification*, and *stack ordering*, supported by corresponding precise static analyses. To begin with, the first idea is for the VM to optimistically allocate a list of objects, generated by a static escape analysis, on the stack frames of the methods it JIT-compiles. This lets us harness the results of a precise escape analysis for performing enhanced stack allocation. The second idea is to timely detect, using a run-time check and a "stack walk", if any objects are allocated on the stack incorrectly by the optimistic scheme; and if so, then to move them onto the heap (along with correction of references). This dynamic heapification allows us to ensure that the optimism does not lead to unsoundness in terms of memory allocation and access. Finally, the third idea uses points-to graphs [70] to statically suggest an "order" for allocating non-escaping objects on the stack during execution. This helps us make a subtle improvement to the heapification checks, in a way that the VM can skip expensive stack walks a majority of

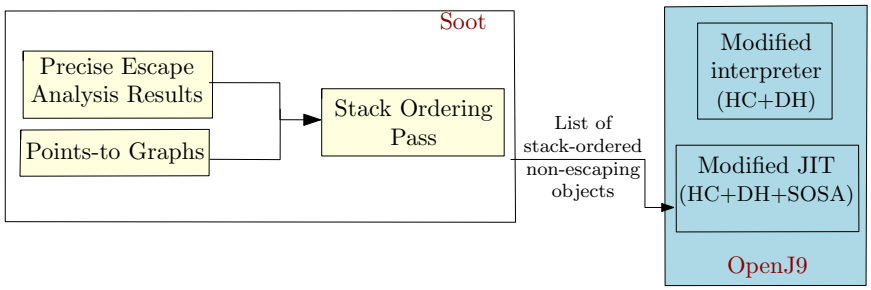


Figure 1.1: Block diagram representing static+dynamic approach for optimistic stack allocation. Abbreviations indicate modifications done in OpenJ9 towards: (i) HC: insertion of heapification checks; (ii) DH: dynamic heapification; (iii) SOSA: stack-ordered stack allocation.

times and instead be done with a simple condition-check to determine the requirement of heapification. Note that a scheme like this hinges on two primary requirements: (i) that any chances of an object being incorrectly on the stack be detected (and the repairs be done) before making any dereference – ensuring functional correctness; and (ii) that any run-time checks be as cheap as possible – to ensure efficiency. The optimistic scheme satisfies these requirements across the tiered infrastructure of a production JVM and ensures that the optimism leads to as many benefits as possible (in terms of stack allocation) using the static analysis.

Figure 1.1 illustrates the key implementation components of our optimistic approach. The static analysis is implemented over the Jimple IR of the Soot analysis framework [67]. In particular, it has (i) a context-, flow- and field-sensitive escape analysis that uses points-to graphs to generate a per-method list of non-escaping objects; and (ii) a stack ordering phase that traverses the points-to graphs to determine efficient stack orders for non-escaping objects. The run-time components are implemented across the tiered infrastructure of the Eclipse OpenJ9 Java Virtual Machine. In particular, the modified OpenJ9 reads static analysis results and uses them to perform optimistic stack allocation. The run-time heapification checks (HC) and dynamic heapification (DH) routines are inserted as part of the codegen phases of the JIT compiler as well as in the interpreter, in the form of new opcodes whose semantics enable the required operations. In addition, the optimistic approach modifies the stack allocation routine of the JIT compiler to allocate objects on the activation record of a method as per the (partial) stack order emitted by the static analyzer (SOSA).

The thesis evaluates various aspects of the optimistic approach over a series of benchmarks from the DaCapo [14] and the SPECjvm [59] suites. Firstly, it shows that the optimistic approach significantly increases the number of objects allocated on stack (on average, by 71%) as well as the number of bytes allocated on stack (on average, by 54%). Second, it shows that the overhead of the run-time checks is bypassed by the performance improvement brought by additional stack allocation. The thesis also presents further result by reducing the heap memory made available to the JVM, and observes that the improvement with optimistic approach translates into fewer GC cycles (on average, 5.3% less), as well as a decent improvement in performance in low-memory environments (on average, 8.8%), for benchmarks with high stack allocation. Overall, the evaluation of the optimistic approach demonstrates the enhanced precision, coupled with high performance, that the proposed approach could impart to existing JIT pipelines.

1.2 Combining Speculation and Static Analysis

Managed runtimes have multiple tiers of translation, comprising interpreters as well as baseline and optimizing just-in-time (JIT) compilers. As JIT compilers cannot afford performing complex program analysis like ahead-of-time (AOT) compilers, their tour-de-force is to generate fast paths of the code being compiled by specializing it to the current execution. These specializations are achieved by maintaining run-time profiles, which record a plethora of information related to call sites, branches, receiver types, and so on. An optimizing JIT compiler uses these profiles to perform various *speculative* optimizations, which often lead to performant binaries.

In contrast, recognizing the imprecision in standard JIT analyses, many recent works have proposed the usage of static AOT analyses to deliver precise optimizations in JIT compilers. These range from usages of partial program analysis in absence of parts of the program [24, 64], to offline analysis of JIT-compiled code [72, 45], to feedback-based approaches that improve results over multiple executions [11]. In order to mitigate concerns that offline-analysis results may lead to incorrectness in presence of dynamic features or changes in program paths, some of these approaches require a closed-world assumption [72], whereas others (like ours) employ run-time techniques to detect issues and/or repair the state before they could alter the program's behavior [6, 34].

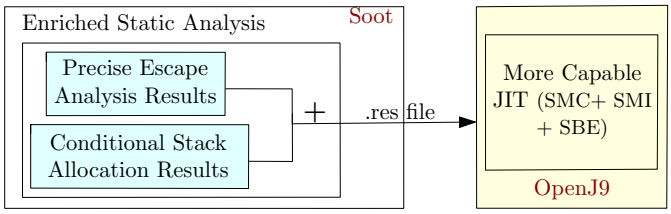


Figure 1.2: Block diagram of our approach `CoSSJIT`. Abbreviations indicate modifications done in OpenJ9 towards the resolution of conditions related to: (i) SMC: speculative polymorphic calls; (ii) SMI: speculative method inlining; and (iii) SBE: speculative branch execution.

In spite of the benefits of offloading complex program analysis to static time and using the generated results to achieve precision in a managed runtime, there are key differences between the precision capabilities of AOT and JIT compilers not captured by any prior work. In particular, just relying on a completely static program analysis can take away what a JIT compiler is better at – speculation based on run-time profiles. A static analysis could generate better results if it accounted for the possibility of speculation in the JIT compiler, thus allowing run-time profiles to be integrated into its otherwise precise results during JIT compilation. A manifestation of such an amalgam would result into the best of both the worlds, and enable the use of highly precise program-analysis results in driving aggressive optimizations during JIT compilation. The thesis proposes the first static+JIT strategy of this kind and implements it in a production setting, to perform an impactful optimization (stack allocation) much more aggressively than the state-of-the-art.

The proposed approach (called `CoSSJIT`) to integrate static escape analysis with run-time speculation, for performing aggressive stack allocation during JIT compilation, has two major components: (i) the "enriched" static analysis; and (ii) the more "capable" JIT compiler; see Figure 1.2. The first component is built over an existing formulation of static escape analysis [62], which implements the classic flow-, field- and context-sensitive pointer and escape analysis proposed by Whaley and Rinard [70, 68].

In order to enrich static analysis with the possibility of run-time speculation, the thesis first introduces *speculative conditions* while performing interprocedural dataflow analysis, at points where a JIT compiler might be able to generate more precise results than a static analyzer. These conditions are added as special values in the lattice of the underlying dataflow analysis, so that the analysis can generate and propagate these conditions apart from the dataflow values in the existing analysis lattice (which are *escapes* and *does-not-*

escape for escape analysis). In particular, for each abstract object in the escape-analysis domain, the static analysis generates results that may be conditional to three kinds of speculations that can be performed by the JIT compiler: *speculative polymorphic call*, *speculative method inline*, and *speculative branch execution*.

Speculative polymorphic-call conditions allow an object to be stack allocated if the set of likely callees at a polymorphic call site is known not to let the passed object escape. *Speculative method-inline conditions* allow an object escaping from a callee to be allocated on the caller's stack if the callee is inlined into the caller and the caller is known not to let it escape. *Speculative branch-execution conditions* allow an object to be stack allocated if it escapes only in branches that are determined to be taken less frequently than others. This thesis describes each of these conditions along with examples in Chapter 4, and implements them in the Soot framework [67].

The second component is the usage of speculative static-analysis results in the optimizing JIT compiler of a production JVM. The idea is to extend the JIT's escape analysis to consult the static-analysis results, and to hook static-analysis results to the JIT compiler's speculation infrastructure. The resolution of speculative method-call and method-inline conditions depends on the dynamic class-hierarchy table (maintained by the JVM to resolve calls and perform type-based optimizations), as well as the receiver profiles at call sites (maintained by the JVM to optimize polymorphic calls). Similarly, the resolution of speculative branch-execution conditions depends on branch or basic-block profiles (maintained by the JVM to perform more aggressive optimization of program hot-spots and for hot-code basic-block ordering). For the speculative conditions that depend on a profile value, there is a tunable "speculation threshold" that determines the profile percentage which is good enough for driving speculative stack allocation. In order to detect and repair incorrect speculative allocations, the thesis utilizes the idea of *heapification* (proposed in the previous section) that uses run-time checks to identify affected objects, and to copy them onto the heap while updating their references. The implementation of all the components associated with this contribution is in the Testarossa JIT compiler of the Eclipse OpenJ9 VM [30].

This thesis evaluates the `CoSSJIT` approach by comparing it with the baseline escape analysis of the Eclipse OpenJ9 VM over benchmarks from the DaCapo 23.11 [13] and 9.12 [14] suites, as well as from SPECjvm2008 [59]. `CoSSJIT` finds that the approach sig-

nificantly increases the amount of stack allocation performed by the JVM (5.7× overall), which translates to decent performance improvements (up to 20% and on average 6.7% across benchmarks with improved stack allocation, and without any noticeable degradation for the rest). The thesis also studies the effects of different kinds of speculative stack allocations handled by `CoSSJIT`, and finds that though they are essentially program-dependent, speculative method-inline conditions are the most impactful over different benchmarks. Finally, the thesis provides a measure of the overheads of the `CoSSJIT` approach in terms of additional time spent during JIT compilation, and finds them to be minimal (0.75%) compared to the huge benefits obtained therein.

1.3 Tuning the VM for Static+Dynamic Analysis

Using static analysis results along with speculative conditions in a managed runtime enables the JIT compiler to perform much more aggressive stack allocation. The JIT compiler can now leverage the speculative conditions and runtime information to be able to allocate more objects on the stack. However, to fully maximize the benefits of static analysis in a managed runtime, this thesis observes that the JIT compiler can be tuned further. Toward that end, this thesis proposes three enhancements for effectively combining static and dynamic analysis, in a JVM.

As the first enhancement, this thesis introduces enhancements to the inlining heuristic used by the JIT compiler when deciding whether to inline a method. The proposed heuristic incorporates information from static analysis by assigning weight to the potential for stack allocation enabled by inlining a given method. This introduces an additional criterion into the existing set of inlining heuristics. Specifically, at a call site with multiple candidate callees, the heuristic evaluates each callee on the number of objects that can be allocated on the caller's stack frame after inlining, as indicated by the static analysis. This allows inlining decisions to account explicitly for the optimization benefits arising from increased opportunities for stack allocation.

Second, the thesis proposes adapting static analysis results to adapt to runtime changes and accordingly update stack allocation decisions. This ensures that the analysis is not fixed but remains flexible enough to accommodate dynamic behavior during execution. For example, the `CoSSJIT` approach relies on statically generated inlining results for a

callee within the context of its caller at a given call site. Consider a method foo for which analysis results are available for a call to bar; if bar is inlined into foo, these results can be directly applied. However, if foo itself is subsequently inlined into another method (e.g., main), potentially along with bar, the original static analysis becomes contextually incomplete. In such scenarios, the analysis results associated with foo—including those pertaining to bar—should be propagated to the new calling context. Therefore, the analysis result must be updated to reflect this extended inlining context, ensuring that the previously computed conditional inlining information remains applicable and continues to guide stack allocation decisions effectively. To support this, the thesis maintains additional object-level information about allocating methods, enabling the static analysis results to be incrementally refined and propagated in response to run-time inlining decisions.

Finally, this thesis proposes a third enhancement to ensure that static analysis results are utilized even when a method is JIT-compiled at an early stage. Escape analysis is typically an expensive analysis, and JIT compilers often delay it until a method exceeds certain invocation thresholds. As a result, methods compiled earlier may not benefit from stack allocation and related optimizations. However, static analysis results may already be available for such methods. To leverage this information, the thesis proposes an additional pass that enables stack allocation based solely on static analysis results, independent of dynamic escape analysis. This allows early-stage compiled methods to also benefit from stack allocation. The proposed approach is implemented at the "cold" optimization level of the Testarossa JIT compiler in the OpenJ9 virtual machine.

The thesis evaluates all the proposed enhancements over benchmarks from the DaCapo 23.11 [13] and 9.12 [14] suites, as well as from SPECjvm2008 [59], and demonstrates that each of them results lead to further increase in stack allocation.

1.4 Formalizing Static+Dynamic Analysis

Building on the staged compilation framework and the challenges discussed earlier, this thesis lately presents the static+dynamic analysis scheme from a more principled perspective. The holy grail in the space of precise program analyses is the ability to analyze the whole program. However, in case of languages such as Java and C#, where the complete program is available only during run-time (e.g., in Java Virtual Machines), performing

```
1  class A {                          7  class B extends A {
2     void foo(A a1) {                8     void bar(A q) {...}
3        A a2 = new A(); // O3        9  }
4        a1.bar(a2);
5     }
6     void bar(A p) {...} }
```

Figure 1.3: A Java code snippet to demonstrate generation of dependencies.

whole-program dataflow analysis during just-in-time (JIT) compilation is prohibitively expensive. On the other hand, owing to the separate compilation assumption [2], it is possible to "partially analyze" various parts of the program statically.

The program analysis technique used in compilation systems where the whole program is not available for analysis is called *partial analysis* [24, 64, 46, 20] . Traditionally, this means generating analysis results without the ability to incorporate the effects of the unavailable parts of the program, which again loses precision. However, for languages like Java and C# where program translation is spread across static and JIT compilation, the idea of "staging" the program analysis itself across the static and the dynamic phases of compilation looks promising. Recent approaches [64, 57] use this idea to statically generate "dependencies" from various elements of the known program to the unknown parts of the program, which are then resolved during JIT compilation. As an example, consider the Java code snippet shown in Figure 1.3; say the object(s) allocated at line l are represented using the abstract object O_l. Here, what happens to the object O_3 depends on what happens to the respective first parameters in methods `A.bar` and `B.bar`. Assuming all the code of class `A` is available statically whereas that of class `B` is available only during run-time, we can resolve the dependencies related to `A.bar` statically, but for `B.bar` only during run-time. The idea behind staging of this kind is to generate such dependencies, resolve as many of them as possible statically, and then complete the analysis results by resolving the residual dependencies during run-time. A point worth noting, however, is that this promising approach has stark similarities with the idea of *partial evaluation* [37].

Observe that staging a whole-program analysis based on prior evaluation of static dependencies and residual evaluation of dynamic dependencies, as described in the previous sections and as illustrated in Figure 1.4, would require a special component (say a

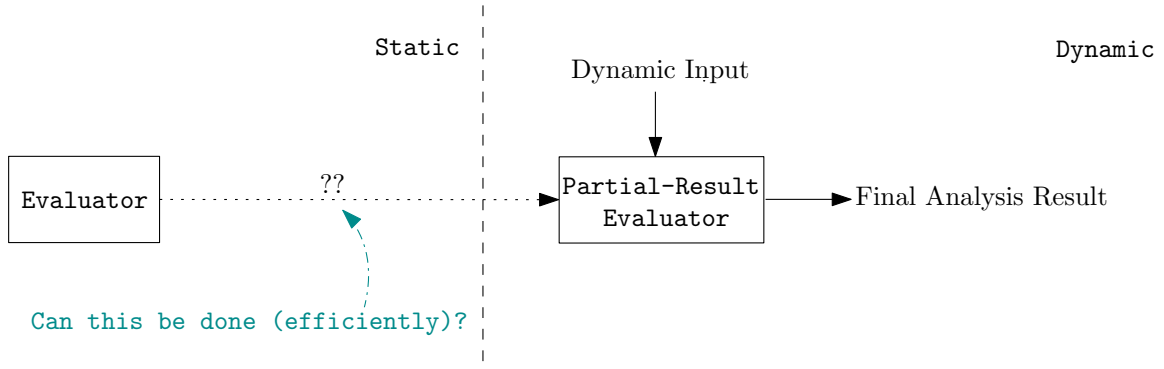


Figure 1.4: Generation of partial-result evaluators.

"partial-result evaluator"), which is capable of consuming dynamic inputs and completing the analysis results. An important question that begs an answer here is whether and how could one generate such special evaluators that can "process partial results". Further, in order to hold the efficiency advantages of staging, it is important that the generation of such evaluators is itself efficient and if possible, offloaded to the static compiler. The next question thence is, can we design a "generator" that efficiently generates partial-result evaluators, given the standard evaluator for a particular whole-program analysis. Finally, assuming these components exist, it is interesting to ask if the staged analysis would generate the same result as the corresponding whole-program analysis. The thesis answers all these questions with a strong affirmation by modeling the staging scheme based on the classic theory of partial evaluation.

The thesis first formulates whole-program dataflow analysis as the process of computing and resolving dependencies of various elements on different parts of the program. Followed by this, it defines partial analysis as the process of partially evaluating those dependencies with respect to the statically available parts of a program, thus generating partial results for the analysis being performed. It then further devises a strategy (called *first AM projection*) to "generate" an evaluator that can process these partial results by performing residual resolution using the evaluated values of dynamically available dependencies. Later, to improve the efficiency of generating such partial-result evaluators for different analyses, the thesis proposes a series of specializations (called *second and third AM projections*). Finally, illuminating the similarities between the model of partial analysis and the theory of partial evaluation, the thesis proves that the results generated by an analysis staged using the proposed scheme would be the same as the ones generated by its whole-program version.

As an extension that shows how a staging scheme proposed in thesis can be used to handle dynamic features that violate a *closed-world assumption* in modern virtual-machine based runtimes, the thesis show how the idea of staging can be used to precisely handle callbacks made from Java libraries to application methods. As callbacks may introduce new edges in the call-graph of an application, parts of the program that depend on possible callbacks are first identified statically, and such dependencies are then encoded in addition to the standard dynamic dependencies from the application to libraries. The thesis shows how such *callback dependencies* still fit into the staging model, by carving out a third stage – again based on partial evaluation – that could resolve them while running the program (as against the resolution of other dynamic dependencies during JIT compilation).

In order to validate the concepts presented, the thesis provides a prototype implementation of the different components of the staging scheme. Specifically, the thesis first designs a simple evaluator that could resolve a set of dependencies and generate the analysis result for a given program element. The evaluator is written in a way that in case of staged analysis (where only static dependencies can be resolved initially), it generates a partial result. Next the thesis implements a specializer that takes the above evaluator and specializes it with the given partial result, to generate the partial-result evaluator. This partial-result evaluator can take evaluated values of dynamic dependencies as input and generate the final analysis result(s). Notably, such partial-result evaluators are independent of the way dependencies are generated for a particular analysis, agnostic of the tiered nature of modern managed runtimes, and can be invoked as soon as the values of dynamic dependencies are available. The prototype demonstrates this by using dependencies generated for escape analysis [15, 62] of Java programs, generating partial-result evaluators for abstract objects therein, and invoking them for dependencies on methods from the Java class library. Furthermore, though the example analysis is based on method summaries, the idea of staging can be applied to other models of program analysis as well (as long as the dependencies can be categorized into static and dynamic components) and extended to other dynamic features (as long as the dependence can be resolved during run-time).

Having described partial-result evaluators and their generators in the said form (which describes standard staging), this thesis observes that certain ways modern tiered runtimes (such as JVMs) operate raise few more interesting questions. For example: “Can residual dependencies be always resolved?” “What if at certain execution points all the dynamic

inputs are not available?" "In case resolution cannot proceed, can we generate a more precise value than falling back to the most conservative solution?" This thesis finally discusses possible directions to address these questions, along with drawing connections with a few other ways of performing program analysis, as interesting future extensions to the foundational model of staged partial analysis proposed therein.

1.5 Contributions made by the Thesis

The main contributions of this thesis are listed below.

(i) Sound and efficient approach for performing aggressive stack allocation:

- This thesis proposes a first-of-its-kind static+dynamic strategy to optimistically perform stack allocation in Java VMs, harnessing the results of context-, flow- and field-sensitive escape analysis.
- This thesis provides a run-time algorithm based on stack walks, implemented across the tiered infrastructure of a real-world JVM, to timely detect and move wrongly stack allocated objects to the heap, while correcting their references.
- This thesis introduces a novel idea of ordering non-escaping objects on the stack to reduce the time spent in stack walks and consequently the overhead of run-time heapification checks.
- This thesis performs a thorough evaluation to measure the efficacy of combining static analyses with run-time strategies in improving the amount of stack allocation that a JIT can perform, and consequently the run-time performance, in allocation-intensive programs.

(ii) Combining static analysis and speculation in a JIT compiler:

- This thesis introduces `CoSSJIT`, an idea that enriches static analysis to account for the possibility of run-time speculation into its results.
- This thesis instantiates the idea for speculatively allocating objects on stack in three scenarios: (i) An object can be allocated on its allocating method's stack frame when it does not escape at a polymorphic call site based on run-time profile information.

(ii) A callee object can be allocated on the caller's stack frame when the callee method is inlined into the caller and the object does not escape further. (iii) A method-local object that does not escape in some of the conditional branches can be allocated on stack if those branches are more likely. Incorrect speculative stack allocations are dealt through the idea of dynamic heapification.

- This thesis provides an implementation of the proposed integration of speculation-enriched static analysis results and speculative stack allocation across the tiered infrastructure of a production JVM.

- This thesis presents a comprehensive evaluation demonstrating that enhancing static analysis with speculative results increases the extent of stack allocation performed by a JIT compiler. This, in turn, improves run-time performance in allocation-intensive programs.

(iii) Tuning the VM for for static+dynamic analysis:

- This thesis presents techniques for tuning the VM to maximize the benefits of static+dynamic scheme. It introduces three enhancements (i) extending the JIT compiler's inlining heuristic to incorporate statically generated stack allocation results; (ii) enabling static analysis to adapt to runtime changes and update its results accordingly; and (iii) ensuring that static analysis results are utilized at lower optimization levels as well.

(iv) Formalizing staged static+dynamic analysis:

- This thesis formalizes the definition of partial analysis and devises a scheme to stage whole-program analyses into static and dynamic components, along with a novel notion of AM projections.

- This thesis establishes the correctness and precision properties of the proposed staging scheme, based on results from the theory of partial evaluation.

- This thesis extends the staging scheme to handle Java callbacks, as an example of supporting dynamic features in modern tiered runtimes.

- This thesis validates the presented concepts by implementing a prototype that generates partial-result evaluators independent of the analysis under consideration.

1.6 Organization of the Thesis

The rest of the thesis is organized as follows. Chapter 2 gives an overview of relevant concepts from an existing staging framework that are necessary for understanding the remainder of the thesis. It begins by introducing managed runtimes, using OpenJ9 as a representative example. The chapter then explains how JIT compilers within managed runtimes leverage run-time information to perform speculative optimizations. The chapter also presents background on escape analysis and its associated optimizations, namely stack allocation and scalar replacement. Finally, the chapter outlines the general principles of the PYE framework, including the generation and evaluation of conditional values through an illustrative example, and discusses its scope and applicability to other analyses.

Chapter 3 describes the ideas of optimistic stack allocation and dynamic heapification which provides a sound and efficient way of using static analysis in a managed runtime. The chapter first illustrates the interesting challenges that the proposed approach handles, with a motivating example. Then the chapter describes the static analysis implementation and the key run-time components of the proposed approach: optimistic stack allocation, run-time checks for heapification, and recursive dynamic heapification using stack walks. This is followed by a novel idea of determining and using stack orders to reduce the number of stack walks toward reducing the overhead of run-time checks. Later, the chapter also states the correctness criterion of the proposed approach, and discusses few design choices made while implementing it for a full language in an industry VM. Finally, the chapter evaluates the improvements imparted by the proposed approach in terms of stack allocation, performance in constrained memory environments, as well as the contribution of the key ideas in imparting efficiency.

Chapter 4 describes `CoSSJIT`, that introduces the idea of combining static analysis and speculation in a managed runtime. The chapter first illustrates the interesting challenges that the proposed approach handles, using a motivating example. Then, the chapter describes the static component of the approach in terms of enriching a static escape analysis to account for possible speculations during JIT compilation. This is followed by describing the key run-time components of the `CoSSJIT` approach for making decisions about allocating objects on stack speculatively, using dynamic class hierarchies, run-time profiles, and method-inlining information. Finally, the chapter provides an extensive

evaluation on the improvements imparted by the `CoSSJIT` approach in terms of increase in stack allocation, impact on performance and garbage collection, as well as the contribution of the key ideas in delivering precision.

Chapter 5 discusses some of the tuning and adjustments required in managed runtimes for getting the best out of a static+dynamic scheme. The chapter first describes inlining heuristics used in the OpenJ9 VM and discusses how to enrich the inlining heuristics based on the static analysis results for performing more aggressive stack allocation. The chapter then presents a way to make static analysis results more adaptive to run-time changes. Finally, the chapter describes how to take early advantage of the static analysis results by utilizing them at lower optimization levels as well. Finally, the chapter presents the improved results (in terms of stack allocation) for the proposed enhancements and concludes with a discussion on improvements achieved through these enhancements.

Chapter 6 describes the formalism behind the idea of staged static+dynamic program analysis. The chapter first describes partial evaluation and Futamura projections in a readily comprehensible manner. Then, the chapter presents the staging scheme for whole-program dataflow analysis, along with the AM projections for generating partial-result evaluators. This is followed by describing the details of the prototype implementation to validate the presented concepts. The chapter then describes how to extend the staging scheme to handle Java callbacks. Finally, the chapter provides an evaluation of the prototypes and highlights a few of the challenges posed by contemporary programming-language runtimes, possible ways to deal with them, as well as connections of the staging scheme with few other ways of performing program analysis.

Chapter 7 discusses prior and related work, along with the ideas of applying staged static analyses in a managed runtime. Finally, Chapter 8 concludes the thesis, discusses some of the limitations of using staged static+dynamic analysis, and outlines potential directions for future research in this area.

Chapter 2

Background

This section introduces few key concepts and background details that are relevant for understanding the thesis. The background starts by providing general motivation for managed runtimes and outlining their advantages. It then presents an overview of the state-of-the-art OpenJ9 virtual machine that serves as the target runtime for the implementation described in this thesis. Further, the chapter describes how a JIT compiler typically performs speculative optimizations based on the run-time profile information and deoptimizes if the speculation fails during execution. The chapter next describes escape analysis and its role for enabling memory optimizations such as stack allocation. The chapter finishes with an overview of using static analysis results in a managed runtime proposed by the PYE framework, including the generation of dependencies in the form of conditional values, their evaluation, and their scope and applicability.

2.1 Managed Runtimes

Managed runtimes are execution environments that handle many low-level responsibilities of running a program so developers do not have to manage them manually. Instead of dealing directly with memory allocation, deallocation, and system-level details, a typical managed runtime provides services like automatic garbage collection, type safety, exception handling, and often just-in-time (JIT) compilation to optimize performance at runtime. This abstraction makes programs safer and easier to write, since issues like memory leaks and buffer overflows can be taken care of by the runtime. By leveraging the

execution profiles for performing speculative optimizations, JIT compilers enable managed runtimes to outperform statically compiled code in many scenarios. Furthermore, managed runtimes often include built-in support for features such as reflection, dynamic class loading and concurrency abstractions, which simplify the development of complex applications. Popular programming languages that use a managed runtime environment include Java, C#, Scala, JavaScript etc. Popular managed runtimes include Java Virtual Machine and the .NET Common Language Runtime, both of which allow code to run in a platform-independent manner while still achieving efficient execution through JIT optimizations. Overall, the managed runtime for any language strikes a balance between abstraction and performance by integrating ease of use with run-time optimizations.

2.2 Eclipse OpenJ9

Eclipse OpenJ9 [30] is a popular, industry-standard open source JVM implementation, used by many performance-critical applications. It is built using the OMR framework [29], which is a set of open-source components that can be used to build robust language runtimes, and executes Java Bytecode using a multi-tier interpretation and compilation mechanism. OpenJ9 comprises of a JIT compiler, Testarossa [30](TR-JIT), which converts Java Bytecode to a Tree intermediate language (Tree IL) and performs optimizations tiered across multiple optimization levels. It uses multiple levels of compilation: noOpt, cold, warm, hot, veryhot and scorching, in order of increasing aggressiveness of the performed optimizations. A method being JIT compiled may travel across one of the compilation levels, wherein different analyses and optimizations may be performed at different levels.

At higher optimization levels, TR-JIT also performs various OO optimizations such as stack allocation and object scalarization (also called non-contiguous stack allocation), enabled by a primarily intraprocedural escape analysis (see Section 2.5 for details) that sometimes can "peek" inside called methods at monomorphic call sites. However, the analysis is only partially interprocedural and context insensitive, and in general, marks objects passed to methods beyond the "peek depth" as escaping. The TR-JIT compiler of OpenJ9 also adapts the idea of partial escape analysis [60] to stack-allocate objects in hot paths and "heapify" them in cold blocks. Other JITs such as Oracle's C2 compiler and GraalVM's Graal compiler have similar capabilities built-in too, which we mention

to assert that the contributions we make in this thesis can be expected to provide similar improvements therein.

As we show later in our thesis (Section 3.5), the typical analysis imprecision discussed above leads to a very small number of objects being actually allocated on method stacks during program execution. The compiler is even more conservative in performing stack allocation (as well as other analyses and optimizations) when dynamic features such as hot-code replacement are allowed by the VM. In this thesis, we propose an approach that can use static analysis results to perform aggressive stack allocation of method-local objects in Java programs, for the Eclipse OpenJ9 VM, while being robust to the dynamic nature of the language.

2.3 Speculation during JIT Compilation

JIT compilers are known to be resource constrained as the compilation time directly affects the execution time of the program being compiled. Consequently, they end up performing imprecise program analyses and miss optimization opportunities, instead of say performing flow-sensitive or interprocedural analyses. However, in contrast to static analyses performed by ahead-of-time compilers, JIT compilers have access to run-time profiles computed either in a previous interpreter pass or while executing code compiled at a lower optimization level. These profiles typically correspond to information such as the popular types of the objects collected as parameters, the branches taken at conditionals or switch cases, and the receiver types or the methods bound at polymorphic call sites [27].

Based on run-time profiles, JIT compilers can make assumptions to perform several optimizations and generate highly specialized code pertaining to the assumptions. Such optimizations are called *speculative optimizations*, whereby the compiled code is protected from unsoundness by anchoring it to "guards" that check the assumptions. Typical run-times perform speculative optimizations such as method inlining, inline caching (based on callsite profiles) and aggressive optimization of program hotspots (based on branch-target profiles). These can often enable many more optimizations in the specialized code, ranging from better constant propagation to better escape analysis and stack allocation (example, a JVM may be able to stack allocate an object whose reference is passed to a method even with an intraprocedural escape analysis, if that method is inlined into the caller).

2.4 Deoptimization during JIT Compilation

When JIT compiler performs speculative optimization based on assumptions derived from the runtime profile, there remains the possibility that speculation could go wrong in a later run of the compiled code, for example type mismatch when optimization is performed on some speculative type. In such scenarios the VM must invalidate the optimized code and fall back to some safe state, this process is called *deoptimization* [36]. The most common fallback strategy used by many virtual machines is to discard the optimized code and resume execution in the interpreter.

To ensure correct deoptimization, the JIT compiler maintains multiple safepoints during execution. At each safepoint, it records detailed metadata required for reconstruction of the program state, including the number of local variables, their values and types, and the state of the operand stack. This information enables the virtual machine to accurately restore execution in a non-optimized context when deoptimization is triggered.

2.5 Escape Analysis and Stack Allocation

Escape analysis [18] is a popular program-analysis technique employed by many JIT compilers to determine objects whose references are not accessible outside their scope of allocation. If a reference to an object is never required beyond accessing its fields, it can often be decomposed into scalar variables (an optimization called *scalar replacement* [41, 60]), and if the object cannot be accessed once the allocating method's lifetime is over then it can be allocated on stack instead of on the heap (an optimization called *stack allocation* [15, 18, 70]). Typical JIT compilers determine the "escape status" of an object based on a reachability test from external and global references over the points-to graph [70] of a method.

If an object can be allocated on stack, its fields can be accessed faster using constant offsets through the stack pointer instead of via indirections in the heap. In addition, a stack-allocated object is not only faster to allocate in the first place, it gets freed away with no cost as soon as the activation record of the allocating method is popped off the run-time stack. For managed languages such as Java, this also leads to reduced garbage collection. Expectedly, an increase in stack allocation has been shown to significantly

improve performance of Java-based systems, and hence it is employed by most Java JIT compilers at their higher optimization levels.

If an object is eligible for scalar replacement then it is replaced by its constituent fields and the object header is removed. In the popular HotSpot VM, this is performed based on a reachability test over the connection-graph representation [18] of the methods being compiled at higher optimization levels of the C2 (Server) JIT compiler [53]. Similarly, the JIT compiler of GraalVM [51] performs scalar replacement aggressively, and *materializes* [60] the fields on the heap in branches where the corresponding object is found to escape.

2.6 Static Analysis for Managed Runtimes

Using statically generated analysis results to guide optimizations is not a conventional approach in managed runtimes; the PYE (Precise-Yet-Efficient) framework [64] describes a way to utilize the results of a precise and time-consuming static analysis while still achieving efficient runtimes. The major motivation is that rather than doing precise analyses during JIT compilation (done rarely due to direct impact on execution time), one could analyze Java Bytecode offline and pass on the results to the JVM. The JVM, during JIT compilation of a given method, uses the results obtained via static analysis to perform aggressive optimizations. The approach described by PYE proposes that it can be used as a general framework that statically generates *conditional values* to denote dependencies across various parts of a program, and then resolves these dependencies to generate final analysis results during JIT compilation. The work showed the efficacy of this approach by eliminating unnecessary synchronization constructs and eliding redundant null checks, implemented in the OpenJDK HotSpot JVM.

However, a major limitation of the PYE approach is that one needs to assume that the results obtained statically correctly reflect the program state in the VM. Thus, the performed optimizations may go wrong if the static analysis was incorrect; similarly, in presence of dynamic features such as dynamic class-loading (which allows to load a new class dynamically) or hot-code replacement (which allows the program to change dynamically), an optimization performed using the PYE framework may have to be invalidated.

2.6.1 Dependencies and Conditional Values

Generating dependencies among the various elements of a program is a classical way of performing program analyses [38]. PYE uses the concept of partial analysis [24] to generate partial results for all the statically analyzable parts of a program, and uses those results during run-time to generate the final result. In order to account for the unavailability of libraries while analyzing applications (and vice-versa) without losing precision, PYE generates *dependencies* across the elements of a program as *conditional values* statically, and evaluates them during run-time. We next describe the generation and evaluation of such conditional values, along with a representation that fits in with the notations that we use throughout this thesis.

2.6.2 Generation of Conditional Values

Given a method m in a program P, a traditional whole-program analysis ψ generates a summary f_m mapping each program element $x \in m$ in the domain $\mathcal{D}$ of the analysis to one of the values in the set of dataflow values `Val` for that analysis. Thus, $f_m(x)$ denotes the *analysis result* for the element x present in method m. As an instance, for escape analysis [15], the set $\mathcal{D}$ could consist of all the abstract objects (e.g. O_3, O_4 and O_6 in the method `A.foo` in Figure 2.1, based on an allocation-site abstraction), and the set `Val` could be $\{D, E\}$, denoting *DoesNotEscape* and *Escapes*. These values form a lattice with D or $\top$ being the most precise and E or $\bot$ being the least precise elements, respectively.

In contrast, let $g_m(x)$ represent the set of conditional values for a program element x present in method m. A conditional value denotes dependence on another program element, and is defined as a 3-length tuple $\langle \Theta, v, v' \rangle$, where $\Theta = \langle n, y \rangle$ represents the dependee element y in method n, and v and v' are values from the lattice `Val` of the traditional analysis. A conditional value $\langle \langle n, y \rangle, v, v' \rangle$ can be evaluated to obtain v' if the analysis result $f_n(y)$ equals v. Additionally, in case of lattices with complementary values (such as the one described above for escape analysis), the last two elements of a conditional value can be skipped to just store the dependee element $\langle n, y \rangle$; we use this notation sometimes for brevity.

For the code shown in Figure 2.1, if the analysis ψ being performed is escape analysis, then the set $g_{\texttt{A.foo}}(O_4)$ of conditional values that determines the escape status of the

```
1  class A {
2     void foo(B b) {
3        A a1 = new A(); // Object O3
4        A a2 = new A(); // Object O4
5        a1.bar(a2);
6        L l1 = new L(); // Object O6
7        l1.lib(a2); }
8     void  bar(A p1) {
9        // p1  remains non escaping
10    } }
```

```
1  class L {
2     // A library class
3     void lib(A r1) {
4        // A library method
5        ...
```

Figure 2.1: A Java code snippet to demonstrate static+JIT analysis. Class `L` is a library class not available during the analysis of the application class `A`.

abstract object O_4 allocated at line 4 is:

$$g_{\texttt{A.foo}}(O_4) = \{\langle\langle \texttt{A.bar}, \texttt{p1}\rangle, D, D\rangle, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, D, D\rangle, \\ \langle\langle \texttt{A.bar}, \texttt{p1}\rangle, E, E\rangle, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, E, E\rangle\} \tag{2.1}$$

Here, the set of conditional values indicates that the escape status of O_4 depends on the escape statuses of the pointees of the first parameters `p1` and `r1` of the methods `A.bar` and `L.lib`, respectively. The conditional values denoting dependence on class `A` can be resolved statically, whereas those depending on the library class `L` are resolved at run-time. Note that we are directly using the names of parameters (that is, `p1` and `r1`) in the conditional values for brevity; in practice they would be placeholders representing the first parameter of the corresponding method.

2.6.3 Evaluation of Conditional Values

Given a set $g_m(x)$ of conditional values generated for an analysis ψ, PYE evaluates them at run-time using an *evaluator*; we denote the same as $\texttt{CEval}_\psi$ (see Figure 2.2). $\texttt{CEval}_\psi$ takes $g_m(x)$ along with the analysis results for all the dependencies contained therein ($\texttt{IN}_{g_m(x)}$ in Figure 2.2), and generates $f_m(x)$ as follows:

$$f_m(x) = \sqcap_{\mathcal{T} \in g_{m(x)}} [\![\mathcal{T}]\!] \tag{2.2}$$

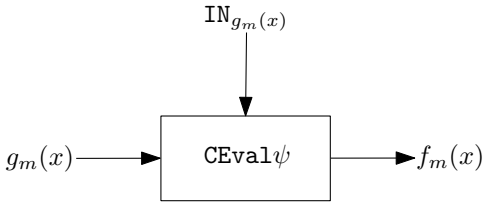


Figure 2.2: Whole-program analysis.

where $\sqcap$ is the operator used to obtain the analysis result from the evaluated values of the dependencies and $\mathcal{T} = \langle\langle \texttt{n}, \texttt{y}\rangle, v, v'\rangle$ is resolved as:

$$[\![\langle\langle \texttt{n}, \texttt{y}\rangle, v, v'\rangle]\!] = (f_{\texttt{n}}(\texttt{y}) == v)\ ?\ v'\ :\ \top \tag{2.3}$$

Using Equations 2.2 and 2.3, $\texttt{CEval}_\psi$ can evaluate the conditional values for each program element and generate final results for the analysis ψ. Note that when a dependence in $g_m(x)$ cannot be evaluated statically (for example, due to the unavailability of the corresponding method's analysis results), PYE simply takes a union of the conditional values present therein, in order to remember the same for later stages.

For example, evaluating $\mathcal{T} = \langle\langle \texttt{A.bar}, \texttt{p1}\rangle, D, D\rangle$ in $g_{\texttt{A.foo}}(O_4)$ amounts to analyzing the method `A.bar` (to obtain $f_{\texttt{A.bar}}$), then looking up the escape status of the abstract object pointed-to by `p1` (i.e. the escape status $f_{\texttt{A.bar}}(O_{p1})$, if `p` points to O_{p_1}), and then checking if it equals D; if yes, then $\mathcal{T}$ gets evaluated to D, otherwise it is resolved to the most precise element of the analysis lattice (which also happens to be D for escape analysis). Finally, after evaluating each conditional value in $g_{\texttt{A.foo}}(O_4)$, we can use Equation 2.2 to compute the meet of the individual evaluated values, in order to obtain the final analysis result $f_{\texttt{A.foo}}(O_4)$.

2.6.4 Scope and Applicability of Conditional Values

In general, the approach of offloading complex analyses to static time and finishing the results during run-time can be used to perform any dataflow analysis with a finite lattice (to bound the number of conditional values), as long as (i) the analysis results for various program elements can be represented in the form of dependencies on other elements; and (ii) there is a possibility of resolving the residual dependencies while generating the executable for a program — straightforward in managed runtimes with JIT compilers,

and possible with the ability to modify code during linking in languages without a managed runtime. Additionally, we assume that the order of evaluation of the generated dependencies is irrelevant; this is the case when the meet operation is commutative.

In the past, the idea of conditional values considered in this thesis has been used to perform various non-trivial analyses and optimizations: escape analysis to elide synchronization and points-to analysis to elide null-checks [64], and even to perform dependence analysis for parallelizing loops [57]. The generation of these conditional values has been shown to be context-, flow- and field-sensitive, following a fixed-point computation model.

A noteworthy point about conditional values is that they propagate in the program as normal dataflow values in the lattice of the analysis under consideration, which means they can be used to model dependencies on different parts of the program and different properties of the analysis being performed. Thus, one could facilitate newer kinds of conditional values by extending the domain of program elements with new kinds of abstract objects; for example, a null-dereference check can be modeled with an abstract object representing the dereference, a synchronization statement can be modeled with an abstract object representing the monitor of the object being locked on, a conditional can be modeled with an abstract object representing the boolean conditional expression, and so on. In this thesis, we use the notion of conditional values not only for performing standard static escape analysis, but we also extend them with newer kinds of conditions to handle dynamic features in managed runtimes. Furthermore, we develop a model of staged static+dynamic partial analyses, which allows us to straightforwardly prove the correctness and precision of the idea of staging as discussed above.

Chapter 3

Optimistic Stack Allocation

This chapter presents an important optimization for object allocation in Java. Section 3.1 begins with a motivating example and outlines our approach, which uses static escape analysis to enable enhanced stack allocation in a Java VM, while ensuring soundness and efficiency. Section 3.2 describes the static analysis implemenetation details and then further explains the key run-time components of our approach *optimistic stack allocation*, *run-time checks for heapification*, and recursive *dynamic heapification* using stack walks. Later, Section 3.3 shows how we determine and use *stack orders* to decrease the number of stack walks towards reducing the overhead of run-time checks. Section 3.4 states the correctness criterion of our approach, and discusses few design choices that we made while implementing it for a full language in an industry VM. Finally, Section 3.5 presents a detailed evaluation of the improvements imparted by our approach in terms of stack allocation, performance in constrained memory environments, as well as the contribution of the key ideas in imparting efficiency.

3.1 Motivating Example

The primary goal of our approach is to propose a robust and efficient scheme that allows static escape analysis of Java Bytecode to be used "safely" for performing stack allocation in the JVM. In this scheme, we let the JVM *optimistically* allocate objects on the stack, using the information given by the static analysis, in such a way that the JVM is able to detect and handle incorrect stack allocations during run-time. In this section, we

```
1  class A { B g; }
2  class C {
3     A f;
4     void doo(C q, C r) {
5        C x = new C(); // O5
6        A y = new A(); // O6
7        x.f = new A(); // O7
8        A p = x.f;
9        x.f.g = new B(); // O9
10       r.zar(p, q);
11       bar(x, y);
12    } /* method doo */
11    void zar(A p, C q) {
12       q.f = new A(); // O14
13    } /* method zar */
14    void bar(C p1, A p2) {
15       p1.f = p2;
16    } /* method bar */
17 } /* class C */
18 class D extends C { /* dynamically loaded */
19    void zar(A p, C q) {
20       q.f = p;
21    } /* method zar */
22 } /* class D */
```

Figure 3.1: Motivating example to explain various aspects of our run-time scheme.

motivate the reader towards the subtle aspects of developing such a scheme, using a running example.

Consider the Java code snippet shown in Figure 3.1. The class A has a field g of type B and the class C has a field f of type A. Denoting the abstract object(s) allocated at line l as O_l, we can see that the reference variable x in the method doo points to the object O_5, object O_5's field f points to the object O_7, and object O_7's field g points to the object O_9. The reference variable p too points to the object O_7 (due to the field load at line 8), which is later passed to the method zar of class C (assuming class D is not known during static analysis). As zar does not make the objects pointed-to by its parameters escape, we can allocate the objects O_7 and O_9 on doo's stack frame, as shown in the left-hand side of Figure 3.2. Prima facie, note that this information required an interprocedural analysis (to incorporate the effect of zar into doo).

Now consider a scenario in which during program execution in the JVM, the method doo is called by passing an object of a dynamically loaded class D (defined at line 20) as the second argument. Here, as the variable r now points to an object of type D, the call site at line 10 would invoke the overridden method zar in D. However, this zar stores the object pointed-to by its parameter p into the field f of its parameter q (at line 20), which in turn points to an object passed to its caller (via the parameter q in doo). As a consequence of this store statement, the lifetime of O_7 becomes longer than that of its allocating method

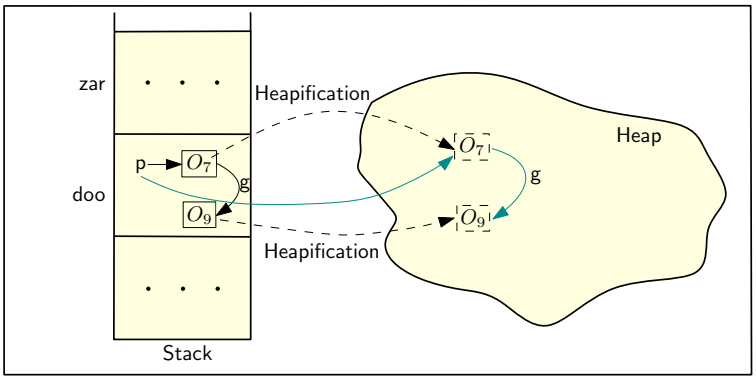


Figure 3.2: A snapshot of the run-time stack and the heap after dynamic classloading in Figure 3.1. Dashed edges represent heapification and green edges represent updated points-to edges post heapification.

doo, and thus O_7 cannot be kept on the stack frame of doo. Furthermore, the object O_9 is reachable from an escaping object O_7 and hence that too cannot be kept on the stack frame of doo. Letting these objects be on the stack is unsound. In order to ensure correctness, we need to first detect the erroneous stack allocation before anybody tries to access the corresponding object, and then ensure that the run-time memory layout is modified to avoid future incorrect accesses during the program's remaining execution. Also, as this needs to be performed on-the-fly during program execution, its efficiency is crucial. We next describe how we perform these operations in context of our motivating example from Figure 3.1.

3.1.1 Detecting Incorrect Stack Allocation

Recall that owing to dynamic classloading in the above discussed example, the object O_7 becomes non-stack-allocatable (i.e., escapes) at line 20 in D's zar. Thus, line 20 is the precise program point at which we could detect, just before storing O_7 into the field of a longer lifetime object (the one pointed-to by the parameter q), that we are going to make O_7 escape. In order to detect this occurrence, we insert a check (using a new opcode in the intermediate language of the JIT compiler) immediately before the write barrier associated with the store operation. This "heapification check" compares the lifetimes of the object being stored (say *rhs*) and the base object on the left-hand side of the assignment (say *lhs*), and thus decides whether we could have wrongly allocated *rhs* on the stack. Note

that the lifetime-comparison check needs to be performed for any statement that may cause an object to escape, along with the more common store statement discussed above.

3.1.2 Moving Objects and Correcting References

Once we have detected that an object is wrongly allocated on the stack, we need to move it onto the heap (we call this operation *heapification*). Moving the object from the stack to the heap is as simple as creating a copy of the object (along with copying the values of its fields) and allocating the new object onto the heap by calling the VM's memory allocation routine. However, there are two more subtle issues to consider. First, the fields of the object being copied may be pointing to other non-primitive objects; for example, the object O_9 pointed-to by the field `g` of the object O_7. This means the heapification operation needs to be repeatedly performed until it has copied all the reachable objects onto the heap. Hence, once we detect an incorrectly stack allocated object, we invoke a "recursive heapification" routine that allocates all the reachable objects that are currently on the stack onto the heap. Second, there may be stale references to the object being heapified (on stack or in registers); for example, after this heapification is done, there remains a stale reference from the variable `p` in the method `doo` to the O_7 on the stack. Such references need to be corrected to point to the right object. We perform this "reference correction" by maintaining a map containing the old and the new addresses of the objects being recursively heapified, traversing the stack frames and registers to find stale references, and then making them point to the right objects from the map. Figure 3.2 illustrates these operations for the code snippet under consideration. (Note that stale references cannot exist on the heap, since a heap object cannot directly reference a stack-allocated object. Any attempt to establish such a reference triggers heapification of the referenced object before the reference is created).

3.1.3 Imparting Efficiency through Stack Ordering

The heapification-related steps described above are required as they affect the functional correctness of the optimistic stack allocation scheme. However, it is easy to see that the check and the heapification are costly. On the bright side, it can be expected that the actual heapification operation would have to be invoked rarely. This leaves us with the cost

of the heapification check, which is likely to be frequent (at each store to a non-primitive field, for example). Though we have not yet described the check routine completely, it is possible to see that the most expensive operation therein would be to traverse the stack frames until we find the *lhs* and the *rhs* objects, so that we can compare their addresses to in turn compare their lifetimes. We reduce the cost of these heapification checks by proposing a novel idea of ordering objects in stack frames, in a way that their addresses can be compared for the heapification checks, without doing stack walks frequently.

Essentially, while statically generating the list of stack-allocatable objects (represented by the bytecode indices of the corresponding *new* bytecodes in class files), we order the objects according to a topological sort of the points-to graph constructed for performing the escape analysis. The VM is modified to allocate objects on stack in this presented order, and the heapification checks are modified to take advantage of this order, whenever possible. We leave the details for Section 3.3, but would point out here that these stack orders bring down the overhead of heapification checks noticeably. Consequently, compared to an implementation of dynamic heapification without stack orders, we see pronounced performance improvements for various benchmarks in our evaluation.

Having motivated the reader towards the interesting challenges that our scheme solves, we now move on to a step-by-step description of the building blocks of our scheme in Sections 3.2 and 3.3.

3.2 Optimistic Stack Allocation in Eclipse OpenJ9

Eclipse OpenJ9 [30] is a popular open-source JVM implementation, which executes Java Bytecode using a multi-tier interpretation and compilation mechanism. The JIT compiler, Testarossa, uses multiple levels of compilation: noOpt, cold, warm, hot, veryHot and scorching, in order of increasing aggressiveness of the performed optimizations. A method being JIT compiled may travel across one of the compilation levels, wherein different analyses and optimizations may be performed at different levels. The hot+ levels of the JIT compiler perform an escape analysis on the code being compiled, whereby the depth of peeking (inside the called methods) varies as the compilation level goes higher. OpenJ9 also implements a pass to allocate eligible objects on stack, based on the results of this escape analysis, as well as a few other constraints. In particular, the VM requires

a fixed stack-frame size and hence precludes the stack allocation of variable-sized arrays and objects that escape a loop iteration. The VM also requires stack-allocated objects to be pre-zeroed at method entry and makes the garbage collector aware of these objects by maintaining their offsets in a metadata field with each method. Similar to other Java runtimes, OpenJ9 is also conservative in performing stack allocation (as well as other analyses and optimizations) when dynamic features such as hot-code replacement are allowed by the VM. As we show in Section 3.5, the imprecision introduced by time constraints during JIT compilation (escape analysis is only partially interprocedural, depending on the peek-depth), as well as the possibility of dynamic features, results in the VM being able to allocate a very small percentage of objects on the stack during program execution. In this section, we propose an approach that uses the results of a statically performed escape analysis to aggressively stack allocate method-local objects in OpenJ9, while being robust to the dynamic features allowed by the runtime.

The next two subsections discuss two major aspects of our scheme. Section 3.2.1 discusses how the static-analysis results are generated and then later used in the runtime to optimistically allocate non-escaping objects on the stack. Section 3.2.2 discusses the dynamic heapification routine, which essentially is the repair that is performed when a stack-allocated object escapes. We discuss stack ordering, which is an optimization to reduce the overhead of the heapification routine, in Section 3.3.

3.2.1 Static Context-Sensitive Escape Analysis

Our static analysis module is used to generate precise escape analysis results such that those results can later be used in the OpenJ9 VM during JIT compilation to perform more stack allocation. The static analysis works in two phases: (i) intraprocedural generation of conditional values; (ii) interprocedural resolution of conditional values. We first explain the kinds of dependencies encoded in conditional values and how they are generated. Next, we describe how the conditional values get resolved to obtain the final escape status of every object. Finally, we describe how we generate stack-ordered static analysis results and use them for allocating objects on stack at runtime.

1. Generating Conditional Values

In the first phase of static analysis, we analyze each method intra-procedurally and

generate conditional values for every abstract object therein. The conditional values for an object are a set of dependencies, that indicate which objects in other methods the current object's escape status depends upon; this is similar to the conditional values in the PYE framework [64], but for all the methods in a program. We use a standard worklist-based algorithm to traverse the code flow sensitively till fixed point, and store the dependencies of escape status of each object. The set of these dependencies for all the abstract objects in a method are considered as the *summary* of the method. Note that along with the generated summaries, we also maintain a *points-to graph* (denoting points-to relationships for reference variables and object fields) at each program point in the method, for performing escape analysis itself.

The conditional values generated based on PYE denote dependencies of escape status on formal parameters, passed arguments, returned values and global (static) variables. For example, for the code shown in Figure 3.1, as the object O_5 is the first parameter taken by the methods doo, the generated dependency is $\langle$Main.bar, $\langle$parameter, 1$\rangle\rangle$. Similarly for the object O_{12} in the method zar, where the object is being stored in the first parameter's field, the generated dependency is $\langle$caller, $\langle$argument, 1$\rangle\rangle$.

2. Resolving Conditional Values

The second phase of our static analysis is a resolution phase where the generated dependencies for all the objects in the first phase are resolved to get the final escape status. The resolution is again a worklist-based fix-point algorithm, which takes one method at a time and evaluates all its dependencies. A method needs to get reanalyzed only if there is an object for which some dependency value is not yet resolved or the escape status of some object on which the current method's object depends has changed.

Maintaining single level of context sensitivity. To get more precise results, we also maintain one level of context sensitivity (in order to cover simple method inlining scenarios). For example, in Figure 3.1, the parameter p1 has a dependency $\langle$caller, $\langle$argument, 1$\rangle\rangle$, which gets mapped to all its callers' first argument. Suppose from a callsite an escaping object is passed and from another callsite a non-escaping object is passed, the meet of escape status for parameter p1 will be escaping, which in turn makes all its dependencies escape. Whereas if we are context sensitive and treat every callsite separately, we achieve more precision by being able to stack allocate the objects dependent on the caller where

the argument does not escape. Note that we maintain one level of context sensitivity, which is sufficient for most cases, as discussed in Section 3.5.2. Thus, if parameter p1 is further passed to another method, it is treated as escaping.

Stack ordering. To further improve the performance at runtime, static analysis performs a final pass over the generated result, and produces results in a specific order for possible stack allocations. The static analysis creates a partial ordering among the possible candidates for stack allocation and reorders the final analysis result accordingly. These static-analysis results are finally saved in text on secondary storage (as ".res files") and conveyed to the VM for performing stack allocation. Section 3.3 provides a detailed discussion of the ordering of objects marked for stack allocation and explains how this ordering is useful in practice.

3. Stack Allocation using Static Analysis Results

The generated result of our static escape analysis is a per-method list of objects that can be allocated on the stack. As the OpenJ9 JIT converts Java Bytecode to a tree-based intermediate language (Tree IL), we represent objects in terms of bytecode indices (BCIs) of the corresponding *new* bytecodes, and provide the static-analysis results to OpenJ9 from the .res file. In OpenJ9, during the escape analysis phase, we match BCIs corresponding to object nodes in Tree IL with these BCIs to determine if the corresponding object can be allocated on the stack. That is, any BCI listed against a method m in the .res file can be allocated on the stack frame of m. We next describe when and how we read and use the static analysis results present in the .res files in the VM.

In an execution enabled with optimistic stack allocation (determined using a command-line argument), we read the supplied .res file when the runtime is being initialized and store the results (say in a map called staticAnalysisNonEscapingMap). This ensures that we incur file-reading overheads only once, in the beginning. In the existing VM implementation, whenever the JIT compiler picks a hot method for compilation, it traverses the Tree IL corresponding to that method, performs a simple (imprecise) escape analysis over the object nodes therein, and generates a list of candidates deemed suitable for stack allocation. We top up this list with the objects whose BCIs are present in the list corresponding to the method being compiled in staticAnalysisNonEscapingMap so that those additional objects also get marked for stack allocation, without actually performing

a complex analysis during JIT compilation. Later, the stack allocation routine allocates the objects present in this candidate list on the stack (instead of on the heap) when the stack frame of the corresponding method is created during execution.

3.2.2 Dynamic Heapification

As we shall see in Section 3.5, our static-analysis based scheme marks a significantly high number of objects for stack allocation during program execution. However, in a language runtime that allows interesting dynamic features such as dynamic classloading (DCL) and hot-code replacement (HCR), which essentially allow arbitrary code changes, an object that was stack allocated may escape during execution. We cannot simply leave such objects allocated on the stack as is, and need a mechanism to repair the memory such that the program behaviour is preserved. The most unique feature of our approach is this timely detection and fallback mechanism for repairing incorrect stack allocation; we describe this next. Also, as a consequence of the presence of a dynamic repair mechanism, we call our static-analysis guided stack allocation *optimistic*.

To handle the possibility of dynamically introduced escapes in the VM, we need a run-time check that timely (that is, before any incorrect access) determines whether we have allocated an escaping object on the stack. For such escaping objects, we also need a mechanism to move the object from the stack to the heap and update all previous references to the object to point to the new heap location. We refer to the introduced run-time checks as *heapification checks* and the procedure to move objects from stack to heap while correcting references as *dynamic heapification*.

We observe that there are five kinds of bytecodes that may cause an object to escape:

1. Returns of references (bytecode `areturn`).
2. Reference stores to instance fields, static fields and object arrays (bytecodes `putfield`, `putstatic` and `aastore`, respectively).
3. Throwing of exceptions (bytecode `athrow`).
4. Calls to native methods that may lead to reference stores in `sun.misc.Unsafe` (`putObject`, `putObjectOrdered`, `putObjectVolatile` and `compareAndSwapObject`).

5. JNI APIs used to perform stores in called C/C++ code that may manipulate Java objects (`setObjectField`, `setObjectArrayElement` and `setStaticObjectField`).

The last three cases typically require the associated object to be on the heap, and hence we trigger a heapification if the object is currently found on the stack. However, for the first two cases (that is, at reference returns and at reference stores), we insert more involved checks to determine the need for heapification, as discussed below. Also, in the remaining text, we describe our heapification checks in detail for the JIT compiler; the changes in the interpreter are similar except that they are introduced directly in the interpretation logic for these bytecodes instead of being emitted in the corresponding native code by the code-generation phase of the compiler.

1. Heapification Opcodes

In order to insert heapification checks, we have introduced two opcodes in the JIT compiler: possibleHeapificationAtReturn and possibleHeapificationAtStore. In the IL generation phase of the JIT compiler, when relevant bytecodes are processed, we insert these opcodes in the generated code. The evaluation of these opcodes performs the heapification check.

When an object is being returned, we add the possibleHeapificationAtReturn opcode before the corresponding instruction. When this opcode is evaluated before a return statement, we check if the return value is allocated on the returning method's stack; if so, it is heapified. This way, we do not unnecessarily heapify objects that were passed as a parameter or were allocated on the heap.

A field-store statement involves two objects: the base object on the left-hand side (say *lhs*) and the object on the right-hand side (say *rhs*) being stored into a field of the base object. Now if the lhs object has a longer lifetime (that is, it is allocated either on the heap or in a stack frame at a higher stack address), then the rhs object is escaping. In case this rhs object was allocated on the stack and is found to be escaping due to this store statement, it would require heapification. We perform this check by adding the possibleHeapificationAtStore opcode at store statements. The evaluation of this opcode involves two components: first, a cheap *address-comparison* check; and second, a more expensive *stack-walk* check that is performed only when the address-comparison check fails.

2. Address Comparisons and Stack Walks

The address-comparison check involves comparing the virtual addresses of the lhs and the rhs objects. During execution, we have access to the bounds of the stack region and can determine whether an object is present in the stack or in the heap. If an object is present in the stack, then we can further perform the address-comparison check. Note that the stack region in OpenJ9 is modelled to grow towards lower addresses and hence newly created stack frames have lower addresses than the previous stack frames. Thus, if two objects have been allocated on the stack, by comparing their addresses we can determine which object was allocated first. However, the existing stack allocation scheme allocates objects in an arbitrary order, within a single compiled-method stack frame. As a result, for the scenario when the lhs and the rhs objects were allocated in the same method, by comparing addresses we cannot always determine which object was allocated first[1]. In fact, their activation records would be cleared at the same time as they are in the same stack frame and hence, both objects have the same lifetime. When this happens, the rhs object does not escape as the lhs object also has the same lifetime. But in case the lhs object was allocated a lower address than the rhs object (due to the arbitrary nature of the existing stack-allocation scheme), we might imprecisely conclude that the rhs object escapes (a false positive). Further, this imprecise conclusion may cause more heapifications than needed.

In order to reduce false positives and make the heapification checks more precise, if we had access to the stack-frame bounds of the method in which an object was allocated, we could easily determine if two objects were allocated in the same method. But since maintaining that explicitly for all the methods during run-time would have a high overhead, we choose to use a stack-walk based approach instead, as discussed below.

A stack-walk check involves walking over the stack to determine if the rhs object was allocated in the same frame as the lhs object or in a deeper frame. A stack walk iterates over the stack frames from the most recently created frame to the deepest one, and compares the stack-frame bounds with the addresses of the rhs and the lhs objects. It stops iterating once it finds a stack frame that has either of these objects in its address range. If the lhs object was allocated in a deeper frame than the rhs object, then we infer

[1]We use this insight in Section 3.3 to *provide* stack orders and improve the address-comparison checks.

that the rhs object will escape and has to be heapified.

Observe that a stack-walk check can precisely determine if an object escapes; however, it involves a call into the JVM from JITed code as well as iteration over stack frames, and thus has a high overhead. Hence, we use the stack-walk check only when the address-comparison check fails.

3. The Heapification-Check Algorithm

Figure 3.3 summarizes the above discussion on heapification checks for store statements (of the form $a.f = b$, where both a and b are reference variables). Say the object pointed-to by a is the "lhs" and the one pointed-to by b is the "rhs". Correspondingly, say lhsReg and rhsReg contain the addresses of the lhs and the rhs objects, respectively. Also, let stackBaseReg and stackEndReg store the starting and the ending addresses of the stack region of the memory, respectively. The algorithm essentially has three consequences. We do not need any heapification either when the rhs object is outside stack bounds (i.e., already on the heap) or when the rhs object has a longer lifetime compared to the lhs object (conditions at lines 2 and 8). We definitely need heapification when we are sure of an escape, which is when the lhs object is on the heap (condition at line 5). We may or may not need a heapification when both the lhs and the rhs objects are inside the stack bounds but the address comparison fails (line 10), whereby we need to perform a stack walk to determine if the rhs object escapes.

We have implemented the stack-walking routine as a helper method in the JVM, called heapifyObjectIfRequired. Essentially, from the JIT-compiled code where the address comparison is performed, we call the heapifyObjectIfRequired method with the lhs and the rhs objects of the store as parameters. The method performs a stack walk to find out if the lhs object was allocated on older stack frame than the rhs object; and if so, then it heapifies starting from the rhs object.

The heapification routine is responsible for moving the object from the stack to the heap and updating all references to the object to point to the new heap location. This is done by allocating a new object on the heap and copying the values of the fields of the escaping object from the stack to the new object. Further, a different stack walk is performed in the JVM to iterate slot-wise through all stack frames and update references to the escaping object to point to the new heap location.

```
1  Procedure HeapificationCheckAtStore(lhsReg, rhsReg)
2      if rhsReg < stackBaseReg OR rhsReg > stackEndReg then
3          No heapification required. /* The rhs object is outside stack bounds */
4      else
           /* The rhs object is present on the stack */
5          if lhsReg < stackBaseReg OR lhsReg > stackEndReg then
               /* The lhs object is outside stack bounds, hence the rhs object escapes */
6              Heapify starting from the rhs object.
7          else
               /* Both lhs and rhs objects are on the stack */
8              if rhsReg ≥ lhsReg then
                   /* The rhs object has been allocated before the lhs object and hence
                      does not escape */
9                  No heapification required.
10             else
                   /* The lhs object has been allocated in either the same frame or a
                      deeper frame as compared to the rhs object */
11                 Perform stack-walk and heapify if needed.
```

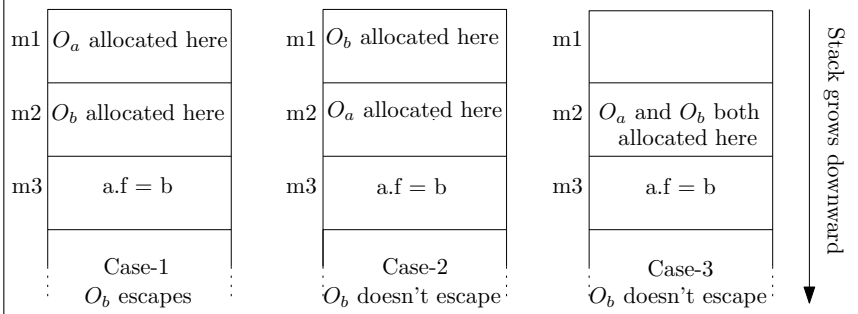
Figure 3.3: The heapification check performed at store statements ($a.f = b$). lhsReg and rhsReg store the addresses of the base object (pointed-to by a) and that of the object being stored (pointed-to by b), respectively.

It is worth noting that the escape of one object can cause the escape of other objects reachable via its fields. This brings up a need for heapifying objects recursively, which we perform by iterating over all the objects referenced by fields of the object that was heapified. If such references are found on the stack, they are also (recursively) heapified and the references to these objects are updated to point to the new heap location. During recursive heapification, we also take care to not heapify the same object multiple times using a hashmap that consists of book-keeping information about addresses of the objects that have been heapified in the current heapification invocation.

Example. After dynamic classloading in Figure 3.1, our heapification check (inserted before the bytecode corresponding to the new store at line 14) would detect an incorrect

```
1 class T {
2   T f;
3   void m1() {m2(...);}
4   void m2() {m3(...);}
5   void m3(T a, T b) {
6       a.f = b;
7   }
```

(a)



(b)

Figure 3.4: (a) Example code to explain the scenarios for stack ordering. (b) Possible stack layouts while processing store statements for the example in (a).

stack allocation for the object O_7, move it to the heap, and recursively heapify the reachable object O_9. Further, it would update the reference variable p in doo to point to the new O_7 on the heap (see Figure 3.2).

3.3 Imparting Efficiency through Stack Ordering

As discussed in Section 3.2, when an object is stack-allocated optimistically, there is a possibility that it escapes at run-time either due to unsoundess in the static analysis or due to other dynamic features. Hence, as discussed previously, we perform checks at various statements to find whether such an optimistically stack allocated object is escaping. Recall that the check at a field-store statement is performed by comparing the lifetimes of the lhs and the rhs objects involved therein. If the lhs of the store is a heap-allocated object then the rhs object is escaping by default. However, in the case where both the lhs and the rhs objects are allocated on the stack, three cases arise.

Consider the code snippet shown in Figure 3.4a. Say there are three methods m1, m2 and m3. Here, m1 calls m2, m2 calls m3, and m3 collects two parameters a and b passed by its caller. Let a be pointing to an object O_a, b be pointing to an object O_b, and the method m3 creates a field reference from O_a to O_b (via the store statement a.f = b). Note that while being in m3, the allocation sites of the objects pointed-to by its parameters, and consequently their lifetimes, are unknown.

In general, the lifetimes of two stack allocated objects involved in a store statement

could vary as shown in Figure 3.4b. The lifetime of the rhs object in a field store could be longer, shorter or equal to that of the lhs object. For the first two scenarios (Case-1 and Case-2), when the lhs and the rhs objects are allocated in different frames, an address comparison between the lhs and the rhs objects is sufficient to determine if the rhs is escaping or not. For Case-3, during the creation of the stack frame, the lhs and the rhs objects could be ordered arbitrarily and an address comparison is not sufficient to conclude whether the rhs object is escaping. Thus, we may have to perform a stack walk to find out if the lhs and the rhs objects are indeed on the same stack frame or not. However, even though the previously discussed stack-walk check provides this information precisely, it is expensive as well.

Now we present an interesting insight. The question under consideration is: *Can we do something such that a simple address-comparison works a majority of times (thus avoiding stack walks)?* We have seen that the address-comparison check works precisely for Cases 1 and 2. For Case-3, if we order the objects on the stack according to the field-store statements deterministically, we can make address comparison work precisely, whenever the points-to graph is acyclic.

In order to address Case-3, our static analysis additionally creates a partial order among stack-allocatable objects, corresponding to the order induced by points-to relationships. For the objects allocated in the same method, we obtain the stack order by performing a topological sort in the points-to graphs (which are anyway used for performing escape analysis) corresponding to those methods. Furthermore, we extend the stack orders interprocedurally while merging points-to graphs across methods. This gives us an order among the objects allocated in a method even if the points-to relationships among them are established inside callee methods. Finally, with stack-ordering enabled, for each method in the .res file, we store the BCIs (corresponding to stack-allocatable object-allocation sites) in these obtained orders.

Example. Figure 3.5a shows the intraprocedural points-to graphs for methods doo and bar from Figure 3.1. As can be seen, there is no points-to edge currently between O_5 and O_6 and hence no stack order. However, once we merge the points-to graphs of doo and bar, as shown in Figure 3.5b, we can infer the stack order between the two objects in doo: O_5 should be placed before O_6.

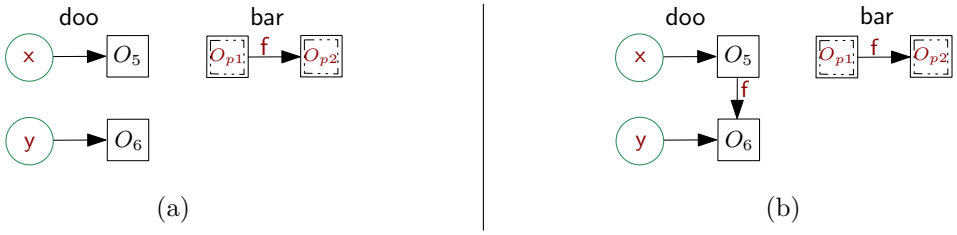


Figure 3.5: Points-to graphs for methods doo and bar from Fig. 3.1 during stack ordering: (a) intraprocedural; (b) interprocedural. Nodes O_{p_1} and O_{p_2} represent the objects pointed-to by parameters p1 and p2, respectively.

In the VM, we use the statically provided stack orders to reorder the candidates list (candidates are the objects that have been marked for stack allocation by the VM; see Section 3.2.1). Afterwards, the stack-creation phase of the JIT compiler carries out allocation according to the reordered candidates list. In a nutshell, these stack orders allow us to allocate the objects in the stack frame deterministically, so that the underlying address comparison underneath heapification checks, which is cheap, is also sufficient whenever possible.

Note that in presence of cycles in the points-to graph, the predetermined stack order may not provide the desired precision. However, if there is a cycle involving n objects, address comparison based on the order of store statements among those objects would still be conclusive for *n-1* store statements and fail for only one store statement. This failing store would require a stack walk to determine whether the rhs object needs to be heapified. Observe that this is a significant improvement over requiring a stack walk for all such store statements towards determining the need for heapification. In Section 3.5, we compare the versions of our implementations with and without stack orders, and observe that they indeed put the nail in the coffin, imparting efficiency to the complete idea of optimistic stack allocation with dynamic heapification.

3.4 Discussion

In this section, we state the correctness theorem for our approach (with an intuitive proof sketch), and bring out a few interesting design decisions and choices that we made while implementing our approach with support for the complete Java language and in an industry-standard JVM.

3.4.1 Correctness

The correctness of stack allocation, irrespective of the enabling analysis, depends on ensuring that any object that outlives its allocating method is found on the heap by any subsequent dereference. We now state the correctness theorem for our optimistic stack allocation approach.

Theorem 1. *An object that outlives the scope of its allocating method is guaranteed to be present on the heap before it could be dereferenced.*

Proof. (Sketch) Consider that an object o outlives its allocating method m, and is about to be dereferenced through a reference x in a method n. We now present a case analysis of the possibilities.

Case 1: o has always been on the heap. x points to o and dereferences it successfully.

Case 2: o was allocated on stack and the reference x was created before it escaped. Our approach inserts a heapification check (possibly followed by recursive heapification) in the interpreter and as part of the code generated by the JIT compiler, at all the statements that could make an object escape. Thus, for o, the heapification routine performs a complete stack walk and makes all the existing references point to the copied object, irrespective of whether the activation of n corresponds to interpretation or JIT compilation[2]. Hence, x successfully finds the copied object on the heap. Also, our heapification routine is atomic; that is, it ensures that the state of the object cannot be changed unless it has been heapified completely.

Case 3: o was allocated on stack and the reference x was created after it escaped. The recursive heapification routine would have already heapified o and hence any new references (including x) point to the object on the heap. □

3.4.2 Design Choices

1. Implementation over existing escape analysis. Recall that we use static-analysis results on top of the existing escape analysis in the JIT compiler (Section 3.2), though in principle we could have replaced the existing escape analysis altogether. However, we found that

[2]We ignore multiple compilation levels and multiple JIT compilers in this discussion for simplicity; our argument can be extended to them straightforwardly.

the existing analysis was able to handle certain scenarios better than a static analysis; for example, at a call site involving virtual dispatch, sometimes the JIT can speculatively inline one of the targets whereas a static analysis would conservatively take a meet among the escape statuses of the objects passed to possibly multiple methods. Furthermore, the existing analysis is quite fast (due to its imprecise nature) and does not affect compilation time noticeably.

2. Avoiding repeated heapifications. A tiered JIT-based runtime like the OpenJ9 VM may recompile a method multiple times due to the availability of better profiles or deoptimization caused by incorrect speculation. Similarly, if a dynamic property causes repeated heapifications for the objects allocated in a method, we could decide to skip the usage of static-analysis results for that method and recompile the method with the baseline escape analysis.

3. Usage of .res files. We have currently used separate text files to store static-analysis results before reading them in the VM. We can simply place these results as attributes in Java class files as well.

3.5 Evaluation

The purpose of our evaluation is twofold, first to show that a static+dynamic scheme like ours can be used to improve the amount of stack allocation in an otherwise resource-constrained environment; and second to demonstrate the impact of our novel approaches in imparting efficiency while maintaining robustness of the statically computed results.

The enhancement in stack allocation can be measured in multiple ways: (i) by counting the number of additional allocation sites that our approach allows to be marked for stack allocation – we compute this by simply incrementing a counter during JIT compilation; (ii) by counting the actual number of additional objects allocated on the run-time stack – we do this by inserting a dynamic counter through the codegen phase of the compiler, which gets incremented every time an object is allocated on the stack during program execution; and (iii) by measuring the actual amount of additional memory that gets allocated on the stack – we compute this by adding up the sizes of all the objects allocated on the stack during program execution, again by instrumenting the codegen phase of the compiler. We

also present insights from a fine-grained analysis of the features that lead to some of the highest stack allocations. These results are discussed in Section 3.5.2.

The impact of additional stack allocation itself on a program's execution can also be measured, albeit in more non-trivial ways. We compute the most direct measure – the run-time performance – in a steady state environment for all the benchmarks under consideration, and show that we sometimes improve but never degrade the performance in default settings. However, reckoning that stack allocation assumes more importance in constrained memory environments, we additionally demonstrate reduction in GC cycles (and consequently, performance improvements) by lowering the amount of heap memory supplied to the JVM. These results are discussed in Section 3.5.3.

Finally, in Section 3.5.3, in order to establish that even the offline cost of our approach is reasonable, we report the times taken by the Soot-based static analysis as well as the size overheads of the .res files that are to be supplied additionally to the VM during program execution.

3.5.1 Experimental Setup

We now describe the benchmarks, the machine, and the comparison modes used for our evaluation.

Benchmarks and machine. We have used benchmarks from the DaCapo suites 23.10-chopin and 9.12-MRI [14], and SPECjvm 2008 [59]. The benchmarks skipped from these suites are those that cannot be analyzed with the used version of Soot and TamiFlex (error reports available on GitHub). We have performed our experiments on a 12th Gen Intel(R) Core(TM) i7-12700 system with 20 cores and 16 GB RAM, running Ubuntu 22.04.1 LTS. Our static analysis has been written using Soot 3.1 [67], with reflection logs for resolving reflective calls generated using TamiFlex 2.0.3 [16]. Our modifications to the VM (both interpreter and JIT) have been made over the latest version of Eclipse OpenJ9 [30] built with JDK8 (OpenJ9 commit b4cc246, OMR commit 162e6f7, JCL commit 2a5e268); we chose this JDK version because Soot+TamiFlex do not yet fully support newer Java bytecodes (such as invokedynamic), requiring our static analysis to handle the call sites involving them conservatively. However, in order to be able to test our scheme with the newer (and larger) benchmarks from the latest DaCapo suite (23.11-chopin) that cannot

be executed with JDK8, we have also ported our implementation to the latest OpenJ9 built with JDK21 (OpenJ9 commit 7c701a7, OMR commit 87019fe, JCL commit 07e5992) and report the results therein.

In all our experiments, we run each benchmark long enough to account for warmup: for DaCapo 9.12-MRI we take the number of required warmup iterations to achieve a steady state from a prior work that used these benchmarks [50], for DaCapo 23.10-chopin we ran each benchmark for 100 warmup iterations, and for SPECjvm benchmarks we use the existing warmup built into its harness. Also, we disabled shared-class cache (a feature in OpenJ9 that allows loading of previously compiled code in subsequent runs) so that our experiments are independent of previous executions.

Comparison modes. For our experiments involving OpenJ9, we have two primary modes: one is the baseline VM (called `BaseLine`) and another is the VM with our changes enabled (called `OPT`). The mode `OPT` supports optimistic stack allocation using the partial stack order in .res files, with code generated to perform heapification checks and dynamic heapification for objects determined to be escaping at run-time. For measuring the impact of stack ordering, we also have a version `MOPT` that has everything from `OPT` minus the support for ordering objects on stack frames.

3.5.2 Enhancement in Stack Allocation

In this section, we compare the stack allocation performed by `BaseLine` and `OPT`, and present insights on the static-analysis features that lead to improvements.

1. Static and Dynamic Number of Allocations

For both the schemes `BaseLine` and `OPT`, the first and the second columns in Table 3.1 show the number of allocation sites marked for stack allocation during JIT compilation ("Static Counts") and the actual number of objects allocated on stack during program execution ("Dynamic Counts"), respectively. The parentheses in each of the columns contain the percentages of objects identified under each category, relative to the total number of static and dynamic objects, respectively. As can be seen, `BaseLine` is able to identify and stack allocate very few objects for most benchmarks, sunflow being the only exception where the dynamic counts go up to 20.08%. This is because the existing JIT analysis is

Table 3.1: Stack allocation statistics for various benchmarks with `BaseLine` and `OPT` schemes. The first six benchmarks (biojava–zxing) are from DaCapo 23.10-chopin, the next eight (avrora–sunflow) are from DaCapo 9.12-MRI, and the last eight (compiler–signverify) are from SPECjvm 2008.

	Non Optimistic Scheme (`BaseLine`)			Optimistic scheme (`OPT`)		
Bench mark	**Static Counts**	**Dynamic Counts**	**Stack Bytes**	**Static Counts**	**Dynamic Counts**	**Stack Bytes**
biojava	0 (0.00%)	0M (0.00%)	0MB	0 (0.00%)	0M (0.00%)	0MB
graphchi	0 (0.0%)	0M (0.00%)	0MB	32 (4.15%)	506.3M (6.9%)	9184.6MB
h2o	1 (0.02%)	2.6M (0.01%)	42.2MB	2 (0.02%)	3.1M (0.01%)	50.7MB
jme	0 (0.00%)	0M (0.00%)	0MB	0 (0.00%)	0M (0.00%)	0MB
kafka	0 (0.00%)	0M (0.00%)	0MB	0 (0.00%)	0M (0.00%)	0MB
zxing	0 (0.00%)	0M (0.00%)	0MB	2 (0.05%)	0.2M (0.01%)	0.47MB
avrora	9 (0.98%)	0.008M (0.01%)	0.1MB	12 (1.12%)	0.03M (0.04%)	0.9MB
eclipse	7 (0.07%)	8.9M (1.06%)	214MB	31 (0.32%)	9M (1.18%)	252MB
fop	10 (0.15%)	0.04M (0.002%)	1MB	50 (0.77%)	9.8M (0.42%)	161.2MB
h2	61 (2.33%)	29M (0.92%)	523MB	94 (3.87%)	452M (13.92%)	10801MB
luindex	35 (1.35%)	3M (2.39%)	98MB	89 (3.49%)	5M (3.49%)	133MB
lusearch	30 (1.09%)	25M (3.23%)	775MB	78 (3.05%)	59M (7.4%)	1686MB
pmd	89 (1.90%)	52M (7.20%)	1310MB	191 (3.97%)	105M (14.2%)	2465MB
sunflow	114 (10.05%)	1258M (20.08%)	30439MB	161 (13.86%)	1260M (20.11%)	30498MB
compiler	93 (1.73%)	94M (5.50%)	1720MB	137 (2.75%)	105M (6.17%)	2329MB
compress	8 (1.30%)	0.01M (3.29%)	0.2MB	13 (2.22%)	0.01M (3.8%)	0.39MB
fft	3 (0.73%)	12 (0.01%)	0.0002MB	3 (0.8%)	187 (0.16%)	0.003MB
lu	6 (0.83%)	85 (0.08%)	0.002MB	11 (3.30%)	379 (0.3%)	0.009MB
montecarlo	9 (1.50%)	2627 (2.02%)	0.09MB	9 (1.63%)	0.006M (4.47%)	0.4MB
aes	8 (1.03%)	0.01M (2.79%)	0.3MB	16 (2.25%)	0.01M (2.8%)	0.3MB
rsa	16 (1.13%)	0.1M (1.1%)	46MB	35 (3.18%)	7M (4.62%)	170MB
signverify	15 (0.84%)	0.24M (0.86%)	6.8MB	51 (3.10%)	2.1M (7.24%)	49.4MB
Overall	514	1473M	35176MB	1017	2524M	57782MB

imprecise and can work well only in scenarios where most of the allocated objects are either local to the method being compiled or passed to methods that easily get inlined.

In contrast, our scheme `OPT` is able to improve the stack allocation significantly. The static number of object allocation sites identified for stack allocation, over all the benchmarks, increases by about 98% (from 514 to 1017), and the dynamic number of object allocations on the stack increases by about 71% (from 1473M to 2524M). Note that the improvement in dynamic percentage of stack allocations could be much higher than that in the percentage of allocation sites identified statically; for example, for h2, `OPT` is able to stack allocate 13.92% of objects on stack even with the static allocation sites being 3.87%. This is because of objects created inside loops, which are identified as stack allocatable even if they are passed to other methods, due to the precise interprocedural nature of the results used by `OPT`. Few notable benchmarks where the improvement is high are the h2, pmd and signverify, wherein `OPT` increases the dynamic percentage of stack allocation from 0.92%, 7.20% and 0.86%, respectively, to 13.92%, 14.20% and 7.24% of the total number of objects.

In order to estimate the actual amount of heap memory savings with our scheme, we measured the amount of stack memory allocated by both `BaseLine` and `OPT` during execution and report them in the third columns for each scheme ("Stack Bytes"). We observe that the number of bytes allocated on stack by `OPT` increases by about 54% compared to `BaseLine` (57782MB over 35176MB, over all the benchmarks under consideration). Though for several SPECjvm benchmarks the absolute numbers are less than 1 MB, the order of increase is significant for all the benchmarks. Few examples worth pointing out are graphchi, h2 and pmd, where the bytes allocated on stack increase by orders of magnitude (9184.6 MB, 10801 MB and 2465 MB, respectively with `OPT`, over 0 MB, 523 MB and 1310 MB with `BaseLine`). These programs are indeed allocation-intensive, and `OPT` shifts the memory allocated from the heap to the stack by a good margin in all these benchmarks.

2. Simulating Longer Runs of Benchmarks with Forced JIT Compilation

Noting the near-zero numbers for both `BaseLine` and `OPT` for many of the DaCapo 23.11-chopin benchmarks, we investigated their profiles and found that during their execution with 100 iterations very few methods got JIT-compiled at levels that perform

escape analysis and stack allocation in OpenJ9. Observe that some of these benchmarks (such as kafka, which is a real-time streaming system) are indeed supposed to be run for a really long duration of time, which might lead to warmer JIT compilations subsequently; we validated this by running some of these benchmarks for thousands of iterations and noticing that stack allocations started to increase. In order to approximate such runs, and to estimate the maximum improvements our scheme could bring therein, we lowered the JIT compilation threshold of OpenJ9 to one invocation and recomputed the number of stack allocations; Table 3.2 shows the benchmarks for which either `BaseLine` or `OPT` scheme showed a significant improvement compared to the "default-mode" runs reported in Table 3.1. As we can see, under this configuration, even `BaseLine` performs a few stack allocations for all the benchmarks. Importantly, we see much better improvements with `OPT` for the benchmarks jme and kafka (where the number of objects allocated on the stack increases to 35.6% and 4.37%, respectively). Similarly, we found noticeably higher improvements for the benchmarks fop and h2 in this configuration. With these observations, we believe that our scheme could be used to make case for a stack-allocation pass even at lower levels of compilation, or devices with even lower JIT budget, as it has the potential to improve the number of stack allocations without performing any escape analysis during JIT compilation.

3. Analyzing Allocation Sites that Lead to High Number of Allocations

We conducted another experiment to understand which kinds of allocation sites get stack-allocated with our scheme but not with the existing JIT analysis. For this, we inserted a counter in the generated code to measure the number of dynamic objects created by each allocation site marked for stack allocation, sorted the results in decreasing order, and mapped them back manually to the corresponding source code. We found two important features of our static analysis that lead to an improved precision.

First of all, as `OPT`'s escape analysis is performed statically, it analyzes all the methods interprocedurally; whereas `BaseLine` performs escape analysis only for methods that are compiled at hot+ levels of compilation. Furthermore, as the peek depth of `BaseLine` is quite limited (usually just zero or one, unless all the callees are compiled at high levels of compilation), it marks objects passed as arguments beyond the peek depth as escaping. Even when `BaseLine` peeks a callee, it is limited in terms of the analysis information (e.g.,

Table 3.2: Stack allocation statistics with forced compilation of all the methods, for the benchmarks where a longer run leads to noticeable improvements.

	Non Optimistic Scheme (`BaseLine`)			Optimistic scheme (`OPT`)		
Bench mark	**Static Count**	**Dynamic Count**	**Stack Bytes**	**Static Count**	**Dynamic Count**	**Stack Bytes**
biojava	128 (1.39%)	0.0002M (0.00001%)	0.04MB	208 (2.25%)	0.007M (0.03%)	0.2MB
graphchi	70 (0.95%)	385M (5.33%)	6160MB	245 (3.33%)	656M (9.11%)	11580.3MB
h2o	173 (1.14%)	0.002M (0.24%)	0.05MB	560 (4.35%)	0.02M (1.63%)	0.4MB
jme	82 (0.88%)	0.01M (0.01%)	0.09MB	272 (2.92%)	171.1M (35.6%)	4104MB
kafka	155 (0.62%)	3.6M (0.07%)	58.6MB	653 (2.61%)	219.6M (4.37%)	3668.2MB
zxing	86 (0.88%)	0.7M (0.08%)	37.1MB	152 (1.55%)	1.35M (0.15%)	50.3MB
fop	170 (1.22%)	24.9M (1.10%)	455MB	495 (3.64%)	45.8M (2.10%)	915MB
h2	144 (1.35%)	115M (3.60%)	1878MB	388 (4.27%)	716M (22.0%)	16352MB

use-def information) that it builds for peeked methods as it does not have the necessary compile-time budget to perform these kinds of analyses in the quest for precision, whereas the `OPT` does not have these kinds of constraints. For example, in the benchmark h2, we found a class RowList whose instances ($>$ 2 million in number) are used to store rows from a database. The benchmark code uses rows as the receiver for a call rows.add, which further calls a chain of methods starting with the same receiver. `BaseLine` fails to perform interprocedural analysis of the nested calls and marks the object pointed-to by rows as escaping (not the case with `OPT`). Note that it is possible to make `BaseLine` more intricate and capture such objects, which gets evident upon noticing the improvements with forced compilation in Table 3.2 for h2 (as well as others), but this increases the time spent in JIT compilation (successfully avoided by `OPT`).

Secondly, in order to achieve more precision than peeking, `BaseLine` requires the target of a call to be inlined into the caller (which is not always possible during JIT compilation). Whereas `OPT` achieves the resultant precision without the requirement of inlining such calls, due to context sensitivity in the static analysis. For example, consider the following code snippet in which a method m is called from two different callers c1 and c2, such that c1 passes an escaping object as argument and c2 a non-escaping one. The

method m stores a new object O_x into the field f of its parameter p, and later the method c2 stores another new object O_y into the field of O_x after the call to m.

```
1  void c1() {
2    // o1 points to an escaping object
3    m(o1); }
4  void c2() {
5    // o2 points to a non-escaping object
6      m(o2);
7      o2.f.g = new Y(); // O_y
8  }
9  void m(X p) {
10     p.f = new X(); // O_x
11 }
```

Now, BaseLine would be able to identify the object O_y as non-escaping only when it is able to inline m into c2, whereas OPT analyzes the calls to m context sensitively and marks O_y for stack allocation, without requiring m to be inlined during JIT compilation. We found several instances of this scenario in the benchmark pmd and believe it would be prevalent even in others.

Overall, we assert that our static+dynamic scheme shows a marked improvement in stack allocation compared to the baseline, and we further measure its impact on performance in Section 3.5.3.

3.5.3 Impact of Additional Stack Allocation

In this section, note that the performance metric reported by the DaCapo harness is time (in msec) and that by SPECjvm is ops/sec; we have normalized the latter to enforce a lower-the-better trend.

1. Impact on Performance (Default Memory)

Figure 3.6 shows the performance box plots over ten iterations in steady state for the three modes outlined in Section 3.5.1: BaseLine, MOPT and OPT. We have skipped showing flat results for benchmarks that show a negligible difference across the various modes. For the nine benchmarks in Figure 3.6, we can sometimes see a small increase in the time taken from BaseLine to MOPT. This is due to the overhead of heapification checks performed as part of evaluating the potential escape-causing statements as discussed in Section 3.2.2. On the plus side, with our stack-ordering enabled implementation OPT, we see that the time taken either matches that of BaseLine (compiler, luindex, pmd) or even improves noticeably (fop, graphchi, h2, lusearch, rsa). Further, though we occasionally see a (very)

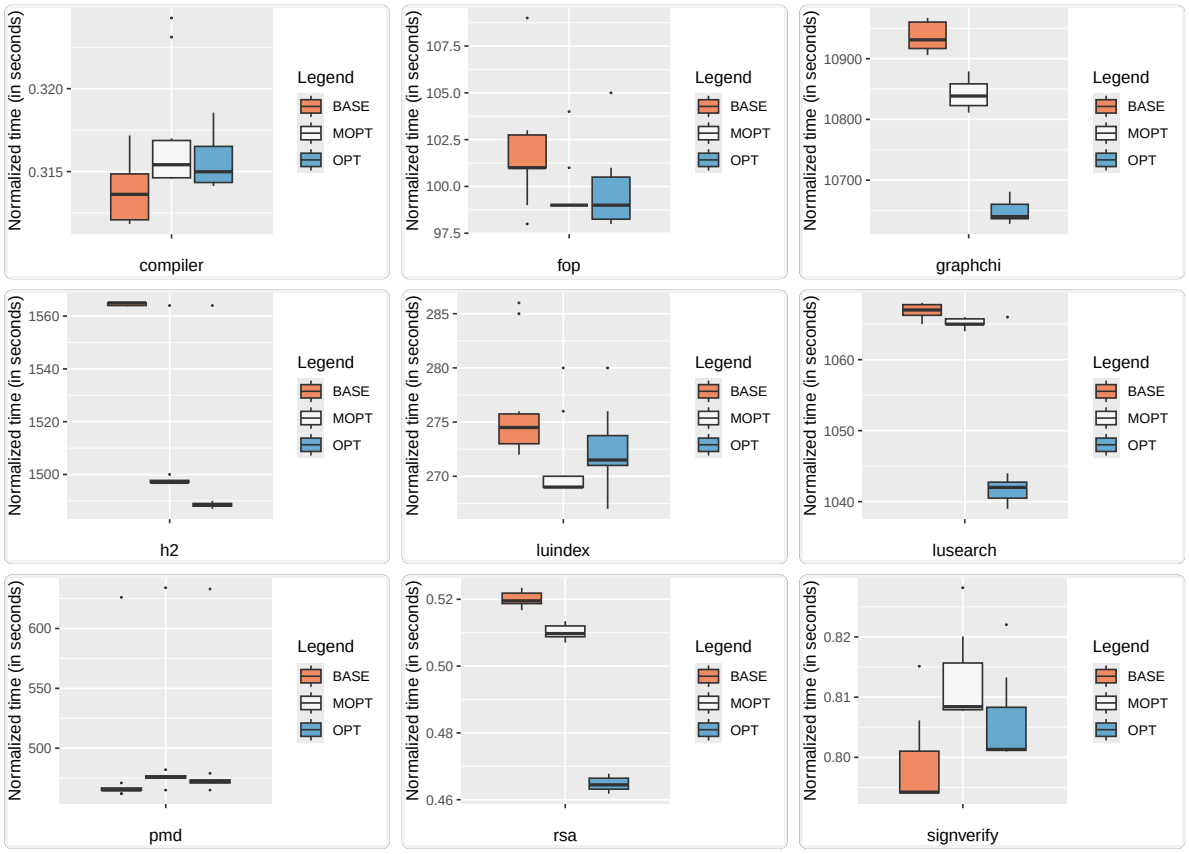
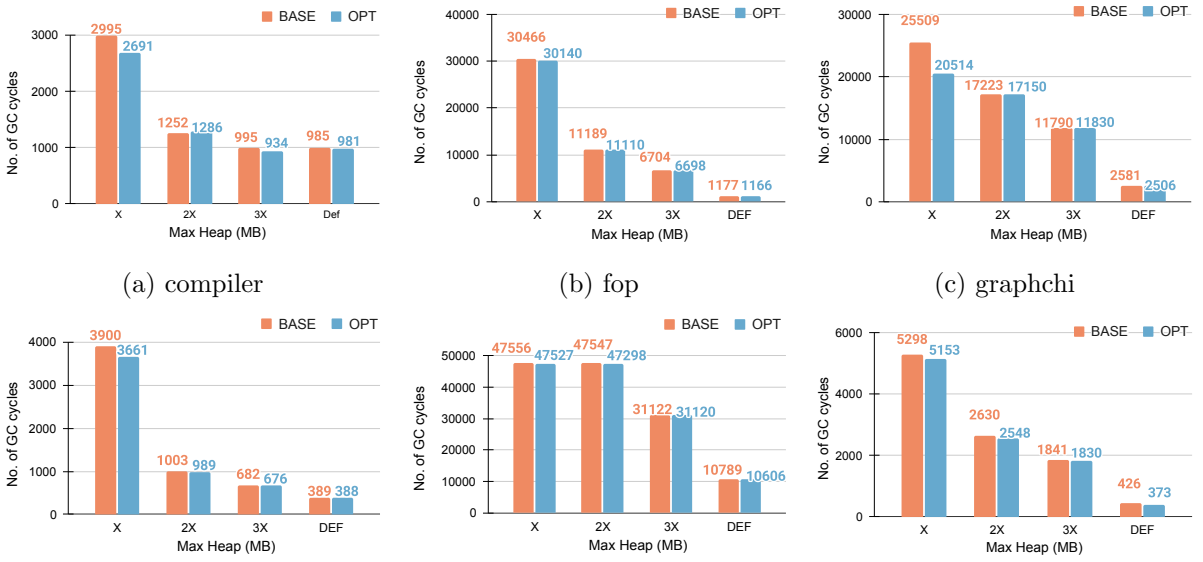


Figure 3.6: Performance comparison (default heap size); lower the better. For DaCapo benchmarks the y-axis is time (in msec) and for SPECjvm benchmarks it is the normalized reciprocal of ops/sec.

slight increase in time with `OPT`, we later show (in Section 3.5.3) that improvements with `OPT` are much more pronounced once we reduce the currently very high amounts of heap memory supplied to the JVM, for the benchmarks with good stack allocation. Thus, it can be asserted that our proposed scheme is capable of allowing much higher stack allocation without causing performance overheads, and that the performance improves too when the additional stack allocation starts showing benefits (possibly in terms of faster variable accesses and reduced GC cycles; we measure the latter next).

2. Impact on Garbage Collection

One of the relevant ways of measuring the impact of stack allocation is to compare the number of garbage collection (GC) cycles between `BaseLine` and `OPT`. To perform this experiment, we vary the amount of maximum heap memory made available to the JVM process (using the `-Xmx` argument) and compute the number of GC cycles (using the `-verbose:gc` argument of OpenJ9) for both `BaseLine` and `OPT`. More the allocation



(a) compiler (b) fop (c) graphchi

(d) h2 (e) lusearch (f) pmd

Figure 3.7: Number of GC cycles with varied heap sizes (X, 2X, 3X and default heap memory); lower the better.

on stack (and conversely, less the allocation on heap), we expect to see fewer GC cycles. Further, lesser the heap memory supplied to the VM, more pronounced should be the impact of allocating bytes on stack rather than on the heap. Now consider the results shown for six benchmarks (chosen based on the object-allocation profile and improvements observed in Section 3.5.2) in Figure 3.7. Here, we calculated the minimum heap size (say X) required to execute each benchmark and then computed the number of GC cycles by setting the available heap memory to X, 2X (moderate heap) and 3X (generous heap), as well as the default heap size (DEF), which is 4 GB for our machine.

We can observe that lower heap sizes lead to more number of GC cycles, in general (due to the increased memory pressure). Importantly, we can observe fewer GC cycles with `OPT` (particularly for compiler, graphchi and h2). This is because once we allocate more objects on stack, we directly reduce the memory pressure on heap, which subsequently translates to reduction in GC cycles.

3. Impact on Performance (Reduced Memory)

Motivated by observing the reduction in GC cycles in Section 3.5.3, specially when the memory made available to the JVM was limited to the minimum heap required to execute a given benchmark, we conducted an experiment to compare performance with

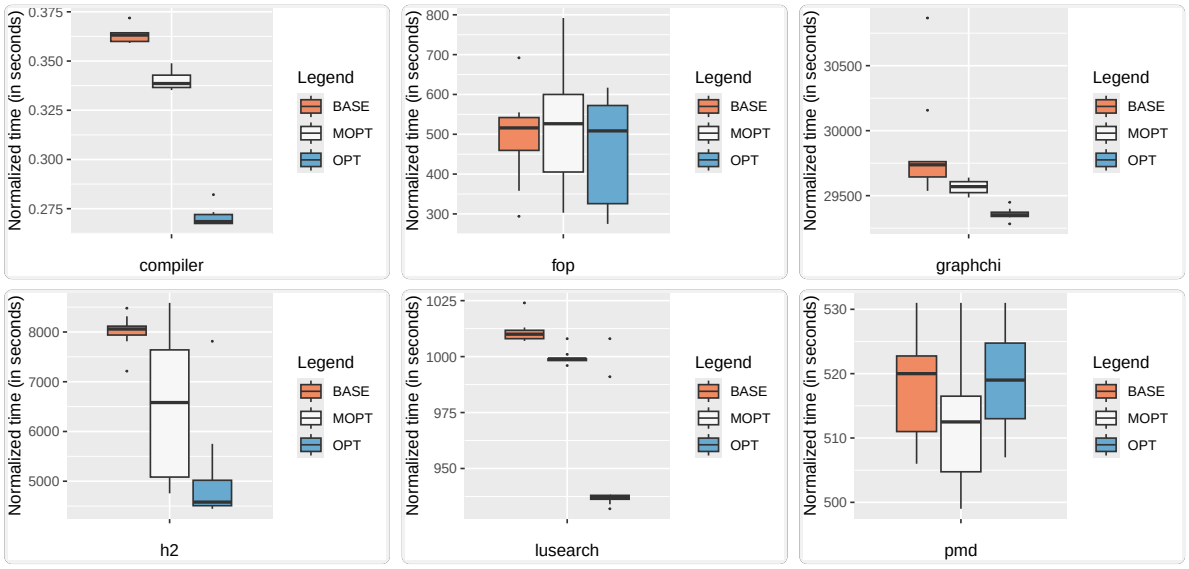


Figure 3.8: Performance comparison (minimum heap size for each benchmark); lower the better. For DaCapo benchmarks the y-axis is time (in msec) and for SPECjvm it is the normalized reciprocal of ops/sec.

our scheme under this configuration. Figure 3.8 shows the obtained box plots for the three modes under consideration for the benchmarks from Section 3.5.3.

We obtain clear performance improvements for five out of six benchmarks with higher stack allocation (compiler, fop, graphchi, h2 and lusearch), both with `MOPT` and `OPT`. With 90% confidence, the average performance improvement with `MOPT` is 4.5% and with `OPT` it is 8.8%. The highest seen improvement is for h2 (about 18.9% with `MOPT` and 37% with `OPT`); this is due to the significantly high increase in stack allocation with our scheme (about 20× more number of stack bytes; see Table 3.1), as well as the reduction in GC cycles at this configuration (from 3900 to 3661 cycles; see Figure 3.7) for this benchmark.

The `MOPT` scheme too shows improvement in this configuration as the impact of additional stack allocation overcomes the overhead caused by heapification checks. Also, for three of the benchmarks (fop, h2 and lusearch), `OPT` improves the performance further (lower differences between `MOPT` and `OPT` are expected as the heapification checks are not a major hindrance to performance any more). Interestingly, for compiler and pmd, we see a minor degradation with `OPT` compared to `MOPT` (though both are better than `BaseLine`); we attribute this to the slight overhead of creating orders in the stack compared to a strategy that has already overcome the effects of heapification checks.

Overall, with the above two measurements (reduction in GC cycles and performance

improvements on lower memory systems), we can assert that our scheme is successful in reducing the amount of GC a Java runtime has to perform during program execution, and that it can allow one to run memory-intensive applications even on lower-memory systems.

4. Cost of Heapification

In order to closely observe the behaviour of our modified VM when optimistic stack allocation needs to be undone, we tried to insert escaping BCIs in our .res files for various small-sized synthetic Java programs. In such a scenario, the VM allocates the corresponding escaping object on the stack frame of the enclosing method, which is then detected by our heapification checks at run-time. We tried to measure the cost of heapification by causing this phenomenon to happen in a loop, and the detection to happen three frames down the run-time stack by inserting an escape-causing store statement therein. Our observations for this small-program experiment were interesting: We found that a million heapifications caused our program to slowdown by one second (original run time 7 seconds); 10 million heapifications caused a slowdown of 10 seconds (original run time 30 seconds), and so on. There was no noticeable slowdown until a few tens of thousands of heapifications. We can infer from this result that the cost of heapification for larger programs, where it may not happen as frequently as our artificial program, may be quite tolerable compared to the benefits it may present. Hence we believe our scheme, coupled with efficient heapification checks, is indeed effective in scenarios where a dynamic feature may cause optimistic stack allocation to fail occasionally, and is thus practical to be employed in standard runtimes.

5. Offline Cost

Finally, observe that the advantages of our scheme – both in terms of additional stack allocation and efficient heapification checks – stem from a precise (flow-, field- and 1-level context-sensitive) static escape analysis; in fact, one could apply any of the recent advances in static analysis, for example object and type sensitivity [47, 58], and keep getting better. The advantage of performing analysis of Java Bytecodes statically is that the amount of time spent in gaining additional precision does not affect the execution time of the program. For completion, we now report the times taken, and the sizes of the .res files generated, by our static analysis implementation; see Table 3.3.

The second column shows the time taken by our static analysis (including stack or-

Table 3.3: Time taken by our static analysis along with the .res file overhead.

Bench mark	Static time (s)	.class size (MB)	.res size (MB)	Bench mark	Static time (s)	.class size (MB)	.res size (MB)
biojava	49.5	5.6	0.14	luindex	33.1	1.3	0.15
graphchi	53.6	9.0	0.20	pmd	39.9	3.7	0.20
h2o	204.0	7.3	0.56	sunflow	44.6	2.7	0.20
jme	142.3	15	0.32	compiler	114.3	2.1	0.39
kafka	367.0	15	0.78	compress	99.5	2.1	0.36
zxing	70.4	11	0.13	fft	96.6	2.1	0.34
avrora	32.1	2.6	0.14	lu	102.6	2.1	0.38
eclipse	170.7	10	0.46	montecarlo	101.6	2.1	0.36
fop	81.9	5.8	0.23	aes	98.5	2.1	0.37
h2	44.2	2.2	0.15	rsa	117.5	2.1	0.39
lusearch	40.8	1.2	0.15	signverify	97.7	2.1	0.37

dering), for each benchmark. As can be seen, the static-analysis times are tolerable (on average, 100.1 seconds) and generally increase with the increase in the size of the benchmark (see the .class size column). Further, the .res files containing the static analysis results are small – on average 0.3 MB and ≈6.0% of the size of the class files – and do not cause any significant storage and reading overhead.

To conclude this section, we note that our scheme is capable of bringing out the benefits of a precise escape analysis to a production Java Virtual Machine, which, previously, could allocate a much lower number of objects on stack even with a sophisticated escape analysis infrastructure in the JIT compiler. We believe this work is just an example of the power of a static+dynamic approach (coupled with a fallback mechanism) for optimizing programs in constrained managed runtimes, and that a strategy like ours could be applied to even more analyses and optimizations. Finally, our implementation for optimistic stack allocation and dynamic heapification is open source and integrated into the OpenJ9 VM.

To summarize, we see that our approach provides an efficient and sound approach for an important OO optimization – that of allocating method-local objects on the stack frames of their allocating methods – and showed that using a static escape analysis to

optimistically allocate identified objects on stack could improve the precision without thwarting the efficiency. Further, in order to ensure functional correctness in case the static-analysis results do not correspond to the run-time environment, it proposed the ideas of (i) dynamically checking incorrect stack allocations before they could cause an issue; (ii) repairing the memory layout by heapifying escaping objects and correcting their references; and (iii) doing so efficiently by ordering the objects on stack in a statically identified manner. Our evaluation on a production JVM, across its interpreter and compiler infrastructure, showed that the proposed scheme is capable of bringing the benefits of a precise escape analysis for improving performance in low-memory systems, by reducing memory pressure and correspondingly, the garbage-collection overheads. Further, it was a significant innovation challenge not just to design a scheme that involved multiple levels of abstractions and coding environments, but also to implement it and reach a stage where it is ready to be integrated into an industry virtual machine.

Overall, we feel that sound and efficient static+dynamic approaches like ours open up possibilities to speed up and improve the precision of various analyses and optimizations, not only for Java Virtual Machines but also for similar languages and runtimes such as C# and the .NET framework, as well as for more dynamic languages such as JavaScript, R and Python.

Chapter 4

Combining Speculation and Static Analysis

This chapter describes CoSSJIT, our contributions for enriching the static analysis with the possiblity of speculation in a JIT compiler and then using its results to perform more aggresive stack allocation during actual JIT compilation. Section 4.1 starts with a motivating example showcasing different scenarios where the existing state of the art static escape analyses generate conservative results. Section 4.2 describes the static component of our approach in terms of enriching a static escape analysis to account for possible speculations during JIT compilation. Section 4.3 describes the key run-time components of our approach for making decisions about allocating objects on stack speculatively, using dynamic class hierarchies, run-time profiles, and method-inlining information. Finally, Section 4.4 evaluates the improvements imparted by our approach in terms of increase in stack allocation, impact on performance and garbage collection, as well as the contribution of the key ideas in delivering precision.

4.1 Motivating Example

Consider the Java code snippet shown in Figure 4.1, which consists of three classes A, B and C. Here, class B and class C inherit from class A, and override its method bar. Denoting abstract object(s) allocated at line l as O_l, we can see that in the method foo the reference variable x points to the object O_5, y points to O_6, and that the field f of

```
1  class A {
2     A f;
3     static A global;
4     void foo(A p1) {
5        A x = new A(); // O5
6        A y = new A(); // O6
7        x.f = new A(); // O7
8        A z;
9        if (p1 instanceof A) {
10           z = new B();  // O10
11       } else {
12           z = new C(); // O12
13           global = y; // Escapes
14       }
15       z.bar(x);
16    } /* method foo */
17    void bar(A z) { ... }
18 } /* class A */
19 class B extends A {
20    void bar(A p2) {
21       // p2 doesn't Escape
22       p2.f = new A(); // O22
23       . . .
24    } /* method bar */
25 } /* class B */
26 class C extends A {
27    void bar(A p3) {
28       // Object pointed-to by p3 Escapes
29       global = p3;
30       . . .
31    } /* method bar */
32 } /* class C */
```

Figure 4.1: Motivating example to illustrate various considerations made by CoSSJIT.

O_5 points to the object O_7 (lines 5-7). Furthermore, depending on the conditional, the reference variable z may point to either the object O_{10} or the object O_{12}. At line 15 of the method foo, a call to bar occurs, which can invoke either the bar from class B or from class C. In the method bar defined in class B, the parameter p2 does not escape, whereas in the bar defined in class C the parameter p3 escapes because it is assigned to the static field global. So, for the object O_5 in method foo, a static interprocedural escape analyzer would take a conservative union of both the possibilities (where it could be passed to either B's bar or C's bar), and as a result mark object O_5 as escaping, which further causes O_7 — the object stored in O_5.f — to escape. Also in the method bar defined in class B, an object O_{22} is allocated and assigned to p2.f, causing O_{22} to escape because it is stored in the parameter p2's field.

Now consider a scenario in which during program execution in the JVM, the method foo is being JIT-compiled with profile information available until that point (collected usually during interpretation and/or in a lower optimizing tier of the JIT compiler). It may happen that one of the possible receiver types at line 15 has been found to be

more likely than the other; say that happens to be B. In such a case, the JIT would usually speculate and either replace the virtual call at line 15 with a direct (guarded) jump to B's bar or, sometimes even inline B's bar. With either of the decisions, note that it would have been safe to allocate the objects O_5 and O_7 on foo's stack, though a JIT that uses the static-analysis results would have lost that opportunity. In fact, it is possible that the runtime had not even loaded class C until this point (typically recorded in a "dynamic class-hierarchy table"), in which case the JIT could have even removed the associated guard.

Next, recall that the object O_{22} in B's bar escapes and hence cannot be allocated on its stack because it is stored into the field of a parameter. During JIT compilation, either because of the dynamic class hierarchy or because of the run-time profile, say the VM decides to inline this bar into foo (at line 15). In such a scenario, it would be possible to allocate O_{22} onto foo's stack instead of on the heap, as the object pointed to by x.f does not escape later in foo. But again, a standard static analysis would have missed this opportunity.

Finally, consider the object O_6 in method foo. It escapes only in the false branch of the conditional at line 9 and not in the true branch. Sophisticated JIT compilers compute frequency information either at edge or basic-block levels, and use it to perform various speculative optimizations (such as optimization of hot paths, basic-block ordering, etc), and can tell us if one of the branches is more likely than the other. Consequently, similar to the previously discussed scenarios for *speculative stack allocation*, the object O_6 can be allocated on foo's stack if the JIT determines the false branch to be more likely than the true branch. But a standard static analysis would take a control-flow merge and lose this opportunity for additional stack allocation as well.

After seeing opportunities for improving escape analysis with JIT speculation, one may wonder what would happen if a precise escape analysis was performed during JIT compilation itself. However, typical JIT compilers do not have enough budget to perform precise program analysis (e.g. interprocedural flow-sensitive) in the first place, which makes them incapable of computing intricate information considering object flows in the kinds of scenarios discussed above.

Our proposed scheme in this thesis generates static-analysis results that are "aware" of the possibility of run-time speculation during JIT compilation using a novel notion of

speculative conditions, and then resolves those conditions by hooking on to a managed runtime's speculation infrastructure before performing stack allocation. Apart from the possibility of "in-favor profiles", our scheme considers the speculation made using dynamic class-hierarchy tables (as illustrated using the example above). Furthermore, in addition to avoiding imprecision for objects from a caller passed to multiple possible callees at polymorphic call sites, our scheme considers objects in callees that could be allocated on caller's stack in case the JIT compiler speculates and decides to inline the callee within a particular caller (achieving *context sensitivity* at the callsite). Finally, our proposed scheme also addresses the imprecision that could arise from objects escaping in certain branches (whether directly within the branch like the assignment to global shown in the example above or inside some deep chain of calls made in the branch), and marks them from stack allocation based on the available branch profiles during JIT compilation.

To complete the motivating example, for the code shown in Figure 4.1, our speculative stack allocation approach would be able to (i) allocate O_5 and O_7 on foo's stack if the JIT compiler speculates that the likely receiver at line 15 is a B type; (ii) allocate O_6 on foo's stack if the JIT speculates that the likely branch for the conditional at line 9 is the *true* one; and (iii) allocate O_{22} on foo's stack if the JIT speculatively inlines B's bar at line 15.

What happens if the speculation goes wrong? The JVM that we use for implementing our scheme (Eclipse OpenJ9) provides a feature called *dynamic heapification*, which allows an object (and its reachables) to be moved from the stack to the heap and correct its references by walking the stack. OpenJ9 currently uses this routine to heapify affected objects in presence of dynamic features like hot-code replacement that change code during execution. Chapter 3 introduced this notion to guarantee functional correctness when leveraging static-analysis results in the VM, and we build our approach on top of its (publicly available) implementation. Essentially, this approach inserts run-time checks in the interpreter and in the JIT-compiled code to detect and heapify incorrectly stack-allocated objects in a timely manner, using multiple tricks to make this process efficient. Thus, in our approach, an object for which speculative stack allocation goes wrong during execution is heapified by the JVM, after which the execution continues without triggering a recompilation or full-fledged deoptimization. Such heapifications of course have penalties, and are better kept at the minimum. Hence we have parameterized our approach to use a "speculation threshold" that sets the in-favor profile percentage beyond which we decide

to speculatively allocate an object on stack; we discuss this in detail in Section 4.3.

4.2 Enriching Static Analysis

We now present our contribution to enrich static analyses with *speculative conditions*. We discuss these in context of an escape analysis whose goal is to identify objects that can be allocated on stack instead of on the heap. However, our extensions are general enough to be applied to other analyses as well, at least with respect to the speculative conditions described herein.

We have implemented our additions over an existing formulation of static escape analysis [62] for Java programs, which implements the classic flow-, field- and context-sensitive pointer and escape analysis proposed by Whaley and Rinard [70]. This base escape analysis takes the class files corresponding to a Java program, and generates a per-method list of bytecode indices corresponding to allocation sites of objects that can be allocated on its stack. As convention goes, the underlying static analysis is conservative, and lists only those objects that are guaranteed not to outlive the allocating method. (Note that an object outlives its allocating method, also is said to *escape* (see Chapter 2), if a reference to that object is stored in a global variable or into the field of a formal parameter or a thread object, or if a reference to that object is returned from the method. In addition, an object escapes if it is reachable from (i.e. accessible through) another escaping object.)

Recall that our objective is to reduce the cases where a static analysis would present a conservative answer, if it is possible to improve it later based on speculations that can be made by the JIT compiler. In context of escape analysis, these pertain to the scenarios where a static escape analysis would determine an object to be escaping across all the possible executions when it may actually be stack allocatable in a few. Based on a thorough understanding of JIT speculations, we classify such scenarios into three kinds of program statements: polymorphic call sites, possibilities of method inlining, and conditional branches. The remainder of this section describes how we handle these three scenarios during static analysis, to record the possibility of a run-time speculation with each object that can be stack allocated based on the same: Section 4.2.1 for polymorphic call sites, Section 4.2.2 for conditional branching, and Section 4.2.3 for method inlining.

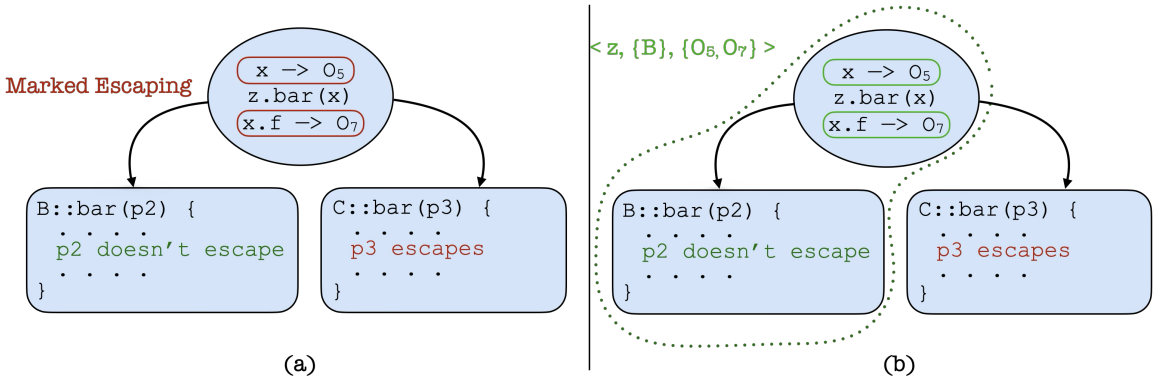


Figure 4.2: Handling a polymorphic callsite: (a) Existing static analysis; (b) CoSSJIT's static analysis. Arrows indicate (may) points-to relationships.

4.2.1 Handling Polymorphic Callsites

In Java, a polymorphic call site refers to a method invocation where the actual method executed is determined by the runtime type of the receiver object. A typical interprocedural static analysis examines all possible callees, merges their effects for the arguments being passed and the values being returned, and generates the result. In the context of escape analysis, an object passed as an argument is conservatively marked as escaping if the corresponding parameter escapes in any of the possible callees. However, it is possible that the object escapes only in a subset of callees that are less likely to be invoked at run-time. Consequently, the object would be stack-allocatable in the caller in a majority of its invocations, despite being classified as escaping by the static analysis.

Our approach begins by identifying objects that are marked as escaping solely because they are passed as arguments at a polymorphic call site and escape in at least one of the possible callees. Instead of conservatively marking such objects as escaping, we introduce a *speculative polymorphic-call condition* of the form $\langle c_{bci}, T_c, O_{bci} \rangle$, where c_{bci} represents the bytecode index of the call site (identifying the call instruction in the corresponding Java class file), T_c denotes the set of possible types of the receiver object for which the passed objects do not escape, and O_{bci} is the set of bytecode indices corresponding to objects that can be stack-allocated. This condition is interpreted as follows: if, at a call site with bytecode index c_{bci}, the receiver object's type belongs to T_c, then the objects in O_{bci} are marked for stack allocation.

Consider the code example shown in Figure 4.1. At line 15 of the method foo, the

method bar is invoked with the object O_5 passed as an argument. Since the run-time type of the receiver z can be either B or C, the method bar defined in one of these classes will be invoked during execution. In the method bar defined in class B, the parameter p_2 does not escape, whereas in the method bar defined in class C, the parameter p_3 does escape. Consequently, a state-of-the-art static analysis will conservatively merge the results from both callees, marking O_5 as well as all objects that can be accessed through O_5 (in this case, O_7) as escaping. In contrast, our approach generates a condition of the form $\langle z, \quad \{B\}, \quad \{O_5, O_7\}\rangle$, where z represents the bytecode index (bci) of the call site, and O_5, O_7 are the objects that can be stack allocated. Figure 4.2 illustrates this case for the given code snippet, contrasting our approach against the traditional one.

4.2.2 Handling Conditional Branching

Conditional statements are a fundamental construct in programs, allowing for different execution paths based on run-time conditions. In such statements, a local object may escape in one branch but may be stack allocatable in the other. Conventional static escape analysis again takes a conservative approach by computing a meet operation over the escape statuses of objects at the merge point of the conditionals. Consequently, if the object escapes in any of the branches, it is uniformly marked as escaping, even if it remains stack-allocatable in a majority of branch targets. A JIT compiler using this result would thus allocate the object on the heap, irrespective of the branch taken out of that conditional at run-time.

Given a conditional statement, our `CoSSJIT` approach extends static escape analysis with the identification of objects that escape in some branch targets but not in all. For such objects, it generates a *speculative branch-execution condition* of the form $\{\langle Cond_{bci}, O_{bci}\rangle\}$ where $Cond_{bci}$ denotes the set of bytecode indices representing branch targets that, if taken, would allow objects (represented again by bytecode indices) in the set O_{bci} to be stack-allocated.

Consider the code example in Figure 4.1. At line 13, within the else branch, the object O_6 escapes as it is assigned to the static field global of class A. Consequently, conventional static analysis marks it as escaping. In contrast, our approach introduces a branch-execution condition of the form $\langle\{9\}, \quad \{O_6\}\rangle$, where 9 represents the bytecode

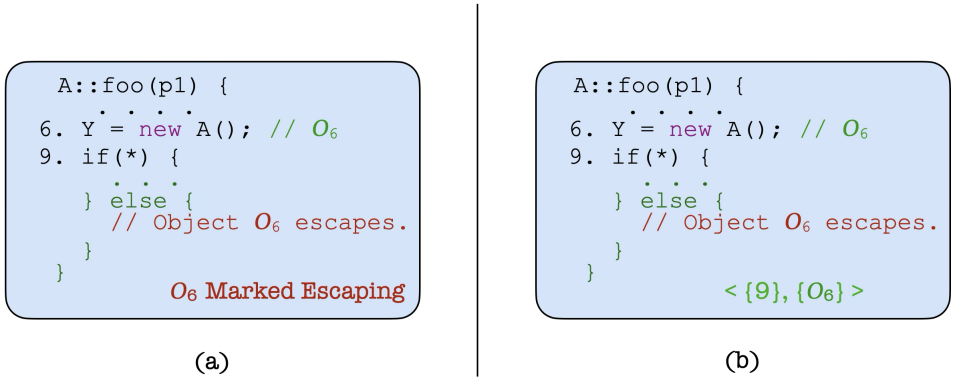


Figure 4.3: Handling objects in branches: (a) Existing static analysis; (b) CoSSJIT's static analysis.

index (bci) of the *true* branch target, and O_6 is the object that can be stack allocated if the corresponding target is taken. Figure 4.3 illustrates this scenario for the given code snippet, contrasting the existing (conventional) approach with ours.

Note that in this instance, the escape is directly observable in the caller; however, it is also possible that O_6 is passed to a method and escapes within that method. Identifying such cases requires interprocedural analysis, such that only those objects are identified and listed that escape in one of the considered branches but do not escape elsewhere due to any other reason. Similar to the manner described in Section 4.2.3, we implement the addition of speculative branch-execution conditions as an additional pass over the existing escape analyzer.

4.2.3 Handling Method Inlining

Method inlining is one of the most fundamental and widely used optimizations performed by optimizing compilers. In context of escape analysis, when a method is inlined at a call site, it may enable the allocation of local objects from the callee on the caller's stack. However, static escape analysis (performed offline) cannot account for the possibility of method inlining at various call sites. As a result, the local objects within the callee that can be allocated on the caller's stack if the callee method gets inlined are conservatively marked as escaping.

In CoSSJIT, for each call site, we analyze the callee to identify objects allocated therein that escape solely due to one of the following reasons: (i) being stored in a parameter's

```
1  class A {
2    void foo(A p1) {
3      . . . // from Fig. 2
4      z.bar(x);
5      z.foobar(x);
6    } /* method foo */
7    void bar(A z) { ... }
8    void foobar(A p3) { ... }
9  } /* class A */
10 class B extends A {
11   void bar(A p2) {
12     // p2's pointee doesn't escape
13     p2.f = new A(); // O13
14     p2.foobar(p2.f);
15   } /* method bar */
16   void foobar(A p4) { ... }
17 } /* class B */
18 class C extends A { ... }
```

Figure 4.4: Extended example from Figure 4.1 to illustrate benefits of conditional inlining by CoSSJIT.

field or (ii) being returned from the callee to the caller. In both cases, if the callee method is inlined (which is decided only at run-time) and then escape analysis is performed, it would be able to successfully mark those objects for stack allocation in the caller's frame. To account for this during static analysis, instead of immediately marking such objects as escaping, we generate a *speculative method-inline condition* of the form $\langle c_{bci}, m_{sign}, O_{bci} \rangle$, where c_{bci} represents the bytecode index of the callsite in the caller, m_{sign} denotes the signature of the invoked method, and O_{bci} is the set of bytecode indices corresponding to local objects in the callee that can be stack-allocated.

This condition is interpreted as follows: if, at a call site with bytecode index c_{bci} in the caller method, the method with signature m_{sign} is inlined at runtime, then the objects in O_{bci} are marked for stack allocation in the caller's stack frame. For a polymorphic call site, we statically analyze and generate information corresponding to each possible method that could be potentially called and inlined therein. Note that we cannot generate this condition simply while processing a call site in the caller; we need to make sure that the objects allocated in the callee, if reachable in the caller after the call, do not escape otherwise from the caller. That is, these conditions need to be generated and maintained while performing the original static escape analysis. We implement this by adding another pass to detect such scenarios over the existing escape analyzer, wherein we iterate over the statements of a method, collect locally allocated objects from callees that escape due to the aforementioned reasons (stored into parameter fields or returned), and filter the ones out that do not escape in the caller due to any other reason.

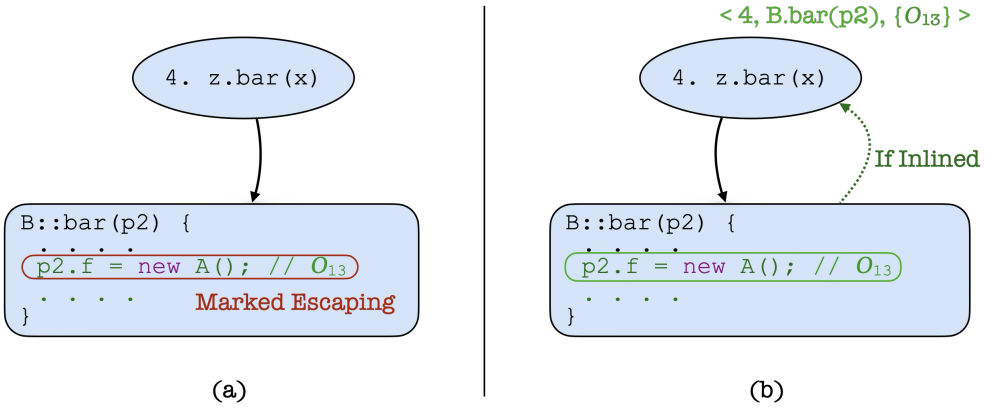


Figure 4.5: A callsite with possible inlining: (a) Existing static analysis; (b) CoSSJIT's static analysis.

Consider the code snippet shown in Figure 4.4, where at line 4, there is a call to the method bar. In the definition of bar within class B, the local object O_{13} is stored into a field of its parameter p2, due to which a traditional static analysis would mark it as escaping. Now let us assume that neither of the two foobar implementations (in classes A and B) let anything reachable from their parameters escape. In such a scenario, during JIT compilation, even if B's bar is inlined at line 4, there is no guarantee that foobar would be inlined at lines 14 and 5, both of which are necessary conditions for a traditional JIT analyzer to determine that O_{13} stays stack-allocatable in the caller foo. On the other hand, our approach statically analyzes the call-tree rooted at line 14, synthesizes a run-time condition about the stack-allocation of O_{13}, propagates it to the caller, analyzes the call-tree at line 8, and maintains the condition until JIT compilation. Specifically, it generates a method-inline condition of the form $\langle 4, \ B.bar(p2), \ \{O_{13}\}\rangle$ where 4 denotes the call site, and O_{13} represents the object that can safely be stack allocated on the caller's frame. Thus, just if B's bar is inlined at line 4, our approach would be able to mark O_{13} for stack allocation in the caller foo, without the requirement of any further inlining and without any further JIT analysis.

Figure 4.5 illustrates case under discussion for the given code snippet, contrasting the existing (conventional) approach with ours. Note that apart from conditionally escaping objects such as O_{13}, in the list of callee objects that become eligible for allocation on caller's stack post inlining, we also include bytecode indices of the objects that callee-local (i.e., unconditionally stack allocatable).

4.2.4 Discussion

As the speculative conditions described in this section are posed to be resolved during JIT compilation, one must pay special attention in ensuring that the overheads in storing and resolving them are kept minimal. We do so by listing the conditions in special text files (called .res files), in a format that can be understood by the VM as it is. For example, we represent objects using the bytecode indices (BCIs) of their allocation sites in Java class files (which are visible during JIT compilation of bytecodes), and we list down method signatures and type descriptors in the same format as represented by the JVM in its internal data structures.

Similarly, to minimize the number of generated speculative conditions and the overhead of resolving them during JIT compilation, we have made a few trade-offs. In particular, we have chosen not to generate branch-execution conditions for objects that are involved in multiple speculative conditions. Thus, if an object escapes through a polymorphic callsite as well as through a conditional statement, we simply mark it as escaping. Same goes when an object is involved in multiple sets of conditional statements. We avoid handling these scenarios, as the exponential blowup from such combinations is a well-known problem in path-sensitive analyses. Note that this choice does not reduce precision for objects that are not involved in non-branch conditions; for example, if an object may escape through multiple polymorphic callsites one after another, we generate a polymorphic-call condition for each of them. For all other scenarios, the merge operation of the extended static analysis is simply a union of the speculative conditions reaching that point, for each object.

4.3 Implementation in OpenJ9

Having described the generation of conditional static-analysis results in Section 4.3, we now turn our attention to the support we add in a managed runtime to be able to take advantage of (enriched) static-analysis results during JIT compilation, by plugging in run-time information.

We begin by describing the relevant run-time and profile information provided by the Eclipse OpenJ9 VM (Section 4.3.1), followed by explaining how we utilize them to resolve

the three kinds of speculative conditions, to consequently allocate more objects on stack (Section 4.3.2). Note that even though we describe our changes with respect to OpenJ9, most Java VMs maintain and provide run-time information of the kinds we require; as a result, we claim that our approach can be applied without any conceptual changes in other JVMs that support stack allocation too, with appropriate engineering prior to its optimization pass.

4.3.1 Working of OpenJ9

We first give a brief overview of how escape analysis operates in OpenJ9, followed by the run-time data structures that allow us to retrieve useful profile information.

1. Escape Analysis in OpenJ9: In the OpenJ9 VM, escape analysis is triggered when a method's optimization level reaches the *warm* level. The extent of escape analysis is determined by the optimization level, which can range from warm, hot, very-hot and scorching, in order of increasing aggressiveness of performed optimization. Each level dictates various constraints, such as the number of analysis passes, the depth of peeking for a method call, and so on (an object passed down the callee chain beyond the peeking depth is marked escaping). During each pass, escape analysis identifies a set of bytecode indices (BCIs) corresponding to objects that are potential candidates for stack allocation and adds them to a list called *Candidates*. These candidates are then evaluated against a series of heuristics and validation checks to determine their eligibility for stack allocation, primarily in an intraprocedural manner. Note that even though the analyzer can sometimes compute interprocedural information at higher optimization levels (through "peeking" or method inlining), these are limited by a budget that depends on several factors that traditionally limit the precision of analysis during JIT compilation. Consequently, the current approach results in missed opportunities for allocating objects on the stack.

2. Profile Info: The OpenJ9 VM also collects various types of profiling information, spanning from basic invocation and loop iteration counts in the interpreter, to detailed profiling data in different levels of the JIT compiler. This profiling data includes *branch information* for conditional statements and *instanceof checks*, *type profiles* at type-cast statements and polymorphic callsites, and so on. The Testarossa JIT compiler retains

this information as persistent data, ensuring the availability of profiling insights across different compilation levels.

3. Dynamic Class Hierarchy: OpenJ9 maintains a dynamic class-hierarchy table that tracks the loaded subclasses of a given class at any point during execution. This dynamic hierarchy table facilitates speculative optimizations based on type information, such as replacement of polymorphic callsites with guarded direct jumps, and speculative method inlining.

4. Inlining Table: Method inlining is a very useful optimization but depends, similar to most compilers, on several heuristics (including type profiles). The OpenJ9 runtime maintains an inlining table that records, for each call site in a method, the list of methods that got inlined during JIT compilation (whether guarded or unguarded).

4.3.2 Speculative Stack Allocation in OpenJ9

We now explain how we utilize run-time information to resolve the statically generated speculative conditions (of the three kinds described in Section 4.2), allowing us to mark more objects for stack allocation than both a standard JIT and an unconditional static analysis can achieve. Algorithms 1 and 2 describe how we resolve speculative polymorphic and branch-execution conditions, and speculative method-inline conditions, respectively.

1. Check if Stack Allocatable in Likely Callees at Polymorphic Callsites.

In a method m being JIT-compiled, for any object O in the *Candidates* list of objects populated by the JIT compiler, the process begins by checking if the object is marked as unconditionally non-escaping either by the local JIT analyzer or by the static analysis, and if so then simply marking it for stack allocation (lines 3-4, Algorithm 1). For other objects, we retrieve the set of speculative conditions generated by our static analyzer ($SA_m[O]$) and proceed as follows. Recall that for each polymorphic callsite c through which the object escapes, our static analyzer generates a set of receiver types for which the object does not escape. We retrieve this list ($SA_m[O][c]$) and check if the set of actual types loaded until now (found by looking up the dynamic class-hierarchy table, CH_m) is a subset of those types; if yes, we can be sure that the object does not escape without checking any run-time profiles.

Algorithm 1: Stack allocation based on speculative polymorphic-call and speculative branch-execution conditions.

Data: SA_m: *Map from objects to speculative conditions from Static Analysis for m.*
CH_m: *Run-time class hierarchy for all callsites in m.*
CPm: *Run-time profile information for all callsites in m.*
BPm: *Run-time profile information for all branch instructions in m.*
ST: *Speculation threshold.*

Result: *List of additional BCIs for stack allocation.*

$Candidates_m$= Set of all objects identified by JIT in method method m.
foreach $O \in Candidates_m$ **do**
  if O *is unconditionally non-escaping* **then**
    Mark the object O for stack allocation.
  else
    markedNonEscaping = false.
    conditionalAllocation = true.
    if $O \in SA_m$ **then**
      // 1. Check if O is likely to escape at polymorphic callsite.
      foreach *Callsite* $c \in SA_m[O]$ **do**
        if $CH_m[c] \not\subseteq SA_m[O][c]$ **then**
          if $\sum_{\forall t \in SA_m[O][c]} (CP_m(t) \leq ST)$ **then**
            conditionalAllocation = false. // *break*
      if conditionalAllocation **then**
        Mark the object O for stack allocation.
        markedNonEscaping = true.
      // 2. Check if O is likely to escape in branches.
      if ¬conditionalAllocation **then**
        conditionalAllocation = true.
        foreach *Branchresult* $br \in SA_m[O]$ **do**
          if $\sum_{\forall b \in SA_m[O][br]} (BP_m(b) \leq ST)$ **then**
            conditionalAllocation = false. // *break*
      if conditionalAllocation **then**
        Mark the object O for stack allocation.

If not (as indicated by the condition at line 10), we retrieve the frequency information of all the receiver types seen at c and analyze them as follows. We compute the sum of frequencies of all statically marked types that permit stack allocation, and if this sum exceeds a tunable speculation threshold (ST), we mark the object for speculative stack allocation. In Algorithm 1, these steps are shown at lines 9-14.

For our motivating example from Figure 4.1, consider the speculative polymorphic-call condition $\langle z, \{B\}, [O_5, O_7]\rangle$ generated by our static analyzer for the objects passed at line 15. While JIT-compiling foo, at line 15, if either the set of receiver types loaded dynamically is found to be just $\{B\}$, or if the receiver type profile suggests that B's frequency is higher than the speculation threshold, our scheme would mark both O_5 and O_7 for stack allocation. We found that a threshold of around 70% is most beneficial for stack allocation. We provide further details in the evaluation section (Section 4.4.2).

2. Check if Stack Allocatable in Some Branches.

The next check involves determining whether an object O from the $Candidates$ list can be allocated on the stack based on the resolved values of the statically computed speculative branch-execution conditions. In Algorithm 1, if the analysis until now has classified O within method m as escaping (i.e. $\neg conditionalAllocation$), the process resumes by retrieving the speculative branch-execution conditions for O (in $SA_m[O]$). These conditions, as explained in Section 4.2.2, indicate whether the object remains non-escaping in certain targets of a conditional in m. We next retrieve the run-time profile information corresponding to each conditional in the current method (i.e. BP_m), and compute the cumulative sum of execution frequencies for branches of that conditional that have been identified as safe for stack allocation by the static analysis. If this aggregated sum exceeds our speculation threshold (ST), we mark O as suitable for speculative stack allocation (lines 14-15). This ensures that the decision is guided by empirical execution trends, allowing the stack allocation of objects that might predominantly remain on stack. In Algorithm 1, these steps are shown at lines 16-22.

For our motivating example from Figure 4.1, consider the speculative branch-execution condition $\langle\{9\}, [O_6]\rangle$ generated by our static analyzer to indicate that the object O_6 does not escape if the branch at line 9 is taken. While JIT-compiling foo, at line 9, if the branch profile suggests that the branch-taken frequency is higher than the speculation

Algorithm 2: Stack allocation based on speculative method-inline conditions.

Data: SA_m: *Map from objects to speculative conditions from Static Analysis for* m.
IT_m: Inlining table for method m.

Result: List of additional BCIs for stack allocation.

// Check for inlining in method m.

1 **foreach** $c \in CallSite_m$ **do**
2 Let $Callers_c$ be the set of callers listed in SA_m.
3 **if** $\exists n$ *s.t.* $n \in IT_m[c]$ ***and*** $n \in Callers_c$ **then**
4 O_{static} = Get the list of all statically marked objects for m.
5 Mark all the objects in O_{static} for stack allocation.

threshold, our scheme would mark the object O_6 for stack allocation.

3. Check if Stack Allocatable Due to Method Inlining.

In addition to the previously discussed speculative stack allocation scenarios, which attempt to allocate caller objects on the stack based on dynamic class-hierarchy and/or run-time profile information, we further extend stack allocation to callee objects when their enclosing method is inlined at specific call sites (where the caller is known not to let the object escape further). To achieve this, we first retrieve the inlining table maintained by the OpenJ9 runtime for the current method m (represented as IT_m in Algorithm 2). This table provides information on which methods have been inlined at various call sites within m. Next, we obtain the list of speculative branch-execution conditions (from SA_m) for the method m (as described in Section 4.2.3), which lists the objects that can be stack-allocated in each caller of m, as determined by the static analyzer. If, for any call site, the statically determined condition aligns with the run-time inlining data (see line 3), all objects in the callee method that were statically identified as eligible for stack allocation are marked for stack allocation in the caller (lines 4-5). Note that this not only leads to the stack allocation of all local (unconditionally non-escaping) callee objects, but also of those that were escaping from the callee only because they were made reachable in the callee from a parameter field or because they were returned to the caller.

For our motivating example from Figure 4.1, consider the speculative method-inline condition $\langle 15, B\ bar(p1), [O_{22}] \rangle$ generated by our static analyzer for objects in bar that can be stack allocated in the caller if it is inlined at line 15. While JIT-compiling foo, if B's

bar is found to be inlined at line 15, our scheme would mark the O_{22} for stack allocation.

4.3.3 Discussion

There are a few ways our static analysis can be made even more precise (albeit with a high cost), and there are a few subtle scenarios where our static analysis does something more precise than apparent. Say a method m calls another method n at a call site c, and that the runtime inlines n at c. Here, based on our speculative method-inlining conditions while JIT-compiling m, if we obtain the list of stack-allocatable objects from n by considering its inlineability only in m, we may miss n's objects that escaped further up from m (e.g. via return x.n();). However, we have chosen to limit our analysis to consider only the immediate caller (i.e. one level of context sensitivity) because otherwise we may have an exponential number of possibilities to consider. An example of such a possibility check would be: While JIT-compiling a caller l of m, determine if m got inlined at a callsite $c1$ while inside m check if n got inlined at a callsite $c2$. Apart from cost, note that JIT compilation and inlining often happen in non-deterministic orders, which means it is not necessary that we have even JIT-compiled an inlining-enabled version of m while JIT-compiling l.

As an example of a scenario where our static-analysis based approach leads to higher precision than trivially apparent, consider that a method n got inlined into a caller m and that the JIT compiler performed a typical intraprocedural analysis of m. One may think that this may already lead to the stack allocation of all those objects from n that we would identify with our complex approach. However, in order to do so, the JIT would have to analyze the flows of those objects more deeply than an intraprocedural analysis. An example is an object allocated in n, which though now potentially stack allocatable in m's frame, is passed by m to other methods at latter call sites. Our interprocedural static analysis would have analyzed such flows beforehand, which is much more precise than what a typical JIT compiler would be able to achieve even after method inlining.

4.4 Evaluation

We now evaluate our presented approach by measuring the improvements imparted by using `CoSSJIT` over the baseline OpenJ9 VM, across its default tiered infrastructure, for a variety of benchmark programs. We aim to answer the following research questions:

- **RQ1:** How many additional objects, and how much extra memory, can be allocated on stack based on our approach?
- **RQ2:** What are the contributions of different types of speculative conditions in increasing the amount of stack allocation?
- **RQ3:** What is the impact of additional stack allocation on overall program performance?
- **RQ4:** Is the overhead of resolving speculative conditions during JIT compilation reasonable compared to the performance improvements it leads to?

4.4.1 Experimental Setup and Benchmarks

Our static analysis is implemented on top of the Soot framework version 3.1 [67], with TamiFlex version 2.0.3 [16] used to populate call graphs in presence of reflection. Our run-time components are implemented on a recent version of the OpenJ9 VM (openj9 commit b4cc246 and OMR commit 162e6f7), built alongside Java Class Libraries (JCL commit 2a5e268) version 8. We had to choose an older version of class libraries because Soot and TamiFlex are not capable of analyzing programs that use more modern Java features. However, note that this does not mean our VM version is stale; the OpenJ9 and OMR repositories evolve independent of JCL, and each of their commits can be built with any supported JCL version. In the subsequent subsections, we call the existing VM without our changes `BaseLine`, and the one with our approach `CoSSJIT`. Our experiments have been performed on a 12th Gen Intel(R) Core i7-12700 system with 20 cores and 16 GB RAM, with Ubuntu 22.04.1 LTS being the operating system. As achieving consistent performance results on JVMs is difficult, we pin the java process to a specific CPU core (out of the available 20) using `taskset`, isolate CPUs using `cpuset`, and do not have frequency scaling enabled on our machine. We also disable shared-class cache in

OpenJ9 (a feature that saves+loads pre-compiled code) in order to keep different runs of a program consistent.

We have measured the impact of our proposed techniques on various benchmark programs from the latest DaCapo 23.11 chopin suite [13] as well as from SPECjvm2008 [59]. We skipped those programs from these suites that could not be analyzed either by Soot or TamiFlex (known issues on their GitHub repositories). However, for some of the unsupported DaCapo benchmarks, we found that we could successfully analyze and execute their older versions from the DaCapo 9.12-MRI suite [14], and hence we added them as corresponding replacements. The first column of Table 4.1 shows the final set of benchmarks used in our evaluation, partitioned based on the suite they were picked from (DaCapo 23.11, DaCapo 9.12, and SPECjvm2008, respectively).

We now present a detailed evaluation to answer the research questions listed above. Section 4.4.2 answers RQ1 and RQ2, Section 4.4.3 answers RQ3, and Section 4.4.4 answers RQ4.

4.4.2 Increase in Stack Allocation

1. Number of Objects and Bytes Allocated on Stack

In order to measure the number of objects marked for stack allocation during JIT compilation, we added a counter in the routine that processes the *Candidates* list in the escape analyzer (which was described in Section 4.3). We increment this counter whenever the JIT marks an object for stack allocation (note that there are some kinds of objects – such as unknown-sized arrays – that are never marked for stack allocation). Table 4.1 reports the value of this counter for both `BaseLine` and `CoSSJIT` under the column "Static Counts", for all the benchmarks under consideration. We see that `CoSSJIT` marks a significantly higher number of objects for stack allocation (overall 3.5×). On some benchmarks this improvement is manifold, for example graphchi (109 instead of 12) and zxing (347 instead of 82). This clearly shows that there are several allocation sites that a JIT analysis would miss, but a static analyzer such as ours would be able to identify as non-escaping, even if under speculative conditions.

In order to measure the actual number of run-time objects that get stack-allocated with our scheme, we instrumented the JIT-compiled code to count object allocations on

Table 4.1: Stack allocation statistics for various benchmarks with `BaseLine` and `CoSSJIT` schemes. The first nine benchmarks (avrora–zxing) are from DaCapo 23.11-chopin, the next four (eclipse–lusearch) are from DaCapo 9.12-MRI, and the last eight (compiler–signverify) are from SPECjvm 2008.

	Base Scheme			Conditional Scheme			
Bench mark	**Static Counts**	**Dynamic Counts**	**Stack Bytes**	**Static Counts**	**Dynamic Counts**	**Stack Bytes**	**Heapify Counts**
avrora	12	104.2M (38.0%)	3335MB	14	106.5M (39.3%)	3391.4MB	0.4M
fop	19	0.67M (0.04%)	15.7MB	152	1.8M (0.10%)	48.2MB	0.2M
graphchi	12	349M (5.20%)	8327MB	109	1041.1M (14.2%)	20020.6MB	0.006M
jme	2	31M (6.50%)	759MB	15	32.1M (6.67%)	769MB	0M
kafka	42	106M (2.13%)	5730MB	67	112M (2.26%)	5933MB	0.04M
pmd	24	1762M (9.80%)	42295MB	92	1835M (10.2%)	43468MB	0.2M
sunflow	100	1077M (20.0%)	27577MB	243	2286M (34.7%)	56042MB	0.19M
xalan	81	135.8M (2.52%)	5186MB	168	162.4M (3.02%)	6356.6MB	0.7M
zxing	82	24.2M (2.61%)	987.6MB	347	153.9M (16.4%)	52796.7MB	0.4M
eclipse	7	8.9M (1.06%)	214MB	47	10.1M (1.20%)	257MB	0.008M
h2	61	33M (1.02%)	579MB	129	525M (16.2%)	12749MB	6M
luindex	30	4.9M (3.16%)	137MB	84	24.2M (15.4%)	746MB	0.06M
lusearch	53	18M (2.04%)	452MB	118	53.1M (5.80%)	1271MB	0.3M
aes	8	0.01M (2.79%)	0.3MB	21	0.02M (2.90%)	0.6MB	0M
compiler	93	94M (5.50%)	1720MB	254	129M (5.60%)	2644MB	1.6M
compress	8	0.01M (3.29%)	0.2MB	18	0.093M (15.5%)	2.75MB	0M
fft	3	12 (0.01%)	0.0002MB	11	253 (0.22%)	0.006MB	0M
lu	6	85 (0.08%)	0.002MB	11	389 (0.30%)	0.009MB	0M
montecarlo	9	0.0026M (2.02%)	0.09MB	11	0.006M (4.48%)	0.4MB	0M
rsa	16	0.1M (1.10%)	46MB	48	16.17M (4.46%)	449MB	3.1M
signverify	15	0.24M (0.86%)	6.8MB	40	3.25M (6.34%)	102.2MB	0.5M
Overall	571	3749.0M	36469.7MB	1999	5491.7M	207047MB	14.7M

stack (OpenJ9 provides a mechanism to attach counters with nodes of its intermediate representation, which are translated to assembly code during code generation); the values of these counters for both the schemes are shown under the columns "Dynamic Counters" in Table 4.1. In addition, we instrumented the heapification code in OpenJ9 to determine how many of the stack-allocated objects got heapified due to failed speculation (column labeled "Heapify Counts"). The difference between these two counters (numbers are in millions) gives a measure of how successful `CoSSJIT` was in allocating additional objects on stack. Furthermore, as the sizes of different class instances in Java may vary significantly, in order to estimate actual improvement in stack memory (and consequently, the reduction in used heap memory), we also multiplied each object with its size to obtain the number of "Stack Bytes" using both `BaseLine` and `CoSSJIT`, again shown in Table 4.1.

We notice a significant increase in the number of run-time objects (1.47×) as well as the number of bytes allocated on stack (5.7×), with our scheme `CoSSJIT` compared to `BaseLine`, across all the benchmarks. For a few benchmarks this increase is very pronounced, such as graphchi for which the memory allocated on stack (across all the iterations of the benchmark) increased from 8 GB to 20 GB, and sunflow for which the dynamic percentage of stack-allocated objects rose from 20% to 34.75%. This testifies the increased precision with our approach, and establishes it as a promising way of improving stack allocation in a managed runtime. We discuss the performance improvements imparted by this additional stack allocation in Section 4.4.3.

Value of speculation threshold. Recall from Section 4.3 that part of the logic for resolving speculative polymorphic-call conditions, as well as the resolution of speculative branch-execution conditions, depends on the value of the speculation threshold (ST). Our initial hypothesis for determining a good threshold was that non-escaping paths that are taken reasonably higher than 50% of times (to be even beneficial) would be good to perform stack allocation speculatively. Based on this, we set the speculation threshold to 70% while performing our experiments (shown in Table 4.1) and observed very few heapifications (i.e. failed speculations) for most of the benchmarks.

To estimate how many heapifications should cause us to worry, and consequently to find what value of ST may be good, we designed small programs focused solely on triggering heapifications and found that about 1,000,000 heapifications caused a slowdown of nearly 0.1 second. As each iteration of typical DaCapo benchmarks takes at least a few

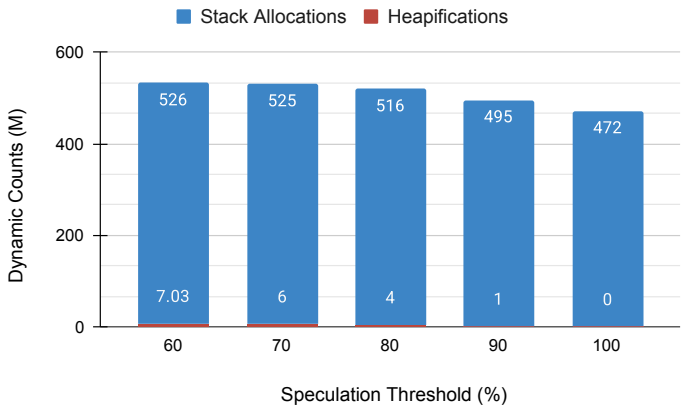


Figure 4.6: Number of stack allocations and heapifications (in millions) at different values of ST (Speculation Threshold) for the benchmark h2.

seconds, we decided to play with the value of ST for benchmarks in which we observed more than a million heapifications; Figure 4.6 shows the results of this experiment for the benchmark h2 (in whose code we found a relevant ladder of if-else conditions). We can observe that lowering the value of ST to 60% increases the amount of stack allocation (blue bars) as well as the number of heapifications (short red bars at the bottom), by similar amounts. Whereas, going in the other direction (ST values from 80-100%) reduces the number of heapifications and also the amount of stack allocation, with reduction in the latter being much higher than the former. Thus, though this threshold can be changed, we believe anything around 70% represents a reasonable trade-off in delivering enhanced stack allocations.

2. Contributions of Different Speculative Conditions

While determining objects for stack allocation with `CoSSJIT`, there are three kinds of analyses in play. Firstly, there is the existing JIT analysis (we leave it on because though it is imprecise, it uses def-use information that is already available, and is very fast). Second, in the static-analysis results, there are objects that can unconditionally be allocated on stack. Finally, there are three kinds of speculative stack allocations that are our primary contribution. In order to determine the effect of each of these categories, we measured the number of stack allocations in five different modes: the existing analyzer (`BaseLine`), only unconditional results supplied from static analysis (Unconditional), and only one kind of speculative conditions supplied over unconditional ones (PolyCall, Inlining, Branching). Figure 4.7 shows the static counts for each of these modes, totaling to the count reported

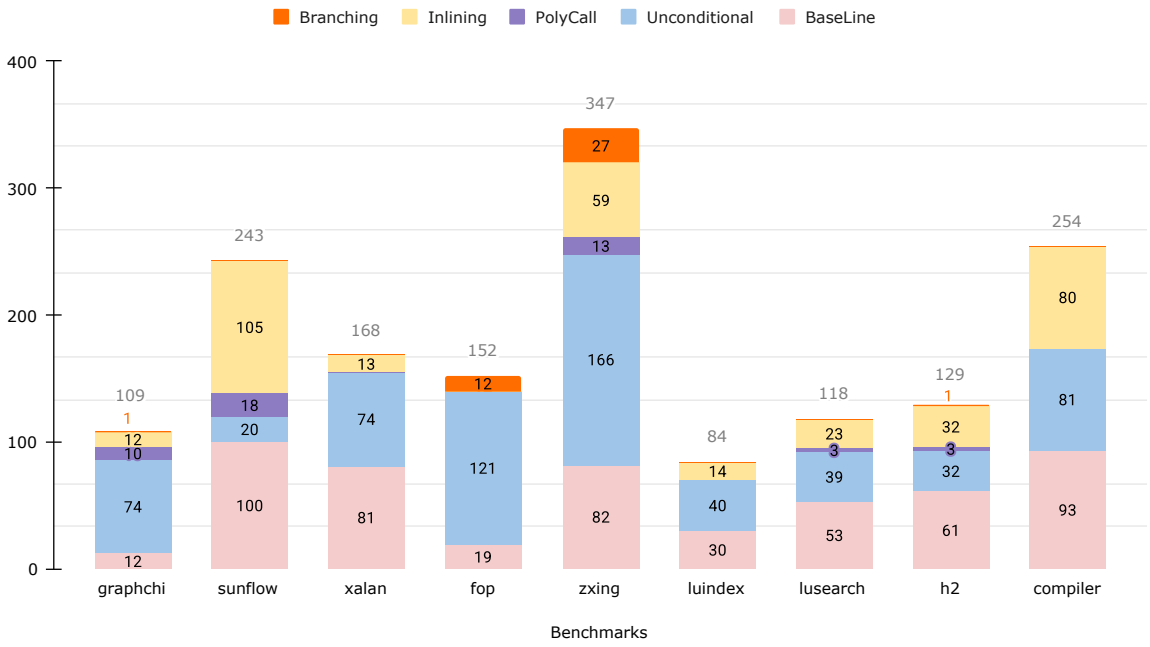


Figure 4.7: Distribution of the contributions made by different speculative conditions, in terms of the number of objects marked for stack allocation during JIT compilation.

under `CoSSJIT` in Table 4.1, for nine benchmarks that had high improvements in static and dynamic counts using `CoSSJIT`.

We observe that different speculative conditions lead to varying amounts of improvements over different benchmarks, which is expected as each condition exploits a particular program behavior. It is apparent that the highest improvement is achieved using speculative method-inlining conditions (see the yellow portions in the stacked bar charts in Figure 4.7). For example, as many as 105 out of the total 243 objects marked for stack allocation in sunflow are obtained because of these conditions; zxing and compiler are two more examples. However, there are benchmarks where branching conditions (see the orange portions) play a significant role too (e.g. fop and zxing). Similarly, for benchmarks such as graphchi, sunflow and zxing, even polymorphic-call conditions (purple portions) lead to a decent number of objects getting marked for stack allocation.

In order to further study the contributions of speculative conditions and correlate our observations with benchmark codes, we computed the correspondence between static counts and dynamic ones for a few benchmarks, and manually mapped the stack-allocated objects back to their source code. We found several interesting scenarios where each of

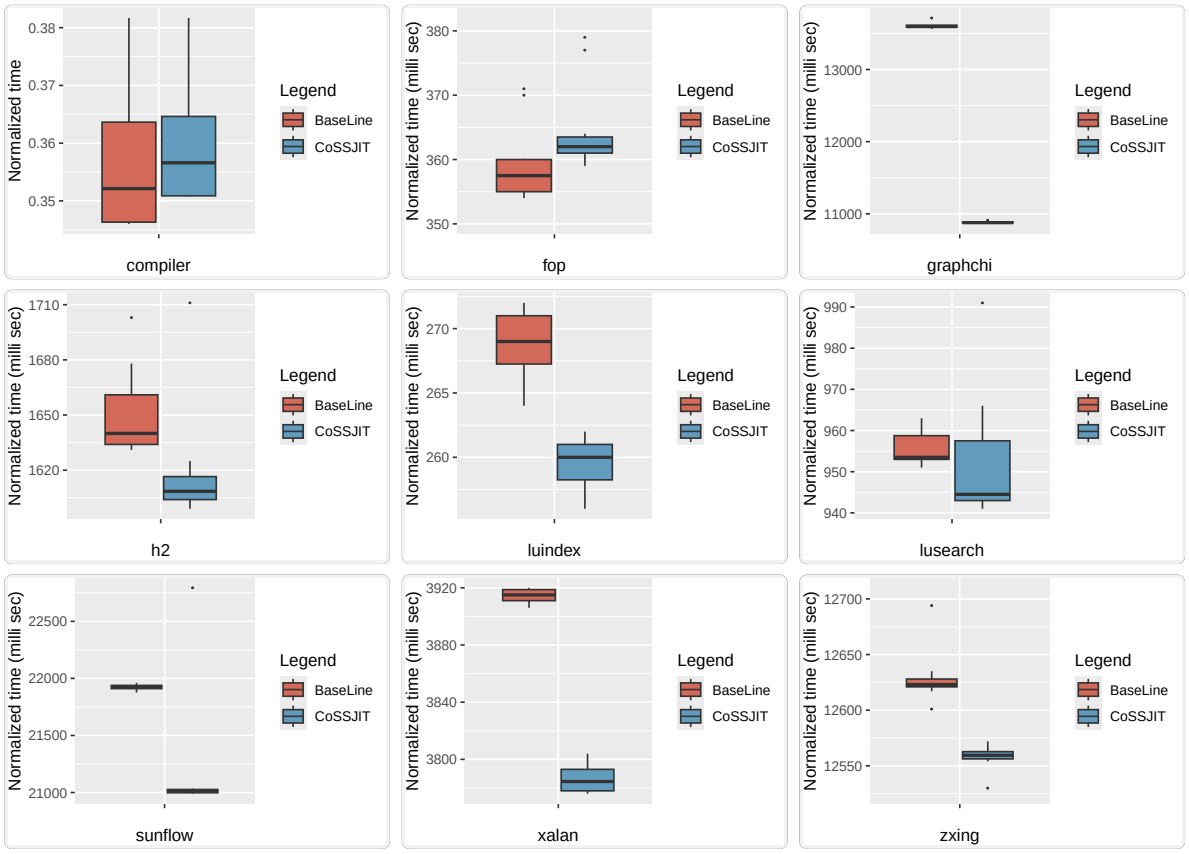


Figure 4.8: Performance comparison between `BaseLine` and `CoSSJIT`; lower the better. For DaCapo benchmarks the y-axis is time (in milli sec) and for SPECjvm benchmarks it is the normalized reciprocal of ops/sec.

the speculative conditions was useful. For example, in the class DataMatrixReader of the benchmark zxing, we found an object of type List⟨ResultPoint⟩ being returned from a method detect, that gets allocated on the stack of its caller decode when detect gets inlined during JIT compilation. Similarly, in the benchmark graphchi, we found an object of type VertexData in the GraphchiEngine class, which escapes only in one branch of an if-else ladder and gets stack allocated in all others.

4.4.3 Impact of Stack Allocation

1. Impact on Performance

Given the importance of stack allocation in reducing object allocation and access times, as well as garbage collection, we measured the impact of additional stack allocation achieved by `CoSSJIT`, over the baseline OpenJ9 VM. While measuring performances in

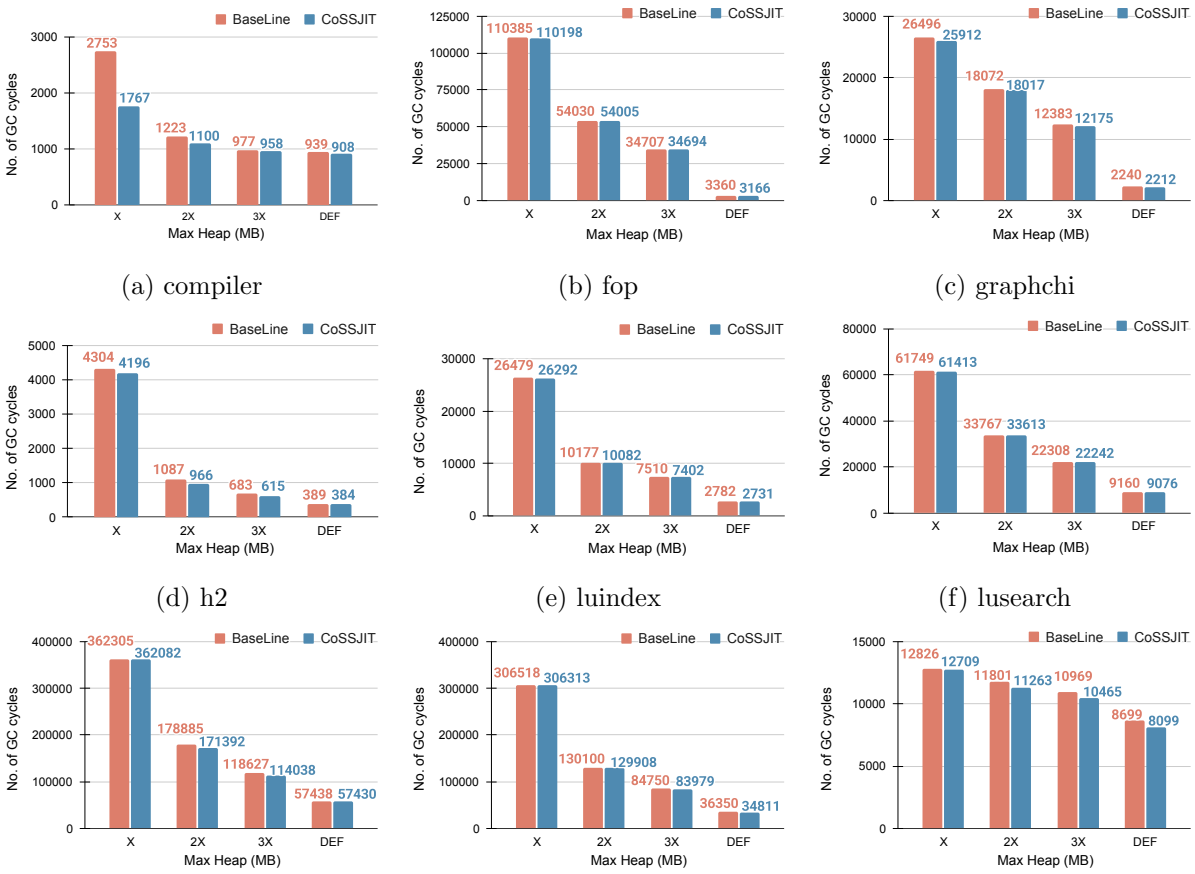
JITted systems can be non-trivial (variations due to non-determinism in order and number of compilations, method inlines, and so on), we tried our best to obtain statistically reliable numbers. For each benchmark in the DaCapo suite (where the performance metric is time), we performed enough warm-up (number of iterations taken from prior work [50] for DaCapo 9.12-MRI and fixed at 100 for DaCapo 23.11-chopin based on approximations done by [6]) and then repeated ten iterations at steady state. Similarly, for each benchmark in the SPECjvm suite (where the performance metric is throughput), we ran the standard warmup of 120 seconds and then repeated ten iterations (standard 240 seconds) at steady state. Figure 4.8 uses box plots to report the performance[1] for the nine benchmarks with high stack allocation from Section 4.4.2 (the performance metric is normalized across the two benchmark suites).

We can observe noticeable performance improvements for all the benchmarks where `CoSSJIT` improved the amount of stack allocation (the benchmark fop initially surprised us, but then we observed that the time difference across its runs is just 1-2 milliseconds and that it had multiple outlier outcomes, which makes it statistically insignificant). In particular, we note pronounced performance improvements in large benchmarks such as graphchi, h2, sunflow and zxing, and an overall improvement of 6.7% across all the benchmarks in Figure 4.8. Given that these improvements are observed amidst a plethora of other optimizations performed by the JVM, and noting how even single-digit improvements are considered significant in the Java community, we believe these results speak for themselves and show the tremendous potential held by innovative amalgams of static and dynamic analyses, for enabling aggressive optimizations in JIT compilers.

2. Impact on Garbage Collection

One of the major consequences of stack allocation is reduced heap allocation, which positively affects the number of garbage-collection (GC) cycles required to manage the run-time heap. This should be particularly useful for systems with lower available heap memories, where spending time in GC may show pronounced adverse effects on program performance. To study this in detail, we measured the impact of our approach on the

[1]For drawing box plots, we use multiple iterations at the approximated steady state within a given run, instead of picking a particular iteration from multiple runs. As we pin the programs to specific cores and run with a constant workload, we do not observe any noticeable difference in the range of performance metrics obtained using either of the approaches.



(a) compiler (b) fop (c) graphchi

(d) h2 (e) luindex (f) lusearch

(g) sunflow (h) xalan (i) zxing

Figure 4.9: Number of GC cycles with varied heap sizes (X, 2X, 3X and default heap memory); lower the better.

number of GC cycles for the programs under consideration. Here, we calculated the minimum heap size (X) required to execute each benchmark and then counted the number of GC cycles used by the VM in `BaseLine` and `CoSSJIT` settings, by restricting the maximum heap memory available to the JVM to X, 2X (moderate heap) and 3X (generous heap), as well as the default heap size (DEF), which is 4 GB for our machine. Note that lower heap sizes can also serve as a proxy for environments where even with higher amount of available memory, the GC overhead is significant (because the program was fundamentally memory intensive). See Figure 4.9 for the results of this experiment.

We observe a consistent drop in the number of GC cycles required in a VM that uses `CoSSJIT` compared to `BaseLine`. We also see that this difference increases at lower heap sizes, which implies that our approach might be even more useful on such machines; see for example compiler, where at the minimum heap size X, we observe a 35% reduction in the number of GC cycles required during its execution (1767 instead of 2753). Noticeable

drops can also be seen for benchmarks such as h2, which are known to have high allocation rates (and consequently, frequent GCs if the available memory is less). Essentially, this means that using `CoSSJIT`, one may be able to execute these benchmarks on machines with reasonably modest memory configurations than required otherwise.

We also measured performance improvements using our approach on lower heap configurations and found them, expectedly, to be more pronounced than the ones reported in Figure 4.8.

4.4.4 Additional Overheads

1. Static Overheads

We measured the time taken by the static-analysis component of our approach, for all the benchmarks under consideration; see "Static Time" in Table 4.2. As can be seen, it ranges in the order of a few minutes, and usually grows with the size of the benchmark (see the column ".class size"). As an example, for the small DaCapo benchmark lusearch the analysis time was 85 seconds and for the large benchmark kafka it was 1930 seconds. Overall, as this analysis is performed offline, we believe the time taken is reasonable. Note that the reported time also includes the time taken by Soot (our underlying static-analysis framework) to create a call graph and control-flow graphs for each method being analyzed.

We also measured the sizes of our static-analysis result files, and compare them side-by-side with the sizes of the class files of the benchmarks in Table 4.2 (see ".res size" and ".class size", respectively). We observe that the files carrying our results to the VM are typically less than one MB. The average storage overhead they represent over class files is 6.8%, which arguably is quite small. This shows the impact of some of the ideas we discussed in Section 4.2.4 in keeping the storage overheads minimal. Note that the results contained in these files can also be added as annotations in class files themselves (using the code-attribute section in Java classes), without any loss of generality.

2. Run-Time Overheads

We also determined the JIT overhead of our approach by measuring the difference between the total time spent in JIT compilation between `BaseLine` and `CoSSJIT`; see column "JIT Overhead" in Table 4.2. Note that this number is obtained across the execution

of a program, and includes warm-up as well as steady-state. Hence for perspective, we also report the total execution time of each benchmark with `BaseLine` ("Total ET") and the delta with `CoSSJIT` ("Improvement").

Table 4.2: Static and JIT overheads of `CoSSJIT`. Improvement column is dashed out for SPECjvm benchmarks because the iteration duration for them is fixed.

Bench -mark	**Static Time (s)**	**.class size (MB)**	**.res size (MB)**	**Total ET (s)**	**Improve -ment (s)**	**JIT Overhead (s)**
avrora	92	8.7	0.29	294	1.6	0.2
fop	735	27	1.40	264	1.0	0.9
graphchi	105	9.8	0.34	2524	325.3	0.2
jme	271	17	0.57	691	0.1	0.3
kafka	1930	62	1.60	582	1.0	1.1
pmd	277	27	0.67	570	41.0	0.8
sunflow	131	11	0.37	448	4.1	0.08
xalan	113	11	0.38	415	6.0	0.7
zxing	157	11	0.45	1298	20.2	0.5
eclipse	665	10	1.50	400	5.0	0.2
h2	160	2.2	0.35	173	4.0	0.06
luindex	90	1.3	0.28	165	6.0	0.7
lusearch	85	1.2	0.27	115	4.5	0.01
aes	270	2.1	0.75	2540	-	0.5
compiler	472	2.1	0.79	2540	-	4.2
compress	390	2.1	0.76	2540	-	0.5
fft	301	2.1	0.77	2540	-	0.2
lu	316	2.1	0.84	2540	-	0.02
montecarlo	310	2.1	0.74	2540	-	0.2
rsa	398	2.1	0.77	2540	-	0.1
signverify	399	2.1	0.76	2540	-	0.8
Average	365.1	10.3	0.70	1285.9	-	0.47

We can observe that compared to the amount of improvement in overall execution time

imparted by `CoSSJIT`, the JIT overhead – essentially for reading static-analysis results and resolving speculative conditions – is very low (for SPECjvm benchmarks there is a dash against Improvement because their iteration duration is fixed). Even for benchmarks such as kafka where the performance did not vary much between `BaseLine` and `CoSSJIT`, we can see that the JIT overhead did not really contribute much. In fact, there are examples where a small increase in JIT-compilation time led to magnitudes of improvement in the execution time of the program (e.g. graphchi), which as observed earlier is a consequence of the enhancement in stack allocation.

Overall, we note that our novel integration of static analysis and JIT speculation leads to a significant improvement in stack allocation, with negligible run-time overhead. The enhanced stack allocation in turn leads to decent improvements in performance for several real-world benchmarks, along with reduced amount of required garbage collection.

In a nutshell, we believe our work exemplifies a sweet spot by combining the precision-efficiency trade-offs of a static analyzer with those of a JIT compiler, harnessing the best that both have on offer. Our approach is implemented on a production Java runtime, drives an impactful and non-trivial OO optimization, and our ideas are generic enough to be applied to other optimizations, particularly those that depend on the precision of pointer analysis (e.g. synchronization elision, null-check elimination, scalar replacement, alias-analysis based loop optimizations, and so on)[2].

3. Comparison with HotSpot VM

In order to understand possibilities of improvements that can be brought by our approach in other JVMs, we also measured the impact of escape analysis in Oracle's HotSpot VM (JDK 25) and compared it with our approach in OpenJ9. Note that HotSpot's C2 JIT compiler uses an aggressive escape analysis to replace object allocations on the heap with scalar fields (an optimization called *scalar replacement*), whereas OpenJ9 uses escape analysis to perform stack allocation (targeted by `CoSSJIT`) in addition to scalar replacement. In this section, to compare the two VMs side-by-side, we count the number of heap allocations elided using either of the optimizations in both HotSpot and OpenJ9. As the

[2]If an optimization does not come paired with an inbuilt speculation-repair technique like heapification, one could still hook on to the managed runtime's deoptimization infrastructure and recompile affected methods on speculation failure.

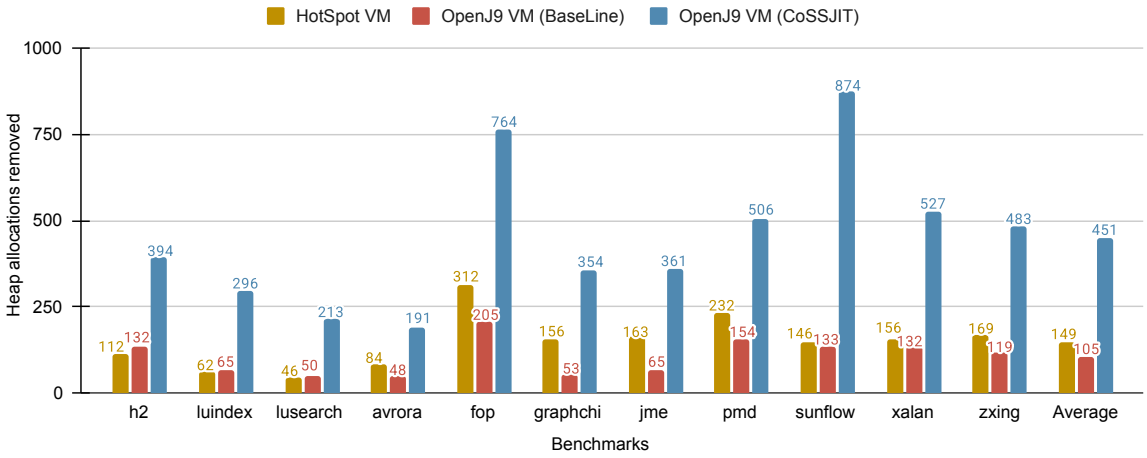


Figure 4.10: Number of heap allocations removed in HotSpot VM, OpenJ9 (BaseLine) and OpenJ9 (CoSSJIT)

compilation and escape-analysis heuristics for both the VMs are different, we compute this number by JIT-compiling all the methods and enabling escape analysis on their first invocation (`-Xcomp` and `-XX:-TieredCompilation` for HotSpot, and `-Xjit:count=1` and `initialOptLevel=hot` for OpenJ9). We carried out this evaluation on the benchmarks from Table 4.1 that are compatible with the JDK 25 version of HotSpot (eclipse and kafka threw null-pointer exceptions, while SPECjvm benchmarks require JDK 8 for execution).

Figure 4.10 shows the number of allocation sites marked to be elided from the heap for HotSpot, OpenJ9 (`BaseLine`), and `CoSSJIT`. For HotSpot, we did this using the flags `PrintEscapeAnalysis` and `PrintEliminateAllocations`; and for OpenJ9, we added compile-time counters similar to the manner described in Section 4.4.2. We can see that by default, both HotSpot and OpenJ9 mark a similar number of objects for elision from the heap (149 and 105, respectively, on average), whereas `CoSSJIT` takes this number up to 451 (on average over all the benchmarks). This result not only illustrates that OpenJ9 and HotSpot are not too far apart by default, but also that our approach can likely be used to improve the resultant optimizations in HotSpot as well.

To summarize, our approach `CoSSJIT` combines the best parts of statically performed AOT analysis and dynamically performed JIT analysis. The hypothesis was that both have their benefits – static analyses have resources to be more intricate and JIT analyses have information about the run-time behavior – and that it is possible to perform the former while leaving holes that can be filled in during the latter, essentially generating

performant compiled code in a managed runtime.

To present a concrete demonstration of our hypothesis, we identified the sources of imprecision in static escape analysis that can be enhanced with run-time information, and then made the static analysis aware of those with the notion of speculative conditions. These conditions captured the possibility of enhancing the precision of statically generated results using the speculation performed during JIT compilation, with respect to polymorphic callsites, method inlines, and branch executions. We added the resolution of these conditions in the JIT compiler of a production VM – using various run-time structures such as the dynamic class-hierarchy table, the inlining table, and receiver-type and branch profiles – and then used them to perform aggressive stack allocation. We measured the enhancement in stack allocation as well as its impact on performance, and found them to be significantly higher than the state-of-the-art.

Our implementation is deployment ready and we are working on bringing it to production in future versions of the OpenJ9 runtime. Furthermore, with the recent advent of more systems that facilitate AOT analysis for JIT compilers (for Java as well as other dynamic languages), we believe that interesting applications and avatars of our approach could lead to novel optimizations being aggressively performed, for other Java runtimes as well as for more programming languages.

Chapter 5

Tuning the VM

Having seen approaches for combining the best from both static and dynamic worlds, this chapter presents our contributions towards tuning the VM to maximize the benefits of the static+dynamic scheme described in Chapters 3 and 4. It first introduces enhancements to the inlining heuristics used by the JIT compiler to determine whether a method should be inlined at a callsite or not. Section 5.1 describes on how using static analysis results as one of the inlining heuristics leads to more informed inlining decisions at a callsite. Next, Section 5.2 discusses mechanisms for adapting static analysis results to the runtime changes, ensuring they remain relevant during execution. Section 5.3 presents an approach of using precomputed static analysis results at lower optimization levels, thereby extending the benefits of escape analysis results to methods getting compiled at lower optimization levels. Finally, Section 5.4 presents the improvements and impact in terms of stack allocation after tuning the VM for using static analysis.

5.1 Inlining in OpenJ9

Inlining at a callsite is one of the most frequently applied and important optimizations performed by the Testarossa JIT compiler in the OpenJ9 virtual machine. By replacing a method invocation with the body of the target method, inlining eliminates method call overhead and exposes additional opportunities for subsequent compiler optimizations. The OpenJ9 VM provides two primary inlining strategies for determining whether a method should be inlined. The first, and more commonly used, is the *DumbInliner*.

This is a lightweight inliner with a very small compilation budget, designed primarily to inline a limited number of extremely small methods. Due to its simplicity and low overhead, it is especially useful at the lower optimization levels, where compilation time must be kept low.

The second inlining strategy is the *MultipleCallTargetInliner*, which is considerably more sophisticated. Unlike DumbInliner, this inliner uses a richer set of heuristics that includes larger inlining budgets, callee method size, and runtime profiling information collected during program execution. The choice of inliner depends on the optimization level at which the method is compiled. The DumbInliner is typically used at the cold optimization level, where it attempts to perform limited inlining within a constrained budget. Whereas the MultipleCallTargetInliner is employed at higher optimization levels, from warm to scorching, where more aggressive and profile-guided optimizations are beneficial.

In both inlining strategies, the decision to inline is mainly dependent on the predefined heuristics, with the size of the callee method being one of the primary factors. Additionally, in the case of the MultipleCallTargetInliner, when a callsite may have multiple possible callees, the inliner also utilizes receiver-type based profiling information to identify the most likely target method for inlining. This allows the inliner to enable more informed and profitable optimization decisions.

Although the existing heuristics are effective, they can be further enriched by incorporating information derived from static analysis results without incurring any additional cost. Integrating static-analysis-based results into the inlining framework can enable the compiler to make more accurate and context-aware optimization decisions. Such additional heuristics may help identify methods that are particularly beneficial for optimizations such as stack allocation, thereby improving both the quality of generated code and overall runtime performance.

5.1.1 Enriching the Inlining Heuristics

We now describe how static analysis must adapt to the runtime changes and update its results accordingly. Generally, in most of the JIT compilers, the decision to inline a method is driven primarily by the size of the callee, with profiling information taken into consideration when competing candidates have comparable sizes. Since inlining is one

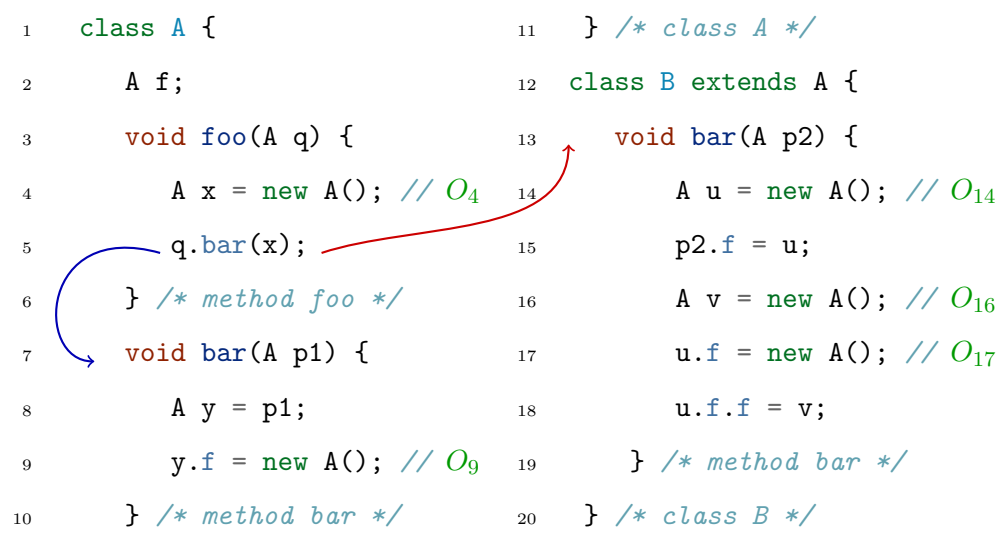


Figure 5.1: Example to explain various aspects of our run-time tuning.

of the earliest optimizations performed during compilation, most of the other analysis results are typically not available at that stage. Even when such results are present, they are mostly conservative as they are performed by the JIT.

Consider the Java code snippet shown in Figure 5.1, where class A defines two methods, foo and bar, while class B extends from class A and overrides method bar. At line 5 in method foo, the call to bar is dynamically resolved to the corresponding bar method depending on the receiver type at run-time. Now at run-time, if the JIT compiler decides to inline this call, it will have to select from these two possible callees. Given that the code size of method bar in class A is significantly smaller, the existing inlining heuristic would typically favour this version. However, from the perspective of stack allocation, the choice of callee has important implications. Inlining the bar defined in class A would lead to one object (Object O_9) from the callee to be allocated on the caller's stack frame. Whereas if the bar defined in class B gets inlined, it would lead to three objects (Objects O_{14}, O_{16} and O_{17}) to be allocated on the caller's stack frame. This tells that a purely size-based heuristic may overlook opportunities for improved stack allocation and overall runtime performance.

To address this limitation, the inliner must be augmented with an additional heuristic that incorporates results from static escape analysis. Now, to make sure that the inliner is aware of the escape analysis results, the inliner needs an extra heuristic to check if the static analysis results are present, and if so, how many possible callee objects can

be allocated on the caller's stack frame. Specifically, when such analysis information is available, the compiler should consider the number of objects that can be safely allocated on the caller's stack for each potential callee. We next describe a cost–benefit model for inlining that integrates static escape analysis results into the decision-making process.

5.1.2 Cost-Benefit Function for Inlining Heuristics

The enhanced inlining heuristic considers three factors: code size, profiling information, and the potential for stack allocation in the callee. To ensure that inlining does not violate existing code size constraints, the new heuristic is applied only when the callee satisfies the predefined size threshold for inlining. Once this condition is met, the heuristic combines profiling information with the number of bytecode indices (BCIs) that can be marked for stack allocation. The weights assigned to these two factors are flexible and can be adjusted. However, after experimenting with different configurations, we observed that giving higher importance to profiling information yields better results. Accordingly, profiling information is assigned a weight of 60%, while potential for stack allocation contributes the remaining 40%. Thus, the overall benefit associated with each receiver type T at a call site is computed as follows:

$$Benefit(T) = 0.6 \cdot Profile(T) + 0.4 \cdot StackNorm(T)$$

where Profile(T) represents the profiling value for type T and $StackNorm(T)$ is a normalized stack allocation component with respect to the maximum number of stack-allocatable BCIs among all candidate callees. The value of normalized stack allocation $StackNorm(T)$ can be computed as:

$$StackNorm(T) = \frac{StackBCI(T)}{\max(StackBCI)}$$

where $StackBCI(T)$ denotes the number of bytecode indices (BCIs) that can potentially be marked for stack allocation when the resolved callee type is T. The denominator, $max(StackBCI)$, represents the maximum number of stack allocation opportunities observed across all possible callee types at the call site. Thus, the generalized formulation for each receiver type T becomes:

$$Benefit(T) = 0.6 \cdot Profile(T) \ + \ 0.4 \cdot \frac{StackBCI(T)}{\max(StackBCI)}$$

To illustrate, for example from Figure 5.1, assume that the profiling information shows that for the call to method bar (at line 5) for both receiver types, A and B, is identical at 50%. Furthermore, the static analysis results provide the possible stack allocation numbers, that is inlining A enables 1 stack allocation opportunity, whereas inlining B enables 3 stack allocation opportunities. In that case, the benefit values can be computed as follows. The cost function will evaluate for each possible type:

$$\text{For type A:} \quad 0.6 \cdot 0.5 + 0.4 \cdot 0.33 = 0.42$$

$$\text{For type B:} \quad 0.6 \cdot 0.5 + 0.4 \cdot 1 = 0.7$$

So, based on this evaluation, the inliner selects the bar defined in class B for inlining at the callsite. Although both receiver types have identical profiling weights, and the default inliner would have selected type A, type B is preferred here due to its higher stack allocation potential, which yields a greater overall benefit under the proposed heuristic.

5.2 Adapting Static Results to Runtime Decisions

The static analysis results are generated using only the information available at static compilation time. Since the analysis must account for all possible runtime execution scenarios, it inherently produces conservative results to ensure correctness under every possible execution path. The `CoSSJIT` approach presented in Chapter 4 introduced the concept of incorporating speculative assumptions that may hold at run-time and generating analysis results based on those possible assumptions. However, there are scenarios where speculative assumptions or run-time changes may influence or invalidate the results produced by the static analysis. In such scenarios, the generated analysis information may no longer remain applicable, which leads to limiting the use of static analysis results. Therefore, to ensure that static analysis results continue to remain useful in the presence of run-time changes, it becomes necessary to adapt and refine the static analysis results.

Towards tuning static-analysis results corresponding to run-time changes, the thesis proposes a second enhancement as an approach that allows the static analysis to adapt to run-time changes and provide updated static analysis results. The key idea is to make sure that the static analysis results generated for a method remain useful even when run-time optimizations or transformations, such as method inlining, change the program

structure. In particular, static analysis results should be reused and propagated when such transformations change the effective calling context. The proposed approach therefore focuses on reusing, propagating, and refining static analysis information in response to such transformations, allowing the compiler to preserve the benefits of static analysis while maintaining correctness under evolving runtime behavior.

A primary motivation for this enhancement arises from the conditional inlining results generated for stack allocation. Static analysis may compute conditional results for a method based on the behaviour of its callees. For example, consider the code in Figure 5.1, where for method foo the static analysis will generate a conditional inlining result for the callsite at line 5 for two of the possible methods; If the method bar from class A gets inlined, then the object O_9 should get stack allocated on foo's stack frame, if the method bar from class B gets inlined, then the objects O_{14}, O_{16} and O_{17} should get stack allocated on foo's stack frame. These are the static analysis results we have for method foo. However, note that these results are associated with the method foo under the assumption that foo is the immediate caller of bar. At run-time, however, further inlining may occur. Suppose the method foo itself gets inlined in its caller (say main) while one of the bars was also inlined in foo. In this scenario, the original static analysis result generated for foo must be adapted such that it remains valid inside the main method as well. Without that, the static analysis result is of no use in main method. So even though objects O_{14}, O_{16} and O_{17} (when bar from class B is inlined) or object O_9 (when bar from class A is inlined), could potentially be allocated on main's stack frame, but the JIT compiler has no mechanism to infer that.

To address this, we extend the JIT compiler is extended to maintain per-object metadata indicating the method in which each object is allocated. This information enables the compiler to reinterpret previously computed static analysis results in the new context. During the analysis of method main, the compiler checks: (i) whether a given object originates from one of the callee methods of foo, and (ii) whether the corresponding method has already been inlined into main. If both conditions hold, the conditional inlining results computed for foo can be safely propagated and applied within main. This mechanism ensures that static analysis results, although initially computed with for a single level of inlining, remain applicable even as the runtime system performs deeper inlining. Consequently, the analysis becomes *adaptive* to run-time changes, enabling more effective exploitation of stack allocation opportunities across multiple levels of inlining.

5.3 Stack Allocation at Lower Optimization Levels

The thesis introduces a third enhancement that revisits the optimization level at which escape analysis is performed within the JIT compilation pipeline. Escape analysis is known to be one of the computationally expensive analyses and is therefore performed only when a method reaches a sufficiently high hotness threshold. As described in Section 2.2, OpenJ9 provides multiple optimization levels for methods to get JIT compiled ranging from cold, warm, hot, veryhot to scorching. In our implementation on OpenJ9 in this thesis, a full fledged escape analysis is performed at the hot optimization level or higher while a significantly more limited and less precise variant is applied at the warm level. As a result, only methods compiled at these higher levels benefit from optimizations enabled by escape analysis. This limitation is particularly important in highly CPU-constrained environments, such as containers allocated only a fraction of a CPU core or heavily overloaded servers, where the JIT compiler may aggressively favor cold-level compilations to reduce interpretation overhead. Although frequently executed methods are expected to be recompiled at higher optimization levels eventually, such recompilations may be substantially delayed under these conditions. Consequently, improving optimization quality at the cold level can provide tangible benefits. Notably, these methods are compiled at the cold optimization level not because they are inherently infrequently executed, but rather due to system-wide CPU resource constraints.

Observe that, the static+dynamic scheme proposed in this thesis can enable the use of precise, statically computed escape analysis information even at lower optimization levels, without incurring the run-time overhead of performing escape analysis during compilation. To exploit this opportunity, the thesis introduces an additional lightweight optimization pass that operates at lower tiers (e.g., cold or warm levels). This pass leverages precomputed static escape analysis results to guide stack allocation decisions during early-stage compilation. By integrating this additional optimization pass into the compilation pipeline, the JIT compiler can further safely allocate much more objects on the stack, even for methods that do not reach higher optimization levels. In effect, this additional pass improves both the reach and efficiency of escape analysis, delivering better performance gains while maintaining low compilation cost.

It is important to note that this optimization pass performs stack allocation only for

those objects that are guaranteed by the static analysis to be safely stack allocatable. Objects whose escape behavior depends on run-time values or profiling information (conditional results) are intentionally left to the traditional escape analysis pass executed at higher optimization levels. This design choice avoids potentially harmful aggressiveness when enough profile information is not available, while still enabling additional optimization opportunities at lower compilation levels.

5.4 Evaluation

This section provides our preliminary results over different benchmarks after tuning the VM. The purpose of our evaluation is to show that a VM that is tuned to adapt for a static+dynamic scheme can be used to improve the amount of stack allocation much more compared to an untuned version. Similar to previous chapters, we measure the enhancement in stack allocation, by counting the number of additional allocation sites that our approach allows to be marked for stack allocation and by counting the actual number of additional objects allocated on the run-time stack. Similar to the chapters, we also measure the actual amount of additional memory that gets allocated on the stack. We have used benchmarks from the DaCapo suites 23.10-chopin, 9.12-MRI [14], and SPECjvm 2008 [59]. The benchmarks skipped from these suites are those that cannot be analyzed with the used version of Soot and TamiFlex (error reports available on GitHub). We have performed our experiments on a 12th Gen Intel(R) Core(TM) i7-12700 system with 20 cores and 16 GB RAM, running Ubuntu 22.04.1 LTS. Our modifications to the VM have been made over the latest version of Eclipse OpenJ9 [30] built with JDK8 (OpenJ9 commit 149f5d5eb1, OMR commit 434d26431, JCL commit cb4b9ae56e). For the experiments, we have considered two primary modes: first the baseline as the `CoSSJIT` approach proposed in the previous chapter and second is the VM with our changes enabled (called `TUNED`).

5.4.1 Static and Dynamic Number of Allocations

For both schemes, `CoSSJIT` and `TUNED`, the first and second columns in Table 5.1 report the number of allocation sites identified for stack allocation during JIT compilation ("Static Counts") and the actual number of objects allocated on the stack during program execu-

Table 5.1: Stack allocation statistics and compile time statistics for selected benchmarks for using `CoSSJIT` and `TUNED` schemes. The first nine benchmarks (avrora–zxing) are from DaCapo 23.11-chopin, the next benchmark luindex is from DaCapo 9.12-MRI, and the last six (aes–signverify) are from SPECjvm 2008.

	BaseLine (`CoSSJIT`)				Static+Dynamic Scheme (`TUNED`)			
Bench -mark	**Static Counts**	**Dynamic Counts**	**Stack Bytes**	**Comp. Time**	**Static Counts**	**Dynamic Counts**	**Stack Bytes**	**Comp. Time**
avrora	14	106M	4.56GB	2.8ms	22	233M	6.50GB	3.1ms
fop	152	1.8M	0.04GB	2.8ms	184	2.40M	0.06GB	3.0ms
graphchi	109	891M	15.0GB	7.0ms	198	895M	15.40GB	7.7ms
jme	15	32.1M	0.76GB	1.1ms	17	34.0M	0.82GB	1.2ms
kafka	42	112M	5.07GB	3.0ms	78	120M	5.49GB	2.9ms
pmd	92	509M	10.2GB	2.8ms	149	635M	13.2GB	3.0ms
luindex	84	24.2M	0.36GB	7.2ms	87	25.4M	0.79GB	7.6ms
aes	21	0.02M	0.0006GB	4.2ms	25	0.034M	0.0007GB	4.2ms
compiler	254	129M	2.6GB	5.0ms	480	138M	3.10GB	5.6ms
compress	18	0.093M	0.00027GB	2.8ms	23	0.095M	0.00028GB	3.2ms
monte_carlo	11	0.006M	0.0004GB	2.9ms	16	0.007M	0.0005GB	2.9ms
rsa	48	16.17M	0.4 GB	3.5ms	89	30.5M	0.68GB	3.7ms
signverify	40	3.25M	0.1GB	4.4ms	80	6.60M	0.15GB	4.5ms
Overall	900	1824.6M	39.0GB	49.5ms	1448	2120.3M	46.19GB	52.6ms

tion ("Dynamic Counts"), respectively. As observed, the base scheme `CoSSJIT` identifies and stack-allocates a good number of objects across most benchmarks, with compiler being the best benchmark, where the static count reaches 254. However, the `TUNED` scheme achieves further improvement in stack allocation numbers with the proposed changes. Notable improvements are observed in benchmarks such as avrora, compiler, rsa and signverify where `TUNED` increases the static stack allocation counts from 14 to 22 (leading to 126M dynamic count), 2543 to 480 (leading to 9M dynamic allocation count), 48 to 89 (leading to 14M dynamic allocation count) and 40 to 80 (leading to 3M dynamic allocation count) respectively, out of the total number of objects.

In order to estimate the actual reduction in heap memory usage achieved by the `TUNED` scheme, we measure the amount of memory allocated on the stack by both `CoSSJIT` and `TUNED` during execution, reported in the third column for each scheme ("Stack Bytes"). The results indicate that `TUNED` further improve the amount of memory allocated on the stack compared to `CoSSJIT`. Notable examples include avrora, pmd, and compiler, where the stack allocation grows more under `TUNED` (4.56 GB, 10.2 GB, and 2.6 GB, respectively), compared to 6.50 GB, 13.2 GB, and 3.10 GB under untuned `CoSSJIT`.

Further, to estimate the compile-time overhead, we measured the average time spent on JIT compilation for each benchmark ("Comp. Time"). We found that, across all benchmarks, the increase in JIT compilation time is negligible (note that the overall execution time of these benchmarks is in the order of tens of seconds). Thus, our approach introduces negligible compile-time overhead.

Overall, these results suggest that the static+dynamic approach improves the precision of the analysis, leading to more effective identification of stack-allocatable objects. Furthermore, the benefits of this approach can be further amplified when the virtual machine is appropriately tuned to support such a scheme.

5.4.2 Impact of Enriched Inlining Heuristics

After enriching the inlining heuristics with information derived from static analysis (Section 5.1.1), we evaluated their effectiveness by measuring the number of callsites at which a more suitable callee method, different from the baseline inliner, was selected for inlining. To perform this evaluation, the VM was instrumented at the point where inlining decisions are made, allowing us to record instances in which a method recommended by the static analysis was ultimately chosen by the inliner.

The results showed that, for several benchmarks, the enhanced heuristics enabled the selection of different inlining targets, which in turn contributed to an increase in stack allocation opportunities. Figure 5.2 presents the number of callsites at which a statically suggested callee method was selected and successfully inlined. In particular, benchmarks such as *jme* and *fop* exhibited noticeable improvements, with 5 and 3 such callsites respectively.

Although the observed numbers may appear modest, their impact becomes signifi-

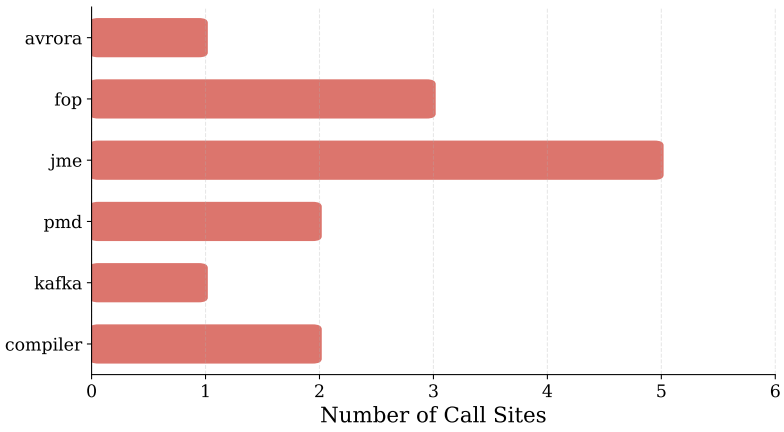


Figure 5.2: Callsites where better callee methods (in terms of number of stack allocatable objects) were identified and selected for inlining.

cantly more pronounced for long-running applications. As execution progresses, the VM gathers additional profiling information and compiles a larger number of methods, thereby exposing further inlining opportunities. Improved inlining decisions at these later stages can subsequently enable more aggressive escape analysis and increased stack allocation of objects. Overall, the experiment demonstrates that enriching the inlining heuristics with static-analysis-guided information can improve the quality of inlining decisions. This not only allows the compiler to select more profitable callee methods for inlining, but also increases the effectiveness of subsequent optimizations such as stack allocation.

5.4.3 Impact of Adapting Static Analysis Results at Run-time

By enabling the static analysis results to adapt to runtime changes (Section 5.2), we observe a significant improvement in the number of objects allocated on the stack. To evaluate the effectiveness of this approach, we instrumented compiled code with additional debug counters (provided in the OpenJ9 VM) to record both the number of allocation sites identified statically and the actual number of objects allocated on the stack during runtime execution.

Figure 5.3 presents these results, where the x-axis denotes the number of additional allocation sites identified for stack allocation and the y-axis represents the corresponding number of run-time objects allocated on the stack. The results clearly demonstrate that adapting static analysis information to run-time changes substantially increases the

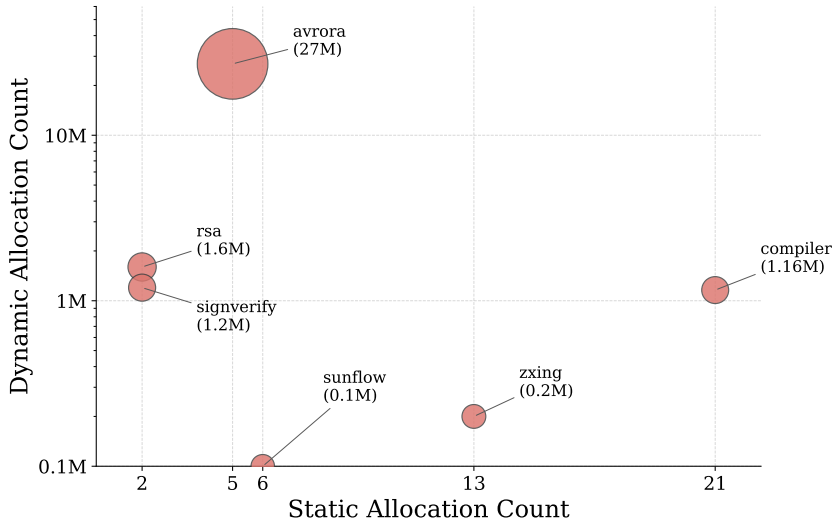


Figure 5.3: Additional objects allocated on the stack after adapting static-analysis results to run-time changes.

number of objects eligible for stack allocation. For instance, in the *avrora* benchmark, an additional 5 allocation sites were identified and marked for stack allocation, resulting in approximately 27 million additional objects being allocated on the stack during execution. Similarly, for the *compiler* benchmark, 21 additional allocation sites were marked for stack allocation, leading to approximately 1.16 million additional stack-allocated objects. These results highlight the benefits of adapting statically generated analysis results to evolving runtime behavior. In particular, incorporating runtime-aware adjustments allows the compiler to preserve and extend optimization opportunities that would otherwise be lost due to runtime transformations such as method inlining. Overall, the proposed approach improves the effectiveness of stack allocation by enabling a larger number of objects to be safely allocated on stack frames rather than on the heap.

5.4.4 Impact of Escape Analysis at Lower Optimization Levels

Performing escape analysis at lower optimization levels (Section 5.3) also contributes to improvements in stack allocation. However, methods compiled at the cold optimization level are typically invoked less frequently during execution. As a result, although additional allocation sites may be identified and marked for stack allocation, their impact on the overall dynamic allocation count is comparatively smaller.

Analyzing the effect of stack allocation for cold-level JIT compiled methods is non-trivial due to the non-deterministic nature of compilation, where methods may or may not get JIT compiled across different runs. However, its impact can be observed in the Table 5.1, where several benchmarks exhibit a noticeable increase in the number of static count, while the corresponding increase in dynamic stack allocations remains relatively modest (the dynamic count). For example, benchmarks such as graphchi and compiler show significantly higher improvements in static allocation counts compared to their dynamic allocation numbers. This disparity arises because many of the optimized methods are executed infrequently, limiting their contribution to the total number of runtime object allocations.

Nevertheless, the results demonstrate that introducing additional analysis passes to refine and reuse static analysis information can effectively increase stack allocation opportunities. Importantly, these improvements are achieved without incurring additional runtime overhead for performing the optimization itself, since the majority of the analysis work is carried out statically during compilation. Overall, the approach improves the effectiveness of escape analysis while maintaining low runtime optimization costs.

5.4.5 Impact on Performance

Having established that our tuning substantially increase stack-allocation opportunities, we next evaluate their impact on performance. Since stack allocation eliminates heap allocation overhead and can improve object access locality, it has the potential to reduce execution time and improve overall application throughput. However, accurately measuring performance in a JIT-compiled environment is inherently challenging due to sources of run-time variability, including compilation timing, garbage collection activity, and operating-system interference. Consequently, we ensured that the reported results are as reliable and reproducible as possible by controlling execution conditions and repeating experiments across multiple runs.

To minimize measurement noise, all experiments were conducted in an isolated execution environment with the benchmark process pinned to a fixed set of 5 processor cores. Furthermore, each benchmark was executed for a sufficient number of warmup iterations to ensure that the application had reached a stable execution state before measurements

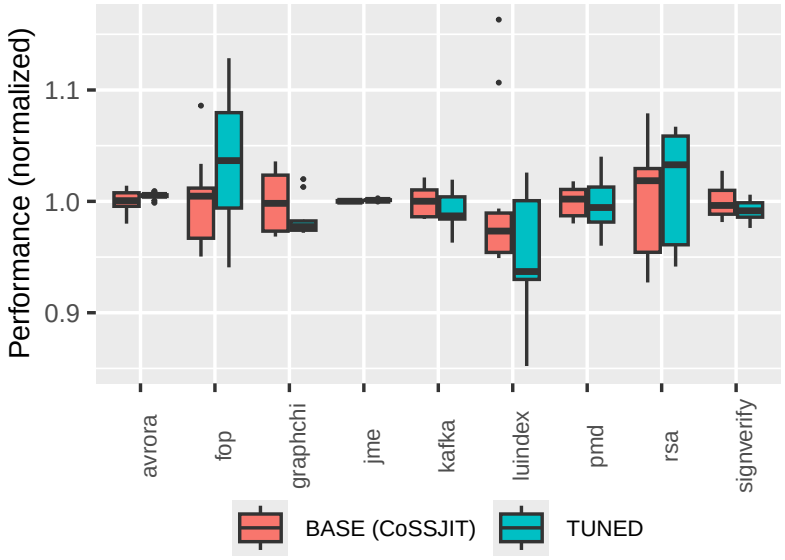


Figure 5.4: Performance at default heap size.

were collected. For DaCapo 9.12-MR1 bechmark suite, we used the warmup iteration counts reported in prior work [50]. For DaCapo 23.11, each benchmark was executed for a fixed 100 iterations, while for SPECjvm benchmarks, where performance is measured in terms of throughput, we performed a standard warmup phase of 120 seconds, followed by ten steady-state iterations of 240 seconds each. Each experiment was repeated ten times as an independent process execution to account for residual run-to-run variation.

The collected measurements were normalized against the baseline configuration and are presented as box plots in Figure 5.4. These box plots illustrate the distribution of performance across the repeated executions, providing insight into both the average performance and the variability of the results. The figure highlights the benchmarks that exhibit the largest increases in stack-allocation opportunities, a trend that is also reflected in the stack-allocation counts reported in Table 5.1. For example, *fop* shows an increase from 152 to 184 stack allocations (21.1%), while *rsa* increases from 48 to 89 (85.4%).

Overall, the results indicate that the proposed optimizations do not introduce any measurable performance regressions and generally maintain performance comparable to the baseline. While the performance of many benchmarks remains similar to the baseline, several benchmarks exhibit noticeable improvements. In particular, *fop*, *graphchi*, and *rsa* demonstrate consistent performance gains, reflecting the benefits of the additional stack-allocation opportunities enabled by our optimizations. These benchmarks also exhibit

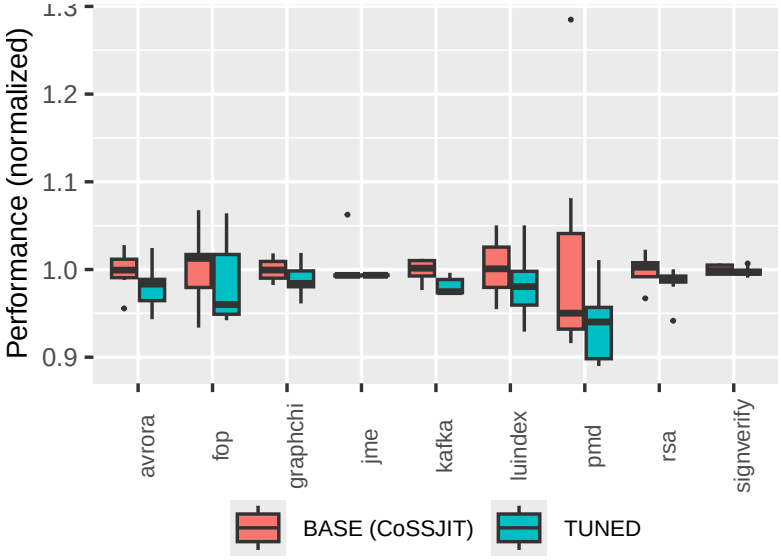


Figure 5.5: Performance at lower heap size.

some of the largest increases in stack-allocation counts. For example, in *fop* the number of stack allocations increases from 152 to 184 (leading to 0.6M dynamic objects), for rsa it increased from 48 to 89 (leading to 14M dynamic objects), similarly for graphchi it increased from 109 to 198 (leading to 4M dynamic objects). Across the entire benchmark suite, the proposed optimizations achieve an average performance improvement of 1.8%. Although the primary objective of this work is to increase stack-allocation coverage to cover the last mile left with a static+dynamic scheme, these results demonstrate that the additional stack-allocation opportunities can also yield tangible performance benefits, while preserving stable performance across a diverse set of workloads.

5.4.6 Impact on Performance at Lower Heap Size

To better understand the impact of increased stack allocation, we also evaluated performance using reduced heap sizes. Allocating objects on stack reduces heap occupancy and garbage collection overhead, which in turn lowers the application's memory requirements. As a result, the benefits of stack allocation are expected to be more pronounced in memory-constrained environments, where smaller heaps increase memory pressure and the cost of heap allocations. For each benchmark, we configured the JVM with the minimum heap size required for successful execution and measured overall performance. Under these memory-constrained configurations, we observed that the performance improvements are

larger than those observed with the default heap size. As shown in Figure 5.5, most benchmarks exhibit improved performance, with particularly notable gains for *avrora*, *fop*, *luindex*, *pmd*, and *rsa*, where we had largest increases in stack allocation. On an average, under the reduced heap size we saw an overall improvement of 2% across all benchmarks. These results indicate that the additional stack-allocation opportunities enabled by our approach yield much better performance gains in memory-constrained environments.

In conclusion, the experiments presented in this section demonstrate that carefully tuning various runtime-dependent factors can significantly improve the effectiveness of a staged static+dynamic optimization scheme. The results demonstrate that the use of statically generated analysis results while accounting for runtime behavior and execution characteristics significantly improves the overall optimization potential. Moreover, although the proposed approach is evaluated for the stack allocation optimization, the same idea can be extended to other compiler analyses and optimizations as well. Thus, this work serves as an example of the broader potential of combining static analysis with runtime-aware optimization decisions.

Chapter 6

Formalizing Static+Dynamic Analysis

This chapter describes a theoretical model for the staged static+dynamic analysis discussed in the previous chapters, based on the classic theory of partial evaluation. First, Section 6.1 highlights how classic partial evaluation and Futamura projections are used to generate specialized program interpreters. Then, Section 6.2 describes *partial analysis* as the process of evaluating dependencies across program elements with respect to the statically available parts of a program, resulting into *partial results*. Next, Section 6.3 devises a strategy (through a novel notion of *AM projections* from Futamura projections) to statically generate specialized evaluators that can process partial results using dynamic dependencies, during JIT compilation. Later, we use our proposed model to straightforwardly establish the correctness and precision properties of the idea of staging, independent of the analysis under consideration. In Section 6.4, we extend our model to soundly handle callbacks made from Java libraries to applications. We demonstrate the applicability of our model by showcasing examples from non-trivial Java program analyses, implementing the pipeline for one of them, and evaluate our prototypes in Section 6.5. Finally, Section 6.6 highlights few of the challenges posed by contemporary programming-language runtimes, possible ways to deal with the same, as well as connections of our staging scheme with few other ways of performing program analysis.

6.1 Partial Evaluation and Futamura Projections

Partial evaluation [37] is a program evaluation technique that specializes a program with respect to its available inputs. The specialized program can take the remaining input[1] and generate the same output as the original program. The program that specializes other programs in this manner is called a *partial evaluator* (traditionally referred to as `Mix`). This way of specializing a program `P` with respect to a statically available input $\texttt{in}_1$ to generate the specialized program $\texttt{P}_{\texttt{in}_1}$ that can take the remaining input $\texttt{in}_2$, often speeds up the overall execution as well. Thus, if the time taken by `Mix` to specialize `P` is $\texttt{T}_{\texttt{Mix}}(\texttt{P}, \texttt{in}_1)$, the time taken by $\texttt{P}_{\texttt{in}_1}$ to generate the final output is $\texttt{T}_{\texttt{P}_{\texttt{in}_1}}(\texttt{in}_2)$, and the time taken by the original program `P` to generate the output in a single run is $\texttt{T}_{\texttt{P}}(\texttt{in}_1, \texttt{in}_2)$, then partial evaluation is often advantageous, as:

$$\texttt{T}_{\texttt{Mix}}(\texttt{P}, \texttt{in}_1) + \texttt{T}_{\texttt{P}_{\texttt{in}_1}}(\texttt{in}_2) < \texttt{T}_{\texttt{P}}(\texttt{in}_1, \texttt{in}_2)$$

In context of just-in-time (JIT) compilers, the time spent in performing program analysis gets added to the execution time of the program, thus making whole-program analysis during JIT compilation practically infeasible. Consequently, JIT compilers resort to very imprecise (e.g., intraprocedural) analyses. Thus, motivated by the possible efficiency advantages of partial evaluation, a promising way to obtain whole-program analysis results efficiently during JIT compilation is to perform partial analysis of the statically available program, and then complete the partial results during JIT compilation. In order to establish that this way of staging whole-program analysis across static and JIT compilation is correct, in this thesis, we formalize a theory of partial analysis based on the prior theory of partial evaluation.

We now present an intuitive formulation of partial evaluation, along with the projections proposed by Futamura [32] to describe the generation of various partial evaluators; we extend this formulation to partial analysis in Section 6.2.

[1]Note that at the machine level, there is an interpreter that actually executes the program along with its input; we are simply avoiding verbosity here.

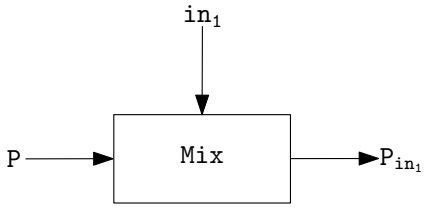


Figure 6.1: Partial evaluation of program P using $\texttt{in}_1$.

6.1.1 Partial Evaluation

Consider the partial evaluation scheme shown in Figure 6.1. For a given program P and its available input $\texttt{in}_1$, the partial evaluator Mix generates the residual program $\texttt{P}_{\texttt{in}_1}$. This partially evaluated program, when given the remaining input $\texttt{in}_2$, yields the same result as running the original program on all of the inputs:

$$[\![\texttt{P}_{\texttt{in}_1}]\!](\texttt{in}_2) = [\![\texttt{P}]\!](\texttt{in}_1, \texttt{in}_2)$$

where $[\![\texttt{P}_{\texttt{in}_1}]\!](\texttt{in}_2)$ denotes the evaluation of $\texttt{P}_{\texttt{in}_1}$ with $\texttt{in}_2$ as input, and $[\![\texttt{P}]\!](\texttt{in}_1, \texttt{in}_2)$ denotes the evaluation of P with $\texttt{in}_1$ and $\texttt{in}_2$ as inputs. The idea behind partial evaluation is that if the input $\texttt{in}_2$ changes more frequently than the input $\texttt{in}_1$, then evaluating the partially evaluated program $\texttt{P}_{\texttt{in}_1}$ on $\texttt{in}_2$ will be faster than evaluating P on the complete input.

Partial evaluation can also be used to generate specialized versions of tools that can generate program evaluators. For example, an interpreter is a program that takes other programs along with their inputs and generates the output for those programs. "What if we use the idea of partial evaluation to specialize an interpreter with respect to a given input program? We get a faster interpreter for that program!" This specialization was described by Futamura as the first Futamura projection (FP), as discussed next.

6.1.2 First Futamura Projection

The first Futamura projection describes how to specialize an interpreter for a given source program; see Figure 6.2 ($1^{\texttt{st}}$ FP). Here, the partial evaluator ($\texttt{Mix}_{(1)}$) essentially applies the interpreter for a language S to a given source program $\texttt{P}_\texttt{S}$ and generates a specialized interpreter, as illustrated by the equation below:

$$[\![\texttt{Mix}_{(1)}]\!](\texttt{Interpreter}_\texttt{S})(\texttt{P}_\texttt{S}) = \texttt{Specialized Interpreter}_\texttt{S}\ \texttt{for}\ \texttt{P}_\texttt{S}$$

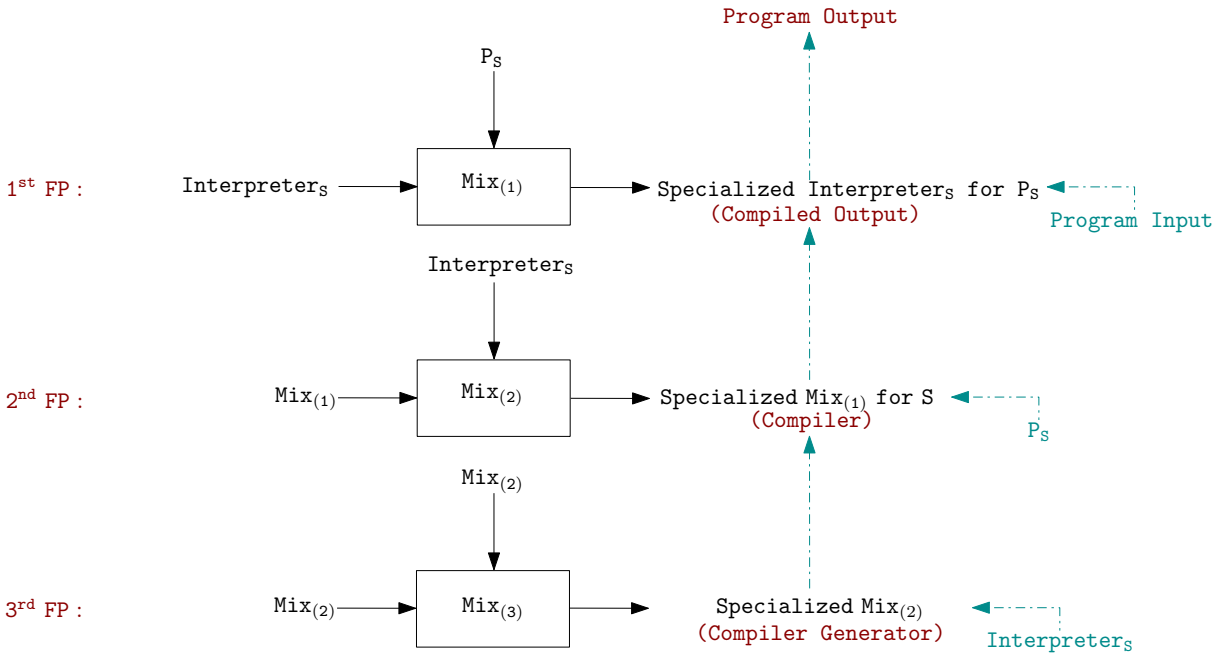


Figure 6.2: Futamura projections in partial evaluation. Note that the three `Mix`es are the same specializers; we have added subscripted numbers for brevity in referencing them.

The generated specialized interpreter can directly take the inputs of the program for which it was specialized and produce the final output. Observe that the behavior of the specialized interpreter is similar to how the binary produced by a compiler (i.e. `Compiled Output`, see Figure 6.2) takes the input of the source program and generates final output. "Can we use the idea of partial evaluation to generate a higher order program, which when given a source program as input, generates its compiled version faster?" This is achieved using the second Futamura projection, as discussed next.

6.1.3 Second Futamura Projection

The second Futamura projection describes how to specialize the specializer ($\texttt{Mix}_{(1)}$) used in first Futamura projection with the interpreter for a given programming language `S`; see Figure 6.2 (2^{nd} `FP`). Here, the specializer ($\texttt{Mix}_{(2)}$) takes the specializer itself as one of the inputs along with the interpreter, and generates a specialized $\texttt{Mix}_{(1)}$ for language `S` as the output, as shown below:

$$[\![\texttt{Mix}_{(2)}]\!](\texttt{Mix}_{(1)})(\texttt{Interpreter}_\texttt{S}) = \texttt{Specialized Mix}_{(1)}\ \texttt{for S}$$

The generated specialized $\texttt{Mix}_{(1)}$ can directly take a program $\texttt{P}_\texttt{S}$ in the language `S` as input

and generate a specialized interpreter for $\texttt{P}_\texttt{S}$. Observe that the behavior of the specialized $\texttt{Mix}_{(1)}$ is similar to how a source program $\texttt{P}_\texttt{S}$ written in a language `S` is compiled. Hence the output of the second Futamura projection can also be called a `Compiler` (see Figure 6.2). "Can we again use the idea of partial evaluation to generate another higher order program, which when given an interpreter for programs written in `S`, efficiently generates a compiler for programs in `S`?" This is achieved using the third Futamura projection, as discussed next.

6.1.4 Third Futamura Projection

The third Futamura projection describes how to specialize the specializer ($\texttt{Mix}_{(2)}$) used in second Futamura projection with itself; see Figure 6.2 ($3^{\texttt{rd}}$ `FP`). Here, the specializer ($\texttt{Mix}_{(3)}$) takes the specializer itself as both the inputs, and generates a specialized $\texttt{Mix}_{(2)}$ as the output, as illustrated by the equation below:

$$[\![\texttt{Mix}_{(3)}]\!](\texttt{Mix}_{(2)})(\texttt{Mix}_{(2)}) \texttt{ = Specialized Mix}_{(2)}$$

The generated specialized $\texttt{Mix}_{(2)}$ can directly take an interpreter for a language `S` as input and generate a compiler for programs written in `S` as the output. Hence the specialized $\texttt{Mix}_{(2)}$ can also be called a `Compiler Generator` for programs written in the language `S`. Note that it is possible to extend this idea further and describe a fourth Futamura projection [33] to generate a compiler-generator generator, and so on.

In a nutshell, the idea of partial evaluation can be used to automatically generate specialized tools in the program translation ecosystem. Though we could not find a standard implementation of the specializer `Mix`, Jones [37] describes it as a two-phase process: first a *division prepass* classifies program inputs into `static` and `dynamic`, followed by which the division and the static inputs are used to *compress* the program, to the extent possible, statically. Further, note that though higher levels of Futamura projections do make sense, the literature finds practical use primarily of the first projection [42], and sometimes the second projection [10]. We next highlight how even staged partial analysis is similar to partial evaluation, and then come up with novel projections that allow one to stage a whole-program analysis into static and dynamic components.

6.2 Staged Partial Analysis

As described in Section 2.6.1, a promising way to avoid incurring the cost of performing precise (whole-program) analysis during JIT compilation is to first analyze the available program statically, and then complete the results when the statically unavailable (or *dynamic*) dependencies are available (could be done either during program execution in a VM, or possibly for each version of the unavailable program, i.e., libraries, ahead of time). In one way, this implies that the evaluation of conditional values for a given analysis ψ, as performed by the conditional-value evaluator $\texttt{CEval}_\psi$, has to be split across static and dynamic components. The consequence of this splitting (or *staging*) is that the static analysis can only compute *partial results*. Such a static analysis, which works on part of the whole program, is called *partial analysis*, and the corresponding module to perform partial analysis can be called a *partial analyzer*. Subsequently, the partial results generated by the partial analyzer need to be completed by resolving the dynamically available dependencies. This in turn requires a *special evaluator* that can take partial results along with the evaluated values of dynamic dependencies, to generate final analysis results.

As one of the key contributions of this thesis, we now present a novel description of the process of performing partial analysis using statically resolved conditional values, followed by a series of specializations to efficiently generate the dynamic component of the conditional-value evaluator.

6.2.1 Partial Analysis

Recall (from Figure 2.2) that computing the final analysis result (of analysis ψ) for a program element x in a method m requires supplying the evaluated values of all the dependencies of x to the conditional-value evaluator $\texttt{CEval}_\psi$. However for languages like Java, several of these dependencies might not be available statically. We now define partial analysis in context of evaluating the set of dependencies available statically; see Figure 6.3 for a list of the notations used throughout Section 6.2.

Definition 1. *(Partial analysis) For a program element x in method m with a set $g_m(x)$ of conditional values, let the set S_x denote the statically available dependencies of x.*

Notation	Description
$g_m(x)$	Set of conditional values for element x in method m
$\texttt{IN}_{g_m(x)}$	Set of all dependencies for evaluating $g_m(x)$
$\texttt{CEval}_\psi$	A conditional-value evaluator for analysis ψ
$f_m(x)$	Final analysis result for element x in method m
S_x	Set of statically available dependencies of x
D_x	Set of dynamically available dependencies of x
$[g_m(x)]_{S_x}$	Set of conditional values specialized with S_x
`Mix`	Specializer program used in partial evaluation
$\mathcal{D}_\psi$	Domain for analysis ψ
$\texttt{Val}_\psi$	Data flow values for analysis ψ

Figure 6.3: A reference to the notations used in rest of the sections.

Here, while trying to evaluate $g_m(x)$, if we supply S_x to the partial evaluator `Mix` *(see Section 6.1), we get the result of partially evaluating $g_m(x)$ with respect to the dependencies present in S_x. This process can also be seen as "specializing" the set of conditional values for the statically available inputs. We formally term this specialization as "partial analysis", illustrated in Figure 6.4.*

When we perform partial analysis of the statically available program, using the schema shown in Figure 6.4, we obtain a specialized set of conditional values ($[g_m(x)]_{S_x}$), which can be termed as the *partial result* for the given element x. For example, in the code shown in Figure 6.5, the set of dependencies for the object O_4, as shown earlier in Equation 2.1, is as follows:

$$g_{\texttt{A.foo}}(O_4) = \{\langle\langle \texttt{A.bar}, \texttt{p1}\rangle, D, D\rangle, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, D, D\rangle,$$
$$\langle\langle \texttt{A.bar}, \texttt{p1}\rangle, E, E\rangle, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, E, E\rangle\}$$

Here, as class `L` is a library class not available for partial analysis, the dependencies in $g_{\texttt{A.foo}}(O_4)$ related to `L.lib` are not available statically. Whereas, the dependencies related to `A.bar` are available, forming the set S_{O_4} (which contains D for both the dependencies as the parameter `p1` of `bar` remains non-escaping. Thus, the partial result $[g_{\texttt{A.foo}}(O_4)]_{S_{O_4}}$

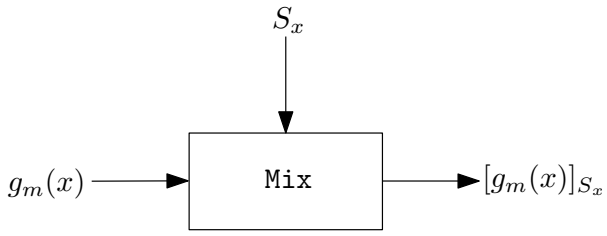


Figure 6.4: Partial analysis: Specializing $g_m(x)$ using the set S_x of static dependencies to obtain partial result.

generated after resolving the statically available dependencies S_{O_4} in Equation 2.1, can be computed as:

$$
\begin{aligned}
&= \sqcap \{D, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, D, D\rangle, D, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, E, E\rangle\} \\
&\qquad\qquad (\because f_n(y) \neq v, so\ \mathcal{T} = \top\ i.e\ D) \\
&= \sqcap \{D, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, D, D\rangle, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, E, E\rangle\} \\
&\qquad\qquad (\because D \sqcap D = D) \\
&= \sqcap \{\langle\langle \texttt{L.lib}, \texttt{r1}\rangle, D, D\rangle, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, E, E\rangle\} \\
&\qquad\qquad (\because D \sqcap X = X, \text{for any CV X}) \qquad (6.1)
\end{aligned}
$$

Thus, the partial result for O_4 is the set $\{\langle\langle \texttt{L.lib}, \texttt{r1}\rangle, D, D\rangle, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, E, E\rangle\}$, and the final analysis result can be computed by taking a meet of the evaluated values of the conditional values present therein. Given such a partial result, we need a special evaluator that can consume the runtime (dynamic) inputs to generate final analysis results for the element x. We now present a series of *AM* (for *Analysis-Mix*) *projections* that generate these special evaluators that can be used to accomplish the same.

6.2.2 First AM Projection

As discussed in Section 6.2.1, the output of performing partial analysis for a given program element x is a partial result (comprising of specialized conditional values $[g_m(x)]_{S_x}$). However, in order to be able to perform any optimization or transformation involving x, we need the final analysis result $f_m(x)$. Thus, we require a new evaluator that can take the partial result $[g_m(x)]_{S_x}$ as input, resolve the residual dependencies based on the evaluated values of dynamically available dependencies (say D_x), and generate $f_m(x)$. We now describe how we can generate such a "partial-result evaluator" for any program element x.

```
1  class A {
2     void foo(B b) {
3        A a1 = new A(); // Object O3
4        A a2 = new A(); // Object O4
5        a1.bar(a2);
6        L l1 = new L(); // Object O6
7        l1.lib(a2); }
8     void  bar(A p1) {
9        // p1 remains non escaping
10    } }
```

```
1  class L {
2     // A library class
3     void lib(A r1) {
4        // A library method
5        ...
```

Figure 6.5: Code snippet from Figure 2.1 reproduced for better readability.

Recall the conditional-value evaluator ($\texttt{CEval}_\psi$) from Figure 2.2, which, when given the set $g_m(x)$ of conditional values for an element x and the set $\texttt{IN}_{g_m(x)}$ of all dependencies of x, generates the final analysis result $f_m(x)$ for the analysis ψ. The first AM projection (see $1^{\texttt{st}}$ `AMP` in Figure 6.6) specializes $\texttt{CEval}_\psi$ with respect to the partial result $[g_m(x)]_{S_x}$. The output is a specialized conditional-value evaluator that can take the set D_x of dynamically available dependencies of x to generate the final analysis result $f_m(x)$. This process of specializing the conditional-value evaluator can be summarized as follows:

$$[\![\texttt{Mix}_{(1)}]\!](\texttt{CEval}_\psi)([g_m(x)]_{S_x}) = \texttt{Specialized CEval}_\psi \texttt{ for } [g_m(x)]_{S_x}$$

Note that the specialized conditional-value evaluator obtained above can be used (see the `Dynamic` module in Figure 6.6) to complete the staging of the whole-program analysis ψ for the program element x. As this evaluator eventually evaluates a given partial result, we name the output of the first AM projection as `Partial-Result Evaluator`.

As an example, for the object O_4 of method `A.foo` from Figure 2.1, consider the partial result obtained in Equation 6.1. Once the analysis result for the library class `L` is available, the `Partial-Result Evaluator` takes D_{O_4} (which is formed by the analysis result available for `L.lib(r1)`) and generates the final analysis result for O_4, as follows:

$$\begin{aligned} f_{\texttt{A.foo}}(O_4) &= \sqcap\,(D, D) \qquad (\text{assuming } f_{\texttt{L.lib}}(\texttt{r1}) = D, \text{ and } \because \top = D) \\ &= D \end{aligned}$$

Thus, the consequence of the first AM projection is not only the fact that it is possible to derive a partial-result evaluator, but also that it can be generated statically. Further,

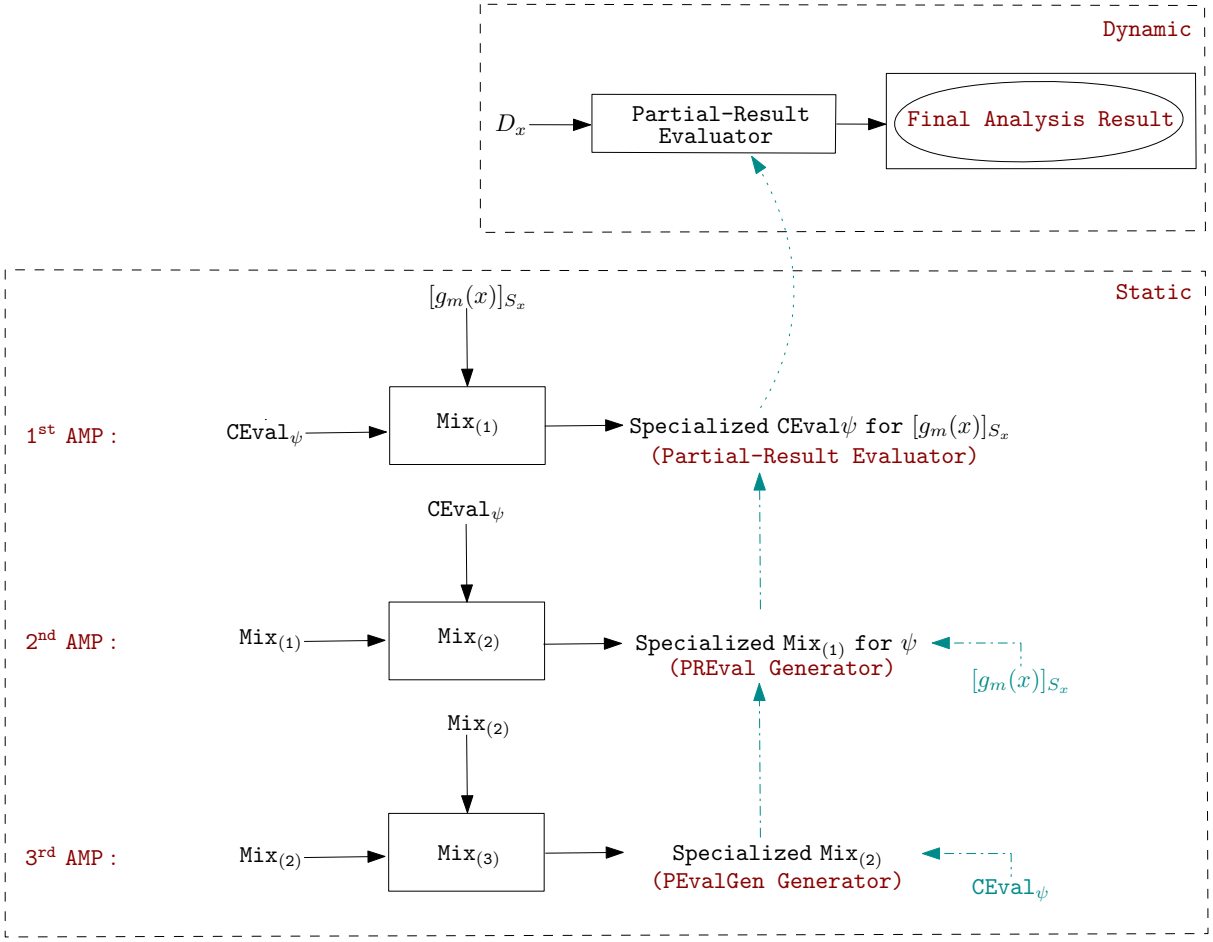


Figure 6.6: AM projections in partial analysis. Note that $\texttt{Mix}_{(1)}$, $\texttt{Mix}_{(2)}$ and $\texttt{Mix}_{(3)}$ are the same specializers as used in partial evaluation; also, we have added subscripted numbers for brevity in referencing them.

the first AM projection also tells that for a given whole-program analysis, there also exists a component that can serve as the generator for such partial-result evaluators (which is the specializer `Mix` from the theory of partial evaluation). Note that the first AMP thus parallels the first Futamura projection; we next show that it is sensible to also extend the latter Futamura projections in the world of partial analysis.

6.2.3 Second AM Projection

Observe that the partial-result evaluator generated by the first AM projection is obtained by specializing the conditional-value evaluator for a single element in the domain of the analysis being staged. However, program analyses often generate results for multiple elements in a given program, implying that one may need to perform this specialization

multiple times. To improve the efficiency of generating such specialized evaluators, we next propose a higher level of specialization, as the second AM projection.

The second AM projection (see $2^{\texttt{nd}}$ `AMP` in Figure 6.6) specializes the specializer $\texttt{Mix}_{(1)}$ itself with the conditional-value evaluator $\texttt{CEval}_\psi$ for the analysis ψ under consideration. This is interesting as such a specializer would be aware of the analysis being performed, instead of being used to apply the common parts of the same analysis to each program element of a given program. The output of this specialization is a specialized mix for ψ that takes the specialized set of conditional values $[g_m(x)]_{S_x}$ for each element x and generates the specialized evaluator $\texttt{CEval}_\psi$ for that x. This process of specializing $\texttt{Mix}_{(1)}$ with $\texttt{CEval}_\psi$ can be summarized as follows:

$$[\![\texttt{Mix}_{(2)}]\!](\texttt{Mix}_{(1)})(\texttt{CEval}_\psi) = \texttt{Specialized Mix}_{(1)}\ \texttt{for}\ \psi$$

As the output of the second AM projection can directly be used to generate the specialized $\texttt{CEval}_\psi$ for each program element x, the second AM projection is a faster way of generating the partial-result evaluator compared to the first. Hence we name the output of the second AM projection as `PREval Generator`. Note that though this generator would give the same partial-result evaluator as the first AM projection, we hypothesize one could adopt the second AM projection in case of time constraints during static compilation.

6.2.4 Third AM Projection

Observe that the `PREval Generator` obtained by second AM projection can generate the partial-result evaluator for any partial result (of the form $[g_m(x)]_{S_x}$), for a particular analysis ψ. However, typical compilers perform many different program analyses. Consequently, in order to perform multiple analyses in a staged manner, one may need to perform the specialization of $\texttt{Mix}_{(1)}$ each time with different $\texttt{CEval}_\psi$, for different analyses being performed by the compiler. To improve the efficiency of generating such specialized generators, we next propose a higher level of specialization, as the third AM projection.

The third AM projection (see $3^{\texttt{rd}}$ `AMP` in Figure 6.6) specializes the specializer $\texttt{Mix}_{(2)}$ with itself. The output is a specialized mix that can take as input the evaluator $\texttt{CEval}_\psi$ for any analysis ψ, and generate the specialized mix for that analysis ψ. This process of

specializing $\texttt{Mix}_{(2)}$ with itself can be summarized as follows:

$$[\![\texttt{Mix}_{(3)}]\!](\texttt{Mix}_{(2)})(\texttt{Mix}_{(2)}) = \texttt{Specialized Mix}_{(2)}$$

A noteworthy point is that just by providing a conditional-value evaluator for any analysis ψ, the specialized mix can directly be used to generate the `PREval Generator` for that analysis ψ. For example, one could provide the conditional-value evaluator for escape analysis, the third AM projection can be used to obtain a module that can generate the evaluator partial-result evaluator for an imbibed program analysis directly. Hence we name the output of the third AM projection as `PREvalGen Generator`.

In a nutshell, the three AM projections describe ways to efficiently generate the dynamic components required for staged static+dynamic whole-program analyses. We can summarize the specializations proposed in the three projections as follows:

1. `Partial-Result Evaluator`
   $= [\![\texttt{Mix}_{(1)}]\!](\texttt{CEval}_{\psi}, \texttt{Partial Result}) = [\![\texttt{PREval Generator}]\!](\texttt{Partial Result})$
2. $\texttt{PREval Generator} = [\![\texttt{Mix}_{(2)}]\!](\texttt{Mix}_{(1)}, \texttt{CEval}_{\psi}) = [\![\texttt{PREvalGen Generator}]\!](\texttt{CEval}_{\psi})$
3. $\texttt{PREvalGen Generator} = [\![\texttt{Mix}_{(3)}]\!](\texttt{Mix}_{(2)}, \texttt{Mix}_{(2)})$

Thus, provided the conditional-value evaluator $\texttt{CEval}_{\psi}$ for a whole-program analysis ψ, a `PREvalGen Generator` can be used to generate a `PREval Generator`, which, when given a statically obtained partial result, can generate the corresponding `Partial Result Evaluator`, which can further be used to obtain the final analysis result given the evaluated values of dynamic dependencies. In Section 6.3, we discuss our implementation of these components using the first AM projection; the higher-order AM projections can be used to generate partial-result evaluators and their generators statically for multiple analyses.

6.2.5 Correctness, Precision, and Efficiency of Staging

In the previous subsections, we have seen how we can stage a whole-program analysis into static and dynamic components, based on ideas taken from the theory of partial evaluation. We now state and prove (by construction) few important properties of such a staging scheme, with respect to the correctness of staging and the precision of the results obtained, and comment on the overall efficiency of the process.

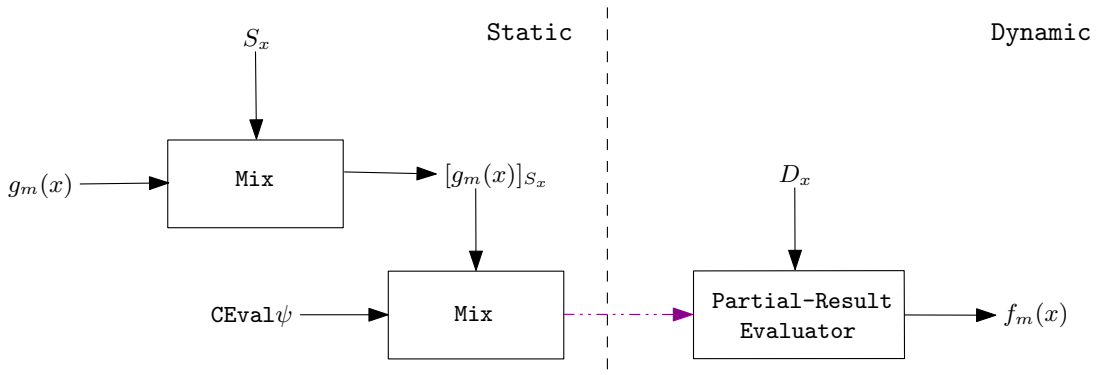


Figure 6.7: A complete staging of whole-program analysis.

Theorem 2. [Correctness and Precision]. *For any program element, the analysis results generated by a whole-program analysis and by the corresponding staged analysis (as summarized in Figure 6.7), are the same.*

Proof. Recall the whole-program analysis schema from Figure 2.2. For a given program element x, its final result $f_m(x)$ after performing a whole-program analysis ψ can be obtained by evaluating the set g_m(x) of conditional values with respect to all the dependencies $\texttt{IN}_{g_m(x)}$. Now consider the staging of the analysis ψ as shown in Figure 6.7. Here, we have broken down the set $\texttt{IN}_{g_m(x)}$ of dependencies into the set S_x of statically available dependencies and the set D_x of dynamically available dependencies. The static component is a specializer (`Mix`) that generates the specialized conditional-value evaluator (`Partial-Result Evaluator`) by specializing the whole-program analysis evaluator $\texttt{CEval}_\psi$ with respect to the partial result $[g_m(x)]_{S_x}$ (which itself is obtained by specializing $g_m(x)$ with S_x).

The generated `Partial-Result Evaluator` forms the dynamic component of the staged analysis, and generates f_m(x) using D_x. As established by the theory of partial evaluation, the result obtained by evaluating a program with respect to all of its inputs is same as the result obtained by evaluating the specialized program with respect to its dynamic input. It can be seen by construction that the way Figure 6.7 stages a whole-program analysis essentially by using partial evaluation in the same way the first Futamura projection specializes a given interpreter for the evaluation of a program: the interpreter being our $\texttt{CEval}_\psi$ and the program used for specialization being $[g_m(x)]_{S_x}$. Hence, the analysis result generated by our staged analysis for all the program elements in the domain D_ψ

of an analysis ψ would be the same as the one generated by the whole-program analysis. This equality can be summarized as follows:

$$\forall x \in \mathcal{D}_\psi \; [\![g_m(x)]\!](\texttt{IN}_{g_m(x)}) = f_m(x) = [\![[g_m(x)]_{S_x}]\!](D_x)$$

□

Discussion. Note that the above discussion makes a *closed-world assumption*; that is, it is assumed that all the interactions across methods (though not necessarily their code) in the program are known statically. If it were to be relaxed in a way that some part of the program was unknown, the above relation may hold true only if the static analysis can approximate all possible interactions that may happen during execution. We show how this assumption could be relaxed for a popular Java feature, namely callbacks, and corresponding modifications to the staging scheme, in Section 6.4.

Efficiency of Staging. The complete premise of a staged static+dynamic analysis is aimed to impart efficiency to otherwise impractical whole-program analysis in a dynamic compiler. Thus, it is important to establish that such a scheme offloads all the computation that can be performed statically to the static phase of the scheme. We first explain this using a notion of *maximality*, and then state the efficiency theorem of our scheme.

Consider a program element x whose dependencies $g_m(x)$ are captured by n conditional values $\mathcal{T}_1, \mathcal{T}_2, ..., \mathcal{T}_n$. Among these, say one of the dependencies $\mathcal{T}_k$ is a dynamic dependency and the others are static. Without loss of generality, the final analysis result $f_m(x)$, for an analysis ψ will be computed as:

$$f_m(x) = [\![\mathcal{T}_1]\!] \sqcap_\psi [\![\mathcal{T}_2]\!] \sqcap_\psi ...[\![\mathcal{T}_k]\!]... \sqcap_\psi [\![\mathcal{T}_n]\!]$$

Assuming that the meet operation $\sqcap_\psi$ is commutative, what we expect after the static phase is the following:

$$[g_m(x)]_{S_x} = v \sqcap_\psi [\![\mathcal{T}_k]\!]$$

where v is the meet of the evaluated values of all the dependencies other than $\mathcal{T}_k$. Obtaining the above makes the dynamic phase more efficient than (i) evaluating all the n dependencies; and (ii) computing a pairwise meet of the evaluated dependencies. We capture this fact using the notion of *maximality*, defined next.

Definition 2. [**Maximal Specialization**]. *As formulated by Jones [37], a partial evaluator specializes a program* `P` *with respect to some input* $\texttt{in}_1$ *by precomputing all expressions that depend on* $\texttt{in}_1$*, and then folding* `P` *to the extent possible, that is, in a loop until fixed point. Thus, the output of a partial evaluator is* maximal *in terms of evaluation of the program for the given set of inputs. We call a program specialized using such a partial evaluator* maximally specialized.

Definition 3. [**Maximal Conditional-Value Evaluation**]. *The evaluation of a set* $g_m(x)$ *of conditional values is maximal, iff (i)* $\nexists \mathcal{T} \in [g_m(x)]_{S_x}$ *s.t.* $\mathcal{T} \in S_x$*; and (ii)* $v_i, v_j \in \texttt{Val}_\psi \wedge v_i, v_j \in [g_m(x)]_{S_x} \implies i = j$.

The first part of the definition states that the partial result contains no static dependency (that is, one that can be evaluated statically). The second part implies that while computing the partial result, no statically performable meet operation has been left out (if there are two or more values from the set $\texttt{Val}_\psi$ in the partial result, then we can take their meet statically itself). In other words, constructing and using a partial-result evaluator for staged analysis through partial evaluation ensures that we have (i) evaluated all the static dependencies while generating the partial result; and (ii) computed a meet of all the evaluated dependencies, statically itself.

Theorem 3. *For a given program element and its statically available dependencies, the partial-result evaluator obtained by the first AM projection is maximal in terms of the conditional-value evaluation that can be performed statically.*

Proof. Follows from Theorem 2 (equivalence of first Futamura projection and our static+dynamic staging scheme) and Definition 2 (maximality of partial evaluation), for a given set of statically available dependencies passed in for generating the partial-result evaluator. □

Theorem 2 establishes the correctness and precision of the proposed staging scheme, and Theorem 3 establishes its efficiency (indicating maximal evaluation during static compilation). Observe that the proofs became straightforward due to two important properties (i) that a whole-program analysis could be modeled as the evaluation of dependencies across program elements; and (ii) that for languages like Java with statically

unavailable program parts, the evaluation of static and dynamic dependencies could be modeled based on the theory of partial evaluation.

We next describe our experience implementing a specializer for generating partial-result evaluators from given conditional-value evaluators for the whole program, similar to the `Mix` described by Jones in his classic book on partial evaluation. We approach this problem by describing a language of conditional values, such that programs in the world of partial analysis become sets of conditional values and program interpretation becomes conditional-value evaluation.

6.3 Implementing Specializers for Partial-Result Evaluation

In this section, similar to a language of programs, we first describe a language of conditional values that could be used to generate sets of conditional values (denoting dependence on various kinds of elements) for different program elements (Section 6.3.1). Next, similar to program interpreters, we design a simple evaluator that could resolve those dependencies to generate the final analysis result for various program elements. We then mirror the process of specializing a program interpreter by designing a `Mix` that could specialize conditional-value evaluators to generate partial-result evaluators (Section 6.3.2). We present extensions to this mechanism for handling callbacks later in Section 6.4 and discuss our experience compiling and running these generated partial-result evaluators in Section 6.5.

Note that though our efforts are independent of the program analysis being staged as long as it follows the criteria mentioned in Section 2.6.4, we need a set of conditional values generated for a given program analysis. In this section, we have chosen a publicly available conditional-value generator for escape analysis, `Stava` [62], written in Soot [67]. `Stava` generates a list of dependencies for each abstract object (program element), denoting its dependence on other program elements towards computing its escape status: one of *Escapes* (E) and *DoesNotEscape* (D).

```
<Start>     ::=  <ProgElem> :  <CV>*

<ProgElem>  ::=  <Class> <Method> <Type> <Ref> <Fields>

<CV>        ::=  <ProgElem> <DepVal> <ResVal>

<Type>      ::=  LOCAL | PARM | ARG | RETVAL | FIELD

<DepVal>    ::=  D | E

<ResVal>    ::=  D | E
```

(a)

```
<Start>     ::=  <ProgElem> :  <TaggedCV>*

<TaggedCV>  ::=  <Tag> <CV>

<Tag>       ::=  STATIC | DYNAMIC
```

(b)

Figure 6.8: (a) The grammar for our division prepass; (b) Extended grammar for conditional-value evaluation.

6.3.1 A Grammar for Conditional Values

Based on the kinds of elements on which the analysis result for a given program element could depend on, Figure 6.8 shows a grammar to generate sets of conditional values (denoting those dependencies) for various program elements. Each program element `ProgElem` belongs to a `Class` and a `Method`, and could be of one of the five `Type`s: (i) local object in current method; (ii) parameter taken by current method; (iii) argument passed to another method; (iv) return value of current method; and (v) field of any other element. `Ref` is a number, denoting the line number of allocation for `LOCAL`, parameter and argument number respectively for `PARM` and `ARG`, and simply a filler for the rest. `Fields` contains a list of fields (e.g., `f1.f2` to denote the element pointed to by `X.f1.f2` for any abstract object `X`). A conditional value (`CV`) denotes dependence on a program element. Essentially, a conditional value `<P1 X1 X2>` evaluates to `X2` if the element `P1` resolves to `X1`; see Section 2.6.2. For example, for escape analysis, `DepVal` and `ResVal` could either be `D` or `E`. Thus, pairs comprising of program elements and the conditional values generated for those elements (e.g., by `Stava`) are the valid members of the language generated by this grammar. (Note that we had to make cosmetic changes in `Stava` to print its output

in a form that can be parsed by our conditional-value grammar.)

6.3.2 Conditional-Value Evaluators and Specialization

Specialization using the theory of partial evaluation uses an auxiliary "division" routine to classify program inputs as static and dynamic. Having described a language of conditional values in the previous section, we next wrote a division prepass that classifies each conditional value as static or dynamic (based on whether it denotes a library dependency while analyzing applications, and vice-versa). We have implemented the division prepass as a JavaCC [61] visitor (about 300 lines of code) over the abstract syntax trees generated for the sets of conditional values prescribing to the grammar described above. We encode the result of division by prefixing each conditional value with a tag `STATIC` or `DYNAMIC`, resulting into a set of conditional values recognizable using the extended conditional-value grammar shown in Figure 6.8(b). This extended grammar describes the language of conditional values that can be evaluated by our conditional-value evaluator $\texttt{CEval}_\Psi$. Algorithm 3 gives an overview of the computation performed by $\texttt{CEval}_\Psi$. The evaluator takes as input the set of conditional values $g_m(x)$ for a program element x belonging to a method m along with the analysis results for all the dependencies contained in $\texttt{IN}_{g_m(x)}$, and generates the partial result $[g_m(x)]_{S_x}$. First, the evaluator intializes a list L with all the dependencies of the program element x, that is, dependencies in $g_m(x)$ at line 1. Then if the dependency value present in $\texttt{IN}_{g_m(x)}$, the evaluator replaces the dependency with the resolved value otherwise the evaluator transitively adds all the dependencies of the given element into the list L (lines 2-6). Next, treating the various dependencies as a graph, the evaluator forms strongly-connected components (SCCs), denoting sets of equivalent resolved values. In case no element in an SCC depends on an element from another SCC, all the elements in that SCC are resolved to the top (most precise value) of the lattice of the analysis under consideration (lines 10-11). Otherwise if the element y in the dependency resolves to v then we resolve the dependency to v', where v and v' are the elements of the lattice. Finally, for the program element x, the evaluator takes a meet as described in Section 2.6.3, generating partial result in case of dynamic dependencies (line 17). The last two steps (lines 10 to 17) are performed till a fixed point.

Algorithm 3: Algorithm to perform conditional-value evaluation.

Data: $g_m(x)$: Set of dependencies of the form $\langle\langle n, y\rangle, v, v'\rangle$ for x

$\texttt{IN}_{g_m(x)}$: Analysis results for all the dependencies.

Result: $[g_m(x)]_{S_x}$: Partial result for x.

```
1  Initialize a list L with all the dependencies from g_m(x) for program element x.
2  foreach d ∈ g_m(x) do
3      if d ∈ IN_{g_m(x)} then
4          Replace d with the resolved value from IN_{g_m(x)} in L.
5      else
6          Add all the transitive dependencies of d to L.
7      end
8  end
9  Form strongly connected components (SCCs) in the list L.
10 repeat
11     foreach strongly connected component S formed above do
12         if ∄e ∈ S s.t. e depends on another SCC then
13             ∀e ∈ S, resolve e to the most-precise element ⊤.
14         else
15             if y in e resolves to v then
16                 resolve e to v', where v and v' are elements in the lattice.
17             end
18         end
19         Take a meet of the resolved values in each SCC: ⊓_{s∈SCC}[[s]].
20     end
21 until fixed point
```

Our evaluator is also implemented as a JavaCC pass over the extended conditional-value grammar (from Figure 6.8(b)), and spans about 1000 lines of code. Note that the only part of the evaluator that depends on the analysis for which the conditional-values are generated is the meet operation (to access its lattice); thus, the evaluator is essentially parametric over the analysis being performed. Hence, for escape analysis (ea), we denote the evaluator as $\texttt{CEval}_{ea}$.

Next we have implemented a `Mix` that takes as input our conditional-value evaluator

```
1  class SpecializedCode1 {
2     public static void main(String[] args) {
3        // Read the values for dynamic dependencies
4        x1 = Resolved value of <L.lib, r1> // first dependence
5        x2 = Resolved value of <L.lib, r1> // second dependence
6        res = x1 ⊓_ea x2
7        print(res);
8     }
```

Figure 6.9: Schema of the partial-result evaluator emitted for $g_{\texttt{A.foo}}(O_4)$.

and a partial result $[g_m(x)]_{S_x}$, and specializes the evaluator for the given partial result. Our `Mix` works similar to the partial-evaluation Mix proposed by Jones [37], but for Java programs of the kind of $\texttt{CEval}_{ea}$ (we could not find any existing implementation that we could reuse). Our `Mix` is implemented in JavaCC (for a grammar that covers the subset of Java required to parse $\texttt{CEval}_{ea}$), and spans about 1500 lines of code. The output of our `Mix` is a partial-result evaluator for the given partial result.

The fundamental idea behind specializing the evaluator is that its code should be executed as much as possible for the static dependencies, and the residual code should take the evaluated values of dynamic dependencies as input to generate the final analysis-result. Observe that Line 11 in Algorithm 3 involves taking the meet (over the lattice `Val`) of the resolved dependencies. However, if any dependence is dynamic, its resolution cannot be performed statically. Hence, in order to specialize the evaluator for a given partial result, we first check the kind of dependence (populated by the division prepass), and for each dynamic dependence, we emit code to read and resolve the same. Next, we emit code to perform meet over the resolved values obtained therein. Finally, we enclose the emitted code as the `main` method of a uniquely named specialized-code class (say `SpecializedCodeN` for a unique `N`), to obtain the corresponding partial-result evaluator. For example, Figure 6.9 shows the schema of the code emitted by our `Mix` for the element $g_{\texttt{A.foo}}(O_4)$ (see Equation 6.1, Section 6.2.1).

Before we could use the partial-result evaluators generated for different program elements, we need to obtain the evaluated values of the dynamic dependencies (libraries in case of Java). For a particular Java installation, the dependencies within the libraries can be generated and evaluated independent of the application. We do so by running

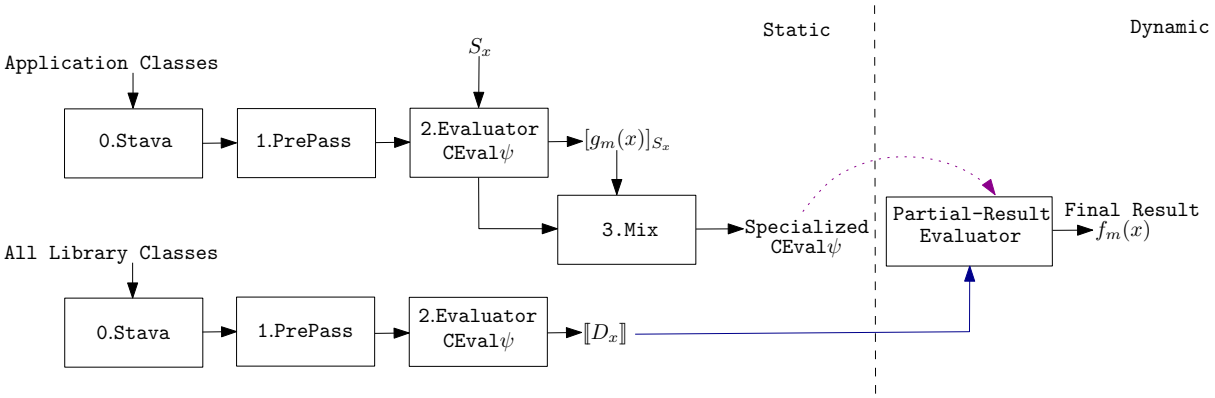


Figure 6.10: Schema for prototype implementation of staged analysis.

an offline pass of $\texttt{CEval}_{ea}$ on the libraries, and obtaining the evaluated values therein. Finally, one can supply these evaluated values to the specialized codes comprising the partial-result evaluators for various program elements, to obtain the final analysis-result. Figure 6.10 shows the complete scheme of our prototype implementation for generating and executing partial-result evaluators.

Observe that partial-result evaluation over the evaluated values of dynamic dependencies, as per our model, can be performed by executing the specialized code as soon as the dynamic dependencies can be resolved. Thus, we have the following options:

- Simply place the partial-result evaluators in a JIT compiler.
- As the partial-result evaluators are generated statically for all the program elements, place them in a VM agnostic to the methods/code-portions that are JIT compiled.
- For the evaluated dependencies corresponding to each version of the libraries, invoke the partial-result evaluators ahead-of-time (i.e., statically itself).

The first option can be mapped to the kind of scheme adopted by staging frameworks such as PYE [64]; however, it would mean we can perform optimizations for only those elements that belong to methods/regions that are JIT compiled (usually very few in tiered runtime systems such as the HotSpot JVM [52]). On the other hand, the second option could be used to obtain analysis results in the VM irrespective of the tiered nature and mode of compilation (by implementing corresponding optimization passes). Finally and most interestingly, the third option makes the process of generating analysis results even independent of the runtime system, and can also be used to demonstrate the full impact a staging scheme could make over performing a whole-program analysis.

6.4 Incorporating Callbacks from Libraries

The generic staging scheme presented in previous sections makes a closed-world assumption and analyzes applications and libraries independent of each other. An important Java feature that violates this assumption and needs to be handled more carefully is that of possible *callbacks* from libraries. With the possibility of Java library methods calling back application code, there could be edges in the call graph that get ignored while performing a static analysis without the source code of the whole program available (a common scenario in modular Java programs where applications and libraries are maintained separately). In this section, we extend our staging scheme to address the possibility of missing flows induced by potential runtime callbacks, which allows the generated results to be sound and precise with respect to a whole-program analysis, even in presence of callbacks for Java-like languages.

Consider the code snippet shown in Figure 6.11. `A` is an application class, containing two methods `foo` and `bar`. As the escape status of the object O_7 in `bar` depends on the first parameter `q` (due to the field store at line 8), the static analysis component would generate the conditional values {`<A bar ARG 1 [] D D>`, `<A bar ARG 1 [] E E>`}. Additionally, the parameter `q` depends on the argument passed as `p` to `foo` (as it is loaded in `foo` from `p.f`). Thus, if no escaping argument is ever passed to `foo`, our staging scheme, based on a statically available call graph, would map O_7 to `D`. However, since class `A` implements the `Callback` interface, `foo` can be called back from the library method `lib` with an escaping object as an argument, causing the staging scheme to risk an unsound result for O_7. We now propose an extension to the staging scheme presented in Sections 6.2 and 6.3 to generate sound results in presence of callbacks. We do this by extending the notion of conditional values with a way to record dependencies on possible callbacks made by Java libraries to applications (Section 6.4.1). We also extend our staging scheme to perform residual evaluation in two stages: first to resolve dynamic dependencies (minus for callbacks) as usual, and second to resolve *callback dependencies* at runtime (Section 6.4.2). Later in Section 6.5, we discuss the tradeoff between achieving precision with these additional callback dependencies and obtaining sound but conservative results in their presence efficiently. We begin by first extending our conditional-value grammar from Section 6.3 with callback dependencies.

```
1  class A implements Callback {
2    void foo(X p) {
3      Y m = p.f;
4      this.bar(m);
5    }
6    void bar(Y q) {
7      Z r = new Z();O7
8      q.g = r;
9    }
10 }
```

```
11 interface Callback {
12   void foo(X p);
13 }
14 class Library {
15   static Callback global;
16   static void lib(Callback cb, X p) {
17     global = p; // p Escapes
18     cb.foo(p);  // callback
19   }
20 }
```

Figure 6.11: Example to demonstrate the impact of callbacks.

```
<Type>  ::=  LOCAL | PARM | CALLBACK | ARG | RETVAL | FIELD

<Tag>   ::=  STATIC | DYNAMIC | CALLBACK
```

Figure 6.12: Extensions to the grammar from Figure 6.8 to handle callbacks.

6.4.1 Computing Callback Dependencies During Static Analysis

We handle callbacks by identifying the program elements that may be dependent on potential callbacks made from the library to the application, and recording them as additional dependencies. Figure 6.12 shows the extended conditional-value grammar that can be used to parse such "callback" dependencies. Essentially, we have introduced a new `Type` of dependency called `CALLBACK` that encodes the potential callback a particular program element depends on; `REF` for callback dependencies (see Figure 6.8) refers to the corresponding parameter of the called back method. For example, the conditional values {`<A foo CALLBACK 1 [] D D>`, `<A foo CALLBACK 1 [] E E>`} can be used to record the dependency of O_7 on the possible callback made to `foo` in Figure 6.11. Further, in the division prepass, we classify callback dependencies separately from the remaining dynamic dependencies, by introducing a new `Tag` called `CALLBACK`.

We next describe how do we add callback dependencies to the program elements that may be affected by potential callbacks made by Java libraries; see Algorithm 4. We have extended the conditional-value generator `Stava` to identify these elements and add callback dependencies described above to the set of conditional values for the same. The

procedure addCallbackCVs maintains a data structure named callback_affected to record methods that may be affected by callbacks. To begin with, lines 2-4 identify library-overridden or implemented methods in the application classes, and mark all their parameters as potentially callback-affected. This is because only those application methods can directly be called back by library methods. Next, for all the callback-affected methods, for each program element that depends on any of the formal parameters, lines 6-9 add a callback dependency on the parameter to the program element. Finally, if any element e with a callback dependency is passed to another method, we add the dependency to the program elements pointed to by e (lines 10-12). We perform the previous two operations till a fixed point, in order to add callback dependencies to all the potentially affected program elements in all the application classes. For example, for the code shown in Figure 6.11, assuming that `foo` is a library-overridden method, the transitive fixed-point nature of `addCallbackCVs` would add the callback dependencies {`<A foo CALLBACK 1 [] D D>`, `<A foo CALLBACK 1 [] E E>`} to the set $g_n(O_7)$ of conditional values for the object O_7.

Having described the generation of callback dependencies, we now move on to discuss the extensions to the staging schema required for their resolution.

6.4.2 Staging Schema with Callback Dependencies

In addition to the conditional values described in Section 6.3, the modified conditional-value generator presented in Section 6.4.1 additionally generates callback dependencies for the program elements potentially affected by callbacks. Thus, our staging schema from Figure 6.10 needs to be extended to handle these additional dependencies. Note that as against the evaluated dynamic dependencies $[\![D_x]\!]$, the set of callback dependencies (say C_x) cannot be evaluated statically (as callback is a runtime property). Thus, schematically, at runtime, the partial-result evaluator can be supplied both – $[\![D_x]\!]$ and $[\![C_x]\!]$ – to generate the final analysis results $f_m(x)$. However, interestingly, we can again use the idea of partial evaluation to further speed up this process in typical virtual-machine based Java runtimes.

Figure 6.13 shows our extended schema to efficiently resolve standard dynamic dependencies and callback dependencies in a staged manner. We introduce a specializer (`Mix`) component that takes the partial-result evaluator along with $[\![D_x]\!]$ as input and generates

Algorithm 4: Adding callback dependency to the program elements.

```
1  foreach application class c do
2      if c overrides or implements a library method m then
3          Mark m as callback_affected.
4      end
5  end
6  repeat
7      foreach callback_affected method m do
8          foreach program element e ∈ m do
9              if e has dependency on a formal parameter p of m then
10                 Add a callback dependency on p to g_m(e).
11             end
12         end
13     end
14     foreach program element e with a callback dependency 𝒯 do
15         if e is passed as an argument q to a method n then
16             Add 𝒯 to g_n(O_q), where O_q ∈ ptsto(q).
17             Mark n as callback_affected.
18         end
19     end
20 until fixpoint
```

a specialized partial-result evaluator (say `SPRE`). The SPRE is specialized for a given set of dynamic dependencies, and can generate the final analysis results provided the evaluated callback dependencies ($[\![C_x]\!]$).

In a typical VM-based Java runtime (such as the HotSpot JVM [52]), one can perform the specialization of the statically generated partial-result evaluator along with $[\![D_x]\!]$ in a JIT compiler (as the libraries on a particular JVM are fixed). Followed by this, the generated SPRE can be invoked while actually running the JIT-compiled program, by supplying the evaluated value of callback dependencies from the runtime (could be done by generating code to detect callbacks). This would ensure that in addition to the minimal

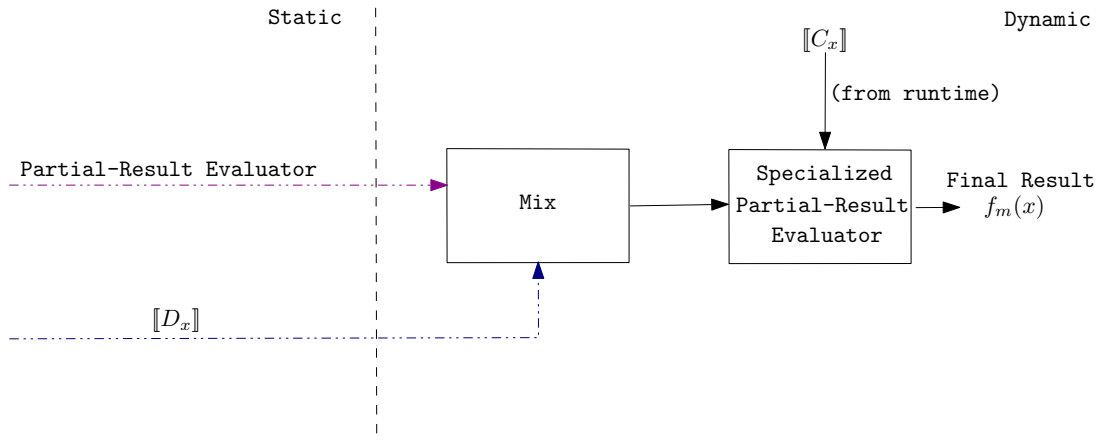


Figure 6.13: Specializing the partial-result evaluator with $[\![D_x]\!]$

dynamic evaluation guaranteed by the first AM projection, a minimal amount of evaluation is to be performed post JIT compilation. We state this as our callbacks-extended maximality theorem below.

Theorem 4. *For a given program element and its dynamic dependencies, the specialized partial-result evaluator obtained in Figure 6.13 is maximal in terms of the conditional-value evaluation that can be performed during JIT compilation.*

Note that apart from the extensions to the dynamic components described in Figure 6.13, we also need to make two minor modifications in the static components: (i) we extend the division prepass to classify the dependencies for each program element as `STATIC` (S_x), `DYNAMIC` (D_x), as well as `CALLBACK` (C_x); (ii) we modify $\texttt{CEval}_\psi$ such that the generated partial result contains dependencies from both D_x and C_x.

6.5 Evaluation

In this section, we present an evaluation to validate the ideas proposed in this chapter. Our primary goal is to demonstrate that staging a whole-program analysis in the proposed manner significantly reduces the computation one may have to perform dynamically. We do this by taking escape analysis as the use case, and measuring the time taken as well as the amount of computation performed by each stage. We also measure the impact of handling callbacks soundly, instead of ignoring them altogether.

We have evaluated our approach on ten benchmarks (those that run successfully with

TamiFlex) from the DaCapo suite [14] version 9.12. The conditional-value generator Stava is built using Soot [67] version 3.1. Our division prepass, the conditional-value evaluator, and the specializer Mix are written using JavaCC version 7.0.11 for parsing the conditional values and generating the partial-result evaluators. Additionally, we use TamiFlex [16] version 2.0.3 to resolve reflective calls while constructing call graphs in Soot. We have also used two small benchmarks from the Java Grande Forum (JGF) suite [25] to manually verify our results. Our experiments have been performed on a 3.00 GHz Intel(R) Xeon(R) system with 48 cores and 100 GB of memory, running Ubuntu 20.04.4 LTS and Java 8 (to ensure compatibility with TamiFlex).

We now describe the computation performed (in terms of time taken and dependencies resolved) by the static components of our staging scheme.

6.5.1 Computation Performed by Each Stage

Recall Figure 6.10 that shows the various steps of our proposed staging scheme. For any given application, the first step is to generate a set of conditional values across different program elements (Step 0). Then these conditional values are classified into `STATIC` and `DYNAMIC` in a division prepass (Step 1). The statically available values are then fed to the conditional-value evaluator, which generates a partial result for each program element in the application (Step 2). Finally, a specializer Mix generates a specialized evaluator for each partial result (Step 3). Figures 6.14 and 6.15 list the number of dependencies (indicating the amount of computation) and the time taken by each step, respectively.

6.5.2 Computation Performed Statically

Columns 2 and 3 of Figure 6.14 respectively show the number of application methods and the total number of program elements for each application under consideration. As can be seen, few of the DaCapo benchmarks contain a very high number of methods and program elements (e.g. fop and eclipse), whereas the programs included from the JGF suite (moldyn and rtracer) are smaller ones that we have used for manual validation. Column 2 of Figure 6.15 shows the time taken by Step 0 (Stava) to generate the dependencies for all these program elements; recall that this step is independent of our staging scheme.

Column 4 in Figure 6.14 shows the total number of dependencies (static as well as

Bench-mark	**App Methods**	**#PE**	$S_x + D_x + C_x$	% D_x	**#PE with** D_x	% C_x	**#PE with** C_x
avrora	527	13344	20562	16.5	2675	0.7	147
batik	949	36211	57924	18.2	7036	3.6	2086
lusearch	199	9054	14179	19.5	2119	2.3	327
luindex	230	11146	17588	20.8	2865	1.2	202
pmd	762	35445	61336	14.6	6318	2.3	1416
sunflow	142	7580	10607	14.0	1159	1.0	104
h2	325	27610	40603	17.4	5297	3.2	1314
xalan	586	30684	51572	11.8	4520	4.0	2084
fop	1116	61030	90276	20.3	14730	2.3	2045
eclipse	2029	163633	262215	10.5	20084	1.6	4089
moldyn	11	473	679	30.1	174	16.1	109
rtracer	19	580	894	27.8	213	7.7	69
GMean	279	13291	20589	17.7	2787	2.6	531.9

Figure 6.14: Number of program elements (PE) and various kinds of dependencies.

dynamic) for each program under consideration. Column 5 shows the percentage of these dependencies that are identified as dynamic by Step 1 of our approach, the time taken by which is shown in Column 3 of Figure 6.15. On an average, we see that 17.7% of the overall dependencies cannot be resolved statically; this indicates the potentially high loss of precision in case we were to use a completely static analysis.

The dependencies generated and classified until Step 1 are next fed to Step 2, which evaluates the statically resolvable dependencies to generate partial results; the time taken to perform this static evaluation is shown in Column 4 of Figure 6.15. We can see that the time taken by Step 2 generally increases with the number of program elements and dependencies, as well as with the amount of recursion among the dependencies. Column 6 of Figure 6.14 shows the number of program elements with residual dynamic dependencies. As can be observed, on an average, for 21% of program elements (2787 out of 13291,

GMean), we cannot generate a final analysis result statically.

The last static step is to specialize the conditional-value evaluators with the above computed partial results (Step 3), the time taken for which is shown in Figure 6.15, Column 5. Even though this is a pretty fast static step, it generates specialized code that can be directly placed in a VM runtime to obtain final analysis results for the program elements with residual dependencies discussed above. Apart from performing all the above steps for individual applications, we also did the same for various Java class libraries and stored the results separately. For example, static evaluation for library classes corresponding to JDK 8 took nearly 6 hours on our evaluation setup. Among the times required for our static steps, we found that parsing the dependencies using JavaCC during steps 1 and 2 took nearly 60% of the overall time for those steps; this can be improved with a faster parsing mechanism in future.

To compare the time taken by our staged scheme with the time required to perform whole-program analysis, we chose a particular version of libraries, generated all the conditional values for each benchmarks, and statically performed evaluation of the same in entirety; the last column of Figure 6.10 shows the time consumed by this "whole-program analysis". As can be seen, the analysis did not terminate in 4 hours for any of the DaCapo benchmarks, and it took more than 2 hours for the smaller benchmarks moldyn and rtracer. This trend indicates the infeasibility of performing a whole-program analysis, particularly in JIT compilers, and strengthens the argument that staging is the way to go.

6.5.3 Computation Performed Dynamically

Given the partial-result evaluators generated by the static analysis for each program element x of method m, the next step of our approach is to supply the evaluated values of dynamic dependencies ($[\![D_x]\!]$, see Figure 6.10) for obtaining final analysis result $f_m(x)$. Column 6 in Figure 6.15 shows the time spent in this stage for computing the final analysis result for all the program elements in an application. On an average, this time (85.9 seconds) is a much smaller fraction of the amount of time a whole-program analysis might take at runtime; for example, we tried to analyze a whole program version of our smallest benchmark moldyn (that is, supplying all dependencies statically for a given library) and found that it terminated in 1 hour 40 minutes (as against 4 seconds at runtime

Bench-mark	**Step 0 (Stava)**	**Step 1 (Prepass)**	**Step 2 (Eval)**	**Step 3 (Mix)**	**Final Result** $f_m(x)$	**WPA (hrs)**
avrora	16.3	0.2	10	1.0	58.5	-
batik	144.8	1.0	100	1.1	375.2	-
lusearch	8.5	0.1	4	1.0	39.3	-
luindex	11.12	0.2	5	1.0	50.0	-
pmd	127.4	1.0	3370	1.1	269.0	-
sunflow	7.3	1.1	3	0.9	17.9	-
h2	53.8	0.4	96	1.1	228.5	-
xalan	104.3	0.9	79	1.2	206.5	-
fop	325.3	1.0	301	1.4	714.9	-
eclipse	2547.1	4.7	3790	1.5	1406.6	-
moldyn	9.2	0.1	1.0	0.7	4	2
rtracer	8.3	0.1	1.1	0.8	4.1	2.5
GMean	45.2	0.4	32.8	1.1	85.9	-

Figure 6.15: Time taken (in seconds) by various steps of Figure 6.10; WPA shows the Whole-Program Analysis time in hours (a '-' indicates that the analysis did not terminate in 4 hours).

with our staging approach). This shows the amount of difference a staging scheme such as ours can make to achieve the precision of a whole-program analysis. Note that the time taken dynamically can be reduced even more by utilizing the architecture of a particular VM-based runtime; for example: (i) obtain results for only those program elements that belong to code that is JIT compiled; (ii) store the partial results in class files instead of reading them from separate files; (iii) evaluate the dependencies in parallel, and so on.

Impact of callbacks.

As discussed in Section 6.4, the staging model discussed previously ignores the possibility of callbacks made from libraries to applications. To handle the same, based on the extended schema shown in Figure 6.13, we modified the static analysis to additionally

identify callback dependencies (negligible increase in the time taken by division prepass) that are left to be resolved dynamically. Column 7 of Figure 6.14 shows the percentage of callback dependencies for each application. As discussed in Section 6.4, these callback dependencies have to be tied to a particular runtime that can provide information about the occurrence of a callback (as $[\![C_x]\!]$) to generate the final analysis result for the affected program elements. Notably, as this step will have to be performed based on a live execution profile of a program, one can also take an alternative route and generate conservative (but sound) results for the program elements having callback dependencies. We computed the potential precision loss for this alternative; see Figure 6.14, Column 8. As can be seen, on an average, by resolving callback dependencies conservatively, we would lose precision for a small proportion of program elements (4%, GMean), and still generate sound results as against a scheme that ignores the possibility of callbacks statically. We leave the choice to the users of our staging scheme in a particular Java runtime.

Overall, noting the improvement in the potential time required during run-time using the staged approach, we conclude that staging partial analyses into static and dynamic components, backed by a theoretical model presented in our thesis, not only opens up avenues in languages with managed runtimes to perform existing optimizations more aggressively than present, but can also be used to perform novel analyses and optimizations that were otherwise practically infeasible.

6.6 Directions and Connections

Having drawn parallels between the classic theory of partial evaluation and a promising way to stage program analyses into static and dynamic components, we now turn our attention on how to extend an existing staged analysis, handle various recent features and challenges pertaining to contemporary programming languages (as described for Java in Chapter 3. In particular, we address topics of interest that could lead to further work built upon our model of partial analysis for languages with static and dynamic compilers. We also discuss connections with few other ways of performing program analysis in managed runtimes. In effect, we believe this discussion would be useful in driving multiple future directions, not only for the Java world but also for the wider community interested in program analyses staged across static and dynamic components.

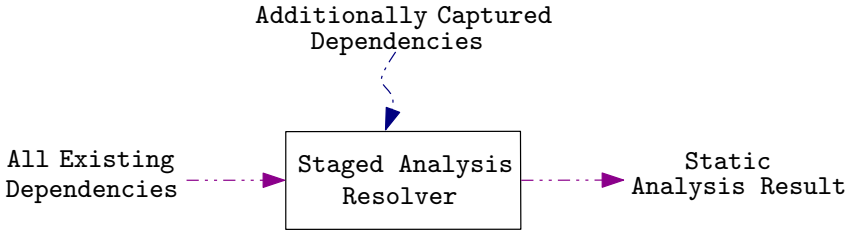


Figure 6.16: Extending a staged static+dynamic scheme with additional dependencies such as speculative conditions.

6.6.1 Extending Staged Analysis with Speculation

Existing staged static+dynamic analysis can be extended to support additional features for any given analysis. A typical static analysis computes results by iterating towards a fixed point solution; however, this approach often leads to conservative results and complicates integration of new attributes during the computation. In contrast, modeling static analysis using the idea of dependency in the form of conditional values facilitates an approach to generate analysis results specific to individual attributes. This allows one to generate partial results when an attribute cannot be fully resolved statically, rather than generating conservative results. The idea is that the analysis can be incrementally enhanced with new attributes, provided that the new attribute can be expressed or represented in terms of dependencies. As shown in Figure 6.16, an existing staged analysis can be extended by capturing additional dependencies which can be supplied to a run-time resolver that can generate final results.

Connection with `CoSSJIT` and Callbacks: A similar observation can be made for the approach adopted by `CoSSJIT` proposed in Chapter 4. `CoSSJIT` built on the underlying static analysis by encoding additional dependencies that correspond to speculative assumptions that are resolved at the run-time. Similarly, as discussed in Section 6.4, the proposed approach adds potential callback edges into the existing static analysis by extending it to generate dependencies that capture the effects that may occur dynamically at run-time. In both the cases, these extensions integrate naturally into the analysis framework proposed in this chapter, because the analysis operates over a set of dependencies, allowing new forms of dependencies to be introduced without fundamentally altering the underlying analysis structure. Consequently, the staged analysis framework provides a general and extensible mechanism for incorporating newer analyses and run-time behaviors, provided

they can be expressed in terms of dependencies.

6.6.2 Runtime Features: Challenges and Possibilities

In this section, we discuss few challenges posed by modern static+dynamic compilation systems, and discuss the kinds of techniques that could be used to improve the precision of the analysis results obtained therein.

As described in Section 2.6.1 (see Equations 2.2 and 2.3), standard resolution and conditional-value evaluation assume that all the dependencies can be resolved at run-time (that is, *closed world assumption*). However, there could be scenarios because of the way tiered JIT compilation works in modern JVMs where the results for few dependencies may not be available at the point of need. For example, if the program being analyzed uses reflective calls, state-of-the-art static-analysis frameworks (such as Soot [67]) miss edges in the call-graph for the program, as a result of which few methods may not get analyzed altogether. On another note, features such as dynamic classloading [40] in JVMs may bring up classes in a statically non-deterministic order. For example, in the partial result for element O_4 (see Equation 6.1), if class `L` is not loaded when the corresponding evaluation is invoked, the conditional values dependent on `L1.lib` cannot be resolved. The only possible sound option in this case would be to take the least precise value E as the resolved value.

On the other hand, for analyses such as control-flow analysis that have a richer lattice, it is possible to improve the precision for unresolvable dependencies. Consider the code snippet shown in Figure 6.17. In method `A.foo`, the set of conditional values representing the possible types of objects that can be pointed-to by the reference variable `z` is $\{\langle\langle \texttt{A.foo}, \texttt{b}\rangle, \texttt{B}, \texttt{Z1}\rangle, \langle\langle \texttt{A.foo}, \texttt{b}\rangle, \texttt{C}, \texttt{Z2}\rangle\}$. Under the existing evaluation scheme, where all the required dependencies are available, the type of `z` will get assigned to either `Z1` if `b`'s type is `B`, or to `Z2` if `b`'s type is `C`. However, say in presence of dynamic classloading, if class `C` has not been loaded when the conditional values for `z` need to be evaluated, then assuming the least precise value of the lattice (which, for control-flow analysis, is the set of all types in the program) is highly conservative and may affect many other optimizations such as method inlining and virtual-call resolution. We now discuss a solution to address this problem.

```
1  class A {
2     void foo(B b) {
3        Z z = b.bar();
4     }
5  }
6
7  interface Z {...}
8  class Z1 implements Z { ... }
9  class Z2 implements Z { ... }
10 class Z3 implements Z { ... }
```

```
11 class B {
12    Z bar() {
13       return new Z1() ;
14    }
15 }
16 class C extends B {
17    Z bar() {
18       return new Z2();
19    }
20 }
```

Figure 6.17: A Java code snippet to demonstrate control-flow analysis.

While performing partial-result evaluation, in case a certain dependency cannot be resolved, instead of always falling back to the least precise value in the lattice, we can statically generate some "fallback values" that can be used as the fallback option during resolution. Observe that the set of possible values that can be obtained as the analysis result for a given program element x, can be formed only from the third elements (say resolution values) of the tuples present in the set of conditional values for x. Thus, we can obtain a fallback value $fb(x)$ by taking the meet of all such resolution values:

$$fb(x) = \sqcap_{\mathcal{T} \in g_{m(x)}} \mathcal{T}[3]$$

where $\mathcal{T}[3]$ gives the third element in each conditional value. As an example, for the reference variable `z` in Figure 6.17, $fb(z)$ would be the set {`Z1`, `Z2`}.

In order to support fallback values, the staging scheme presented in Section 6.2 (see Figure 6.7) can be modified as follows. In the static component, for each element x, apart from the partial result $[f_m(x)]_{S_x}$, we additionally need to store $fb(x)$. On the other hand, in the dynamic component, we can modify the partial-result evaluation as:

$$f_m(x) = \sqcap_{\mathcal{T} \in g_{m(x)}} [\![\mathcal{T}]\!]$$

$$[\![\langle\langle \mathtt{n}, \mathtt{y}\rangle, v, v'\rangle]\!] = (is\_available(f_{\mathtt{n}}(y))) \ ? \ ((f_{\mathtt{n}}(\mathtt{y}) == v) \ ? \ v' \ : \ \top) \ : \ fb(x)$$

Here, for a given element x, while trying to resolve a conditional value $\mathcal{T} = \langle\langle n, y\rangle, v, v'\rangle \in g_m(x)$, we first check if $f_n(y)$ is available; if yes, we proceed with normal resolution, else we use $fb(x)$ as the fallback value. Thus, for the example shown in Figure 6.17, even if

the runtime has no information about the caller of `A.foo` (that is, no knowledge about the type of the objects pointed-to by `b`), using statically computed $fb(\texttt{z})$, we can resolve the conditional values for `z` generated above to obtain the set {`Z1`, `Z2`} as the analysis result for `z`.

Apart from the challenges posed by runtime features discussed above, another important consideration for static+dynamic analyses is to guarantee/verify that the static-analysis results correspond to the bytecodes being executed. In case of a difference, one may need to invalidate the partial-result evaluator for the corresponding and dependent methods. This can at a simpler level be done by maintaining a list of affected methods with the statically resolved dependencies, and be improved by precisely identifying the effect on various program elements. We believe this to be an interesting future research direction.

6.6.3 Drawing Newer Connections

In this section, we first discuss few interesting aspects related to cross-pollination of ideas between Futamura and AM projections, along with few subtle points related to generation of conditional values in partial analysis. We then highlight how our staged analysis scheme can be used along with various other applications involving static and dynamic analyses.

1. Cross-pollination of specialization ideas. We have mentioned previously that AM projections are similar to yet different from Futamura projections. To elucidate this point, note that the partial-result evaluator generated by the first AM projection is a result of specializing the conditional-value evaluator for an already specialized set of conditional values (see Figure 6.6). On the other hand, the output of the first Futamura projection is a result of specializing the interpreter for the original source program (see Figure 6.2). It is possible to take cue from the first AM projection and modify the first Futamura projection to specialize the interpreter too with a partially evaluated program. Doing so would generate a faster interpreter, as the input program can anyway be specialized with the statically available input. Similarly, the compiler generated by the second Futamura projection can also take a specialized program as input to efficiently generate the specialized interpreter obtained in the first Futamura projection.

As another possibility to explore in the space of specialization, it can be seen in Section 6.2 (Figures 2.2 and 6.7) that the input taken by our model of whole-program as

well as partial analyses is the set of conditional values, denoting dependencies, for a given program element. It is possible to visualize the process of generating these conditional values: from a given program analysis specified as an abstract interpreter, we can identify the set of statements required to compute the final analysis result for a particular program element x, as the dependencies of x. This process of generating conditional values can be made faster: one could model program analyses as "conditional-value generators", and then specialize them with respect to a given program, similar to the specialization of conditional-value evaluators done by AM projections. Also note that though this "per element" modeling is a bit different from the way usual iterative dataflow analyses [49] are implemented (as aggregate transfer functions over all the variables in the domain), it fits well with various recently popular ways of writing program analyses, as discussed next.

2. Query- and feedback-driven analyses. In general, whole-program analyses generate results for all the program elements in all the methods of a program. On the other hand, one of the growingly popular ways to scale precise analyses in resource-constrained environments is to compute information only for elements that are of interest, often specified as a set of queries generated by various client optimizations [56, 35]. These analyses are called "query-driven analyses". Staged schemes of the kind proposed in this chapter can be integrated directly with such analyses: First, the client can generate the list of program elements that are of interest for the query under consideration, based on which the partial analysis can generate the relevant sets of conditional values. Afterwards, our staging scheme can be used to specialize the corresponding set of conditional values and further generate the `Partial-Result Evaluator` only for the elements of interest.

For languages that support static+dynamic compilation, one of the ways to improve the outcomes produced by static analyses is to perform profiling in the dynamic compiler and give feedback to the static compiler [12, 69, 26]. As an instance, Bastani et al. [12] make optimistic assumptions for the unavailable portions of a program, detect during runtime if an assumption goes wrong, abort execution if it does, and then refine the static analysis accordingly; this process is repeated until no assumptions fail. The staged scheme of our kind suits such analyses particularly well: the feedback from runtime can be used to obtain the list of affected elements that need to be re-specialized by the static analysis for subsequent runs of the re-created partial evaluator, thus requiring the (partial) static analysis to be re-performed only for the affected elements.

To summarize, in this work we propose a formal model to stage whole-program analyses into static and dynamic components. Our staging scheme took inspiration from the theory of partial evaluation and specialized the evaluators for whole-program analysis with partial results to generate partial-result evaluators. Similarly, based on the notion of Futamura projections for partial evaluation, we proposed a novel notion of AM projections that describe the generation of partial-result evaluators and their generators. The generated partial-result evaluators can also be placed in managed runtimes to generate final analysis results using the dependencies that become available dynamically. This model allowed us to establish the correctness and precision of the idea of staging in a straightforward manner. Moreover, in order to address the challenges presented by the dynamic nature of modern tiered runtimes, we also discussed possible future directions to extend the staging scheme and showed a possible extension to handle Java callbacks. To the best of our knowledge, ours is the first scheme that backs the staging of whole-program analysis into static and dynamic components, using the established theory of partial evaluation.

Overall, we observe that the theory of partial evaluation, apart from its standard use in designing efficient code generators, has an important use in performing staged program analysis as well. This way of performing analyses is getting popular in languages with staged compilation systems, and modulo the proportion difference among static and dynamic features can also be used for languages that allow far more dynamism than Java. Thus, we envisage that our formulated theory of partial analysis would be used not only to promote the design of staged partial analyzers, but to also perform erstwhile infeasible optimizations, for and beyond Java.

Chapter 7

Related Work

The main idea of this thesis is using static+dynamic analysis soundly within a managed runtime environment and to improve its precision using speculation. To the best of our knowledge, ours is the first work that uses static analysis to perform stack allocation in a VM while supporting a dynamic repair mechanism involving heapification and stack ordering. Although existing works involving JVMs do perform escape analysis and the resultant optimizations of various kinds, they do not provide such a unified and sound framework. These approaches are discussed in detail in this section. More broadly, the challenges of achieving both precision and efficiency in JIT-based analyses are well recognized, and a substantial recent research has focused on solving these challenges. In this context we discuss on the approaches that range from partial to feedback-driven static analysis, to incremental and offline analysis of JITted code. Finally, we discuss partial evaluation and its various applications, along with relevant work on partial-program analysis, staged analysis, and modular analysis.

7.1 Escape Analysis

One of the first extensive proposals to perform escape analysis for Java was implemented in the Jalapeño VM [5, 70], which introduced the points-to escape graph notation for denoting points-to relationships as well as performing reachability-based escape analysis. Another popular abstraction for performing escape analysis [19], is that of connection graphs, which do not maintain points-to relationships directly but allow one to perform

reachability checks faster. A partially interprocedural version of this approach is used by the C2 JIT compiler of the HotSpot VM [53] to perform synchronization elision and scalar replacement. In contrast, the proposed approach in this thesis for improving the partially interprocedural escape analysis of Eclipse OpenJ9 [30] is context as well as flow sensitive, and uses points-to graphs as they additionally allow us to determine stack orders (which will not be possible with connection graphs). Further, the escape analysis used in this thesis is performed statically, which avoids overheads during JIT compilation.

Kotzmann and Mössenböck [40, 41] proposed an escape analysis that works in presence of dynamic classloading for the C1 compiler of the HotSpot JVM. Essentially, they reallocate objects replaced by scalars if the VM deoptimizes to the interpreter. The more recent GraalVM builds up on this idea and performs scalarization of objects while rematerializing them in branches where they escape [60]. As some of the stack-allocated objects can also be replaced with scalars, it would be interesting to complement our implementation with scalar replacement. Interestingly, the current VM-only versions of these works require additional bookkeeping in order to materialize objects, which can possibly be reduced if (part of) the analysis was performed statically like our approach. We mark this as an interesting future extension.

The closest related works are those by [22] and [21]. The first one proposes an address check at write barriers to detect objects that live longer than the stack frame being destroyed, but performs stack allocation primarily around loops and by maintaining a separate region for object allocation on stack. The second one proposes to address dynamic classloading that may cause previously non-escaping objects to escape, by modifying the classloader and maintaining special dirty bits on the stack frame, but has no evaluation. The proposed optimistic approach, on the other hand, covers dynamic features including dynamic classloading, and does not require modifications to the classloader, stack management, or to any other traditional routines of the JVM; further, our work is supported by an extensive evaluation over a production JVM.

A recent work proposed a framework [64] for performing expensive analysis statically and using its results during JIT compilation, in context of libraries that may differ between static and dynamic compilation, and another work formalized the idea of staging program analyses across static and JIT compilation [8]. Similar to them, we use bytecode indices to represent static-analysis results that are usable in a JVM. However, adapting

their approaches for stack allocation would pose the exact problem that we solve in this thesis: timely invalidating the results in presence of dynamic features such as hot-code replacement, dynamic classloading, and even the possibility of different libraries. Nevertheless, we are encouraged by this trend and believe static+dynamic analysis has many further applications that are yet to be explored.

An alternative approach of optimizing memory, popular particularly in the functional programming community [65], is to divide the run-time memory into inferred regions that can lead to faster access and cleanup of objects in the current region (a region may include multiple stack frames). Region inference is more expensive than escape analysis and is not employed by current object-oriented runtimes. However, we feel approaches like ours can be used to improve the efficacy of region-based memory allocation and open up the possibility of experimenting with interesting stack and heap allocation strategies, for popular OO runtimes as well.

7.2 Speculation and Runtime Tuning

Few prior works have attempted to leverage run-time profiling information to enable speculative optimizations [28]. These approaches typically perform the required analyses dynamically in order to determine when such speculative optimizations should be applied. Since many of these analyses can be computationally expensive, existing techniques often adopt conservative strategies to limit run-time overhead. In contrast, the `CoSSJIT` approach proposed in this thesis performs expensive analysis statically ahead of time and augments the generated results with additional information. The runtime system then uses lightweight profile-guided decisions to determine when the corresponding optimizations should be applied.

With respect to tuning and improving runtime heuristics, an abstract interpretation based strategy [50] was proposed for estimating the benefits of inlining and guiding the inliner in selecting appropriate callees. The approach presented in this thesis builds upon that idea but for a specific optimization by incorporating additional runtime-dependent factors into the tuning process. The results suggest that, depending on the optimization being targeted, several other aspects of the runtime system can also be adapted to further maximize the effectiveness of a staged static+dynamic optimization scheme.

7.3 Imparting Efficiency to Precise JIT Analysis

Since the introduction of JIT-based runtimes for popular programming languages, the de-facto approach of program translation has been to only achieve platform independence statically (e.g. by generating Java Bytecode or .NET CIL), and to perform optimizations exclusively in the JIT. However, acknowledging the limitations of resources available to a JIT, and the absence of a bridge to take advancements in static analyses to the JIT world, recent works proposed the usage of partial program analysis [24] to statically generate *conditional dataflow values* that can be resolved during JIT compilation [64, 8]. Our notion of *speculative conditions* can be seen as extending the kinds of conditions a static analyzer could generate, though the reason prior works introduced such values was completely different – unavailability of parts of a program during static analysis.

In recent years, building systems that can use offline analysis has picked up a lot of interest, particularly with the introduction of AOT compilers in HotSpot and OpenJ9 VMs, and the development of Native Image in GraalVM [71]. The former approach allows specializing portions of programs beforehand, aimed primarily at reducing warmup times, but fails to deliver peak performance due to absence of JIT speculation. Whereas the latter approach allows offline analysis of code produced in JITted executions (e.g. [17]), it restricts the enabled optimizations to a closed-world assumption. A recent work proposed a two-level dispatcher that uses previously compiled JIT code and allows program to change in future executions, in context of the R programming language [45]. We believe such approaches can be extended with the offline analysis attempts made on Native Image, and mark this as a promising direction for future.

While AOT analyses can be used to drive aggressive optimizations, a major challenge is to either ensure safety before using their results, or to detect violations and ensure functional correctness during execution. We are aware of two recent attempts on solving this problem, one from each category. Akin to bytecode verification in JVMs, a recent work proposed fast in-VM verification of static-analysis results, and used them to eliminate guards associated with virtual calls [34]. Similarly, another work proposed timely detection and repair of incorrect stack allocations during execution [6]. In the thesis, we chose to demonstrate our novel approach of integrating static (AOT) analysis with JIT speculation for stack allocation itself, and hence used the publicly available implementa-

tion of the latter idea. However, for enabling other optimizations using our approach, one could use the ideas proposed in the former as well.

Though we have primarily restricted our discussion here to works that use AOT information in JIT compilers, there have been relevant within-JIT attempts to increase precision too. As an example, a close cousin of speculative stack allocation is the notion of decomposing objects initially and materializing them later in paths where they escape – called *partial-escape analysis* in GraalVM [60] and *cold-block heapification* in OpenJ9. While these ideas are also affected by the standard interprocedural limitations of JIT analysis, we mark a detailed empirical comparison (and possible static-analysis versions of these ideas) as interesting future work.

Recognizing that many static analyses written for modern programming languages fail to handle dynamic features (or handle them incorrectly [43]), a recent work proposed the notion of *eventual soundness*, which iteratively obtains sound results for Java pointer analysis [11]. Such feedback-driven ideas are gaining traction in the JavaScript community as well [31], where it is difficult (or extremely imprecise) to statically model the shapes of objects and consequently enable well-known AOT analyses and optimizations. We believe interesting fusions of our idea with these approaches would be a promising direction to explore, and may lead to notions of *eventual precision* that could push JIT optimizations in dynamic languages to the next level.

7.4 Partial Evaluation and its Applications

Partial evaluation is a well-known technique to specialize programs with statically available inputs. Jones [37] formalized the theory of partial evaluation in context of constructing compilers and compiler generators by specializing subsequent levels of interpreters, using Futamura projections [32]. Perugini and Williams [55] underlined the difficulty in understanding partial evaluation and Futamura projections, and devised a diagrammatic approach to visualize the working of partial evaluation, by modeling program execution using a box-substitution notation. In the thesis, we have also tried to explain the three Futamura projections, particularly by showing the connections among the outputs and inputs of the subsequent projections. Our goal behind this visualization is to later build a mapping from our proposed model of partial analysis, to generate partial-result evaluators.

There have been works that use partial evaluation to speed up different parts of a program's execution lifecycle. One such implementation [42] speeds up the execution of code during JIT compilation by specializing the AST interpreter for a given language specified in the Truffle [73] framework. This avoids redundancy in code generation during JIT compilation. Marr and Ducasse [44] compare the performance of the previous approach with that of tracing JIT compilation (which optimizes a program by JIT-compiling traces obtained by profiling). On the other hand, in this thesis, we have used the idea of partial evaluation to speed up the process of obtaining whole-program analysis results (possible during or just before JIT compilation). Our scheme can be augmented to the Truffle approach by performing partial analysis of the available program during AST specialization, and refining the results during JIT compilation.

7.5 Partial Program Analysis

The idea of analyzing partial programs was first formalized in a tool called PPA [24], wherein the goal was to infer types for incomplete Java programs. In presence of ambiguities, PPA uses heuristics based on the structure of a program to generate imprecise but sound results. Similarly, Melo et al. [46] generate missing type annotations for incomplete C programs, while handling challenges imposed by C's weak type system using "placeholder pre-types". In presence of ambiguities, Melo et al. fill the placeholder with an "orphan" type in the lattice of pre-types. Our staged scheme, though performs an analysis of the partial program to generate partial results statically, is able to generate sound as well as precise answers based on dynamic inputs, wherein the components needed to evaluate and complete partial results are generated statically, using novel AM projections based on the theory of partial evaluation.

There have been prior works [48, 74, 54] that generate results for incomplete programs by trying to obtain possible solutions based on examples and then ranking them for suitability. The limitation of these techniques is that they may generate unsound results. Our staged scheme, on the other hand, would always generate sound results, with varying precision based on the dynamic features supported by a given runtime.

Allen et al. [4] present a scheme to analyze Java libraries in absence of application code. They model concrete objects using allocation sites, while approximating unknown

objects using static types, thus generating a combined lattice. Our approach, though different in the sense that it works for both application and library code, uses the idea of using static types as fallback values in absence of dynamic inputs. On the other end of the spectrum, Ali and Lhoták [3] generate call-graphs to analyze Java application code in absence of libraries, by approximating library methods as stubs. In comparison, our staging approach, instead of approximating the libraries, uses their analysis results (obtained offline) to complete application results during dynamic compilation.

7.6 Staged and Modular Analysis

Staging, though a general idea, has not often found place in static+dynamic analysis systems. Chug et al. [20] compute and check information-flow properties for Javascript programs statically, while leaving residual checks that depend on dynamic inputs for the runtime. Albarghouthi et al. [1] develop specifications for unknown methods in context of program synthesis. In context of Java, Thakur and Nandivada [64] proposed the PYE framework, which uses the idea of conditional values to denote dependencies on dynamic input and resolves them to generate final results during JIT compilation. In this thesis, we have proposed a general staging framework based on the theory of partial evaluation that can be used not only to model such approaches, but also to establish and prove the correctness and precision of the same. Importantly, our base scheme (Section 6.2) is independent of the language and framework under consideration, and the extensions for Java runtimes (Section 6.6) can be used to devise corresponding strategies for other runtimes with novel dynamic features.

Various approaches have been introduced to analyze different parts (or modules) of a program independently and then to compose the *summaries* of the different modules to form the results for the whole program. Cousot and Cousot [23] propose the idea of separating the analysis for various parts of a program and present its formal semantics. Whaley and Rinard [70] devised a compositional pointer analysis for Java programs, which can not only be incrementalized [69] but be also used to skip the analysis of certain regions of the program. Recently, Utture and Palsberg [66] use this idea to identify a subset of the relevant library for answering queries about an application and skip the time-consuming parts without generating unsound results. Similar to these prior works, our thesis targets

program analysis that is modular and compositional; on the contrary, none of these works have tried to model modular analysis in a way that some modules are analyzed statically and the others dynamically.

7.7 Handling Dynamic Features

Most existing program analyses implemented in managed runtimes are implemented wholly in JIT compilers, and as a virtue of the environment, can handle dynamic features (albeit with an imprecise analysis due to resource constraints). Kotzmann and Mössenböck presented a strategy to handle features such as dynamic classloading in the HotSpot C1 compiler [40]. Whereas Tang et al. present a library summarization approach that allows to complete dependence analysis in presence of callbacks at run-time [63]. In this thesis, we presented an optimistic approach using a staged static+dynamic model, that can be used to handle callbacks or any other dynamic features offered by the language or the runtime.

Chapter 8

Conclusion and Future Work

Static and dynamic program analyses have historically been at two ends of a spectrum: researchers keep coming up with advancements to enhance the precision of static-analysis abstractions, and practitioners keep designing novel engineering techniques to keep language runtimes efficient. However, seldom do the ends meet, which is evident by the choice of imprecise analyses employed by JIT compilers to balance the tradeoff with efficiency.

This thesis picked up an important OO optimization – that of allocating method-local objects on the stack frames of their allocating methods – and showed that using a static escape analysis to optimistically allocate identified objects on stack could improve the precision without thwarting the efficiency. Further, in order to ensure functional correctness in case the static-analysis results do not correspond to the run-time environment, it proposed the ideas of (i) dynamically checking incorrect stack allocations before they could cause an issue; (ii) repairing the memory layout by heapifying escaping objects and correcting their references; and (iii) doing so efficiently by ordering the objects on stack in a statically identified manner. It evaluated the approach on a production JVM across its interpreter and compiler infrastructure, and showed that the proposed scheme is capable of bringing the benefits of a precise escape analysis for improving performance in low-memory systems, by reducing memory pressure and correspondingly, the garbage-collection overheads.

Next the thesis proposed an approach to combine the best parts of statically performed AOT analysis and dynamically performed JIT analysis. The hypothesis was that both have their benefits – static analyses have resources to be more intricate and JIT analyses

have information about the run-time behavior – and that it is possible to perform the former while leaving holes that can be filled in during the latter, essentially generating performant compiled code in a managed runtime. To present a concrete demonstration of our hypothesis, the thesis identified the sources of imprecision in static escape analysis that can be enhanced with run-time information, and then made the static analysis aware of those with the notion of speculative conditions. These conditions captured the possibility of enhancing the precision of statically generated results using the speculation performed during JIT compilation, with respect to polymorphic callsites, method inlines, and branch executions. Further, it added the resolution of these conditions in the JIT compiler of a production VM – using various run-time structures such as the dynamic class-hierarchy table, the inlining table, and receiver-type and branch profiles – and then used them to perform aggressive stack allocation. It measured the enhancement in stack allocation as well as its impact on performance, and found them to be significantly higher than the state-of-the-art implementation.

Towards formalizing the idea of static+dynamic analysis, the thesis presented a formal model to stage whole-program analyses into static and dynamic components. The staging scheme took inspiration from the theory of partial evaluation and specialized the evaluators for whole-program analysis with partial results to generate partial-result evaluators. Similarly, based on the notion of Futamura projections for partial evaluation, the thesis proposed a novel notion of AM projections that describe the generation of partial-result evaluators and their generators. The generated partial-result evaluators can be placed in managed runtimes to generate final analysis results using the dependencies that become available dynamically. This model established the correctness and precision of the idea of staging in a straightforward manner. Moreover, in order to address the challenges presented by the dynamic nature of modern tiered runtimes, the thesis also discussed possible future directions to extend the staging scheme and showed a possible extension to handle callbacks in Java.

Overall, we feel that sound and efficient static+dynamic approaches like the ones proposed by this thesis, with a theoretical backing, open up possibilities to speed up and improve the precision of various analyses and optimizations, not only for Java Virtual Machines but also for similar languages and runtimes such as C# and the .NET framework, as well as for more dynamic languages such as JavaScript, R and Python.

8.1 Limitations

While the proposed static+dynamic approach can be used for several optimizations, it may incur additional overhead in the presence of certain language features or program behaviors. We next discuss some of the key challenges associated with this approach.

1. Challenges with reflection: Performing static analysis in the presence of reflective calls remains a significant challenge, as reflection often introduces unsoundness into the generated analysis results. Since reflective targets are typically resolved dynamically at run-time, accurately determining the set of possible callees during static analysis is difficult. One commonly used approach for handling reflection is to record the possible target methods observed at reflective call sites during execution and subsequently use this information to update the call graph for future analysis runs. This technique is employed by the TamiFlex tool [16]. All the implementation part of this thesis utilize TamiFlex to handle reflective calls while generating static analysis results.

However, the effectiveness of Tamiflex depends heavily on the completeness and accuracy of the collected runtime information. If certain calls are not captured during execution, or certain JVM features such as hidden classes cannot be adequately modeled, the resulting call graph may remain incomplete and could lead to unsound results. Nevertheless, in context of stack allocation, the heapification mechanism proposed in this thesis can handle such unsoundness, but with some additional runtime overheads.

2. Challenges with phase change in the speculative approach: The `CoSSJIT` approach proposed in this thesis enables the JIT compiler to perform more aggressive stack allocation by incorporating statically generated speculative assumptions with run-time profiling information. However, any optimization strategy that relies on run-time profiling information can get affected by phase changes in program behavior. Run-time profiles capture execution characteristics observed during a particular phase of execution. However, these characteristics may evolve over time as the application workload changes. Thus, it may happen that the trend shown in the profiles during optimization no longer holds in the later execution of the program. In such scenarios, performing optimization based on run-time profiles could lead to increased deoptimization and potentially reduce the overall effectiveness of the optimization strategy. Therefore, determining when and how aggressively to optimize using speculative assumptions remains a critical challenge while staging.

8.2 Future Work

It would be quite interesting to extend the applicability of the static+dynamic approach for other popular analyses (such as control-flow analysis, dependence analysis, and other related analysis), while ensuring the availability of a sound fallback option for each such analysis. Integrating static analysis with possibility of speculation at run-time has potential to yield significant performance improvement across all theses analyses.

With respect to the idea of object allocation, it would be interesting to explore region-based allocation strategies for objects. In particular, one possible direction is to identify the caller method where exactly escaping objects are captured and allocate such objects within a specialized region. Additionally, it would also be interesting to create new regions within the heap to allocate objects that get heapified, as well as modify the behaviour of the underlying garbage collection strategy for handling such objects more efficiently.

Another promising direction is to implement the staged static+dynamic idea within a JIT Server [39] environment, where the server-side JIT compiler can perform expensive analysis and optimizations asynchronously in the background. Furthermore, such a setup could also maintain multiple optimized versions of the compiled code, allowing the most appropriate version to be dispatched to the client based on run-time behaviour and requirements.

Finally, it would be interesting to extend the proposed static+dynamic scheme to other programming languages and runtime environments. We believe that adapting this approach across different execution environments could enable new forms of aggressive optimizations, thereby opening additional avenues for further research.

References

[1] Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. Maximal Specification Synthesis. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 789–801, New York, NY, USA, 2016. Association for Computing Machinery.

[2] Karim Ali. *The Separate Compilation Assumption*. PhD thesis, University of Waterloo, Waterloo, Ontario, Canada, 2014.

[3] Karim Ali and Ondřej Lhoták. Averroes: Whole-Program Analysis without the Whole Program. In Giuseppe Castagna, editor, *ECOOP 2013 – Object-Oriented Programming*, pages 378–400, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[4] Nicholas Allen, Padmanabhan Krishnan, and Bernhard Scholz. Combining Type-Analysis with Points-to Analysis for Analyzing Java Library Source-Code. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2015, page 13–18, New York, NY, USA, 2015. Association for Computing Machinery.

[5] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[6] Aditya Anand, Solai Adithya, Swapnil Rustagi, Priyam Seth, Vijay Sundaresan, Daryl Maier, V. Krishna Nandivada, and Manas Thakur. Optimistic Stack Allocation and Dynamic Heapification for Managed Runtimes. 8(PLDI), 2024.

[7] Aditya Anand, Vijay Sundaresan, Daryl Maier, and Manas Thakur. CoSSJIT: Combining Static Analysis and Speculation in JIT Compilers (Supplementary Material). *Proc. ACM Program. Lang.*, 9(OOPSLA2), June 2025.

[8] Aditya Anand and Manas Thakur. Principles of Staged Static+Dynamic Partial Analysis. In Gagandeep Singh and Caterina Urban, editors, *Static Analysis*, pages 44–73, Cham, 2022. Springer Nature Switzerland.

[9] Aditya Anand and Manas Thakur. Partial Program Analysis for Staged Compilation Systems. *Formal Methods in System Design*, 65(1):195–230, April 2025.

[10] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. RPython: A Step towards Reconciling Dynamically and Statically Typed OO Languages. In *Proceedings of the 2007 Symposium on Dynamic Languages*, DLS '07, page 53–64, New York, NY, USA, 2007. Association for Computing Machinery.

[11] Osbert Bastani, Rahul Sharma, Lazaro Clapp, Saswat Anand, and Alex Aiken. Eventually Sound Points-To Analysis with Specifications. In Alastair F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, volume 134 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:28, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[12] Osbert Bastani, Rahul Sharma, Lazaro Clapp, Saswat Anand, and Alex Aiken. Eventually Sound Points-To Analysis with Specifications. In Alastair F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, volume 134 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:28, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[13] Stephen M Blackburn, Zixian Cai, Rui Chen, Xi Yang, John Zhang, and John Zigman. Rethinking Java Performance Analysis. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2025, Rotterdam, Netherlands, 30 March 2025 - 3 April 2025*. ACM, 2025.

[14] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton,

Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM.

[15] Bruno Blanchet. Escape analysis for JavaTM: Theory and practice. *ACM Trans. Program. Lang. Syst.*, 25(6):713–775, November 2003.

[16] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 241–250, New York, NY, USA, 2011. ACM.

[17] Rodrigo Bruno, Vojin Jovanovic, Christian Wimmer, and Gustavo Alonso. Compiler-Assisted Object Inlining with Value Fields. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 128–141, New York, NY, USA, 2021. Association for Computing Machinery.

[18] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape Analysis for Java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, pages 1–19, New York, NY, USA, 1999. ACM.

[19] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and Precise Datarace Detection for Multithreaded Object-oriented Programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 258–269, New York, NY, USA, 2002. ACM.

[20] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged Information Flow for Javascript. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, page 50–62, New York, NY, USA, 2009. Association for Computing Machinery.

[21] Kevin Cleereman, Michelle Cheatham, and Krishnaprasad Thirunarayan. Runtime Support of Speculative Optimization for Offline Escape Analysis. In *Proceedings of the International Conference on Software Engineering Research and Practice*, pages 484–489, 2007.

[22] Erik Corry. Optimistic Stack Allocation for Java-like Languages. In *Proceedings of the 5th International Symposium on Memory Management*, ISMM '06, page 162–173, New York, NY, USA, 2006. Association for Computing Machinery.

[23] Patrick Cousot and Radhia Cousot. Modular Static Program Analysis. In R. Nigel Horspool, editor, *Compiler Construction*, pages 159–179, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[24] Barthélémy Dagenais and Laurie Hendren. Enabling Static Analysis for Partial Java Programs. *SIGPLAN Not.*, 43(10):313–328, October 2008.

[25] Charles Daly, Jane Horgan, James Power, and John Waldron. Platform Independent Dynamic Java Virtual Machine Analysis: The Java Grande Forum Benchmark Suite. In *Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande*, JGI '01, pages 106–115, New York, NY, USA, 2001. ACM.

[26] Jeffrey Dean, Craig Chambers, and David Grove. Selective Specialization for Object-Oriented Languages. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, page 93–102, New York, NY, USA, 1995. Association for Computing Machinery.

[27] Gilles Duboscq. *Combining Speculative Optimizations with Flexible Scheduling of Side-effects*. PhD thesis, Linz, Austria, 2016.

[28] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '13, page 1–10, New York, NY, USA, 2013. Association for Computing Machinery.

[29] Eclipse Foundation. Eclipse OMR, 2023.

[30] Eclipse Foundation. Eclipse OpenJ9, 2023.

[31] Mafalda Ferreira, Miguel Monteiro, Tiago Brito, Miguel E. Coimbra, Nuno Santos, Limin Jia, and José Fragoso Santos. Efficient Static Vulnerability Analysis for JavaScript with Multiversion Dependency Graphs. *Proc. ACM Program. Lang.*, 8(PLDI), June 2024.

[32] Yoshihiko Futamura. Partial Evaluation of Computation Process—AnApproach to a Compiler-Compiler. *Higher Order Symbol. Comput.*, 12(4):381–391, December 1999.

[33] Robert Glück. Is there a fourth Futamura projection? In *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '09, page 51–60, New York, NY, USA, 2009. Association for Computing Machinery.

[34] Shashin Halalingaiah, Vijay Sundaresan, Daryl Maier, and V. Krishna Nandivada. The ART of Sharing Points-to Analysis: Reusing Points-to Analysis Results Safely and Efficiently. *Proc. ACM Program. Lang.*, 8(OOPSLA2), October 2024.

[35] Nevin Heintze and Olivier Tardieu. Demand-Driven Pointer Analysis. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, page 24–34, New York, NY, USA, 2001. Association for Computing Machinery.

[36] Urs Hölzle, Craig Chambers, and David Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, page 32–43, New York, NY, USA, 1992. Association for Computing Machinery.

[37] Neil D. Jones. An Introduction to Partial Evaluation. *ACM Comput. Surv.*, 28(3):480–503, September 1996.

[38] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[39] Alexey Khrabrov, Marius Pirvu, Vijay Sundaresan, and Eyal de Lara. JITServer: Disaggregated Caching JIT Compiler for the JVM in the Cloud. In *2022 USENIX*

Annual Technical Conference (USENIX ATC 22), pages 869–884, Carlsbad, CA, July 2022. USENIX Association.

[40] Thomas Kotzmann and Hanspeter Mössenböck. Escape Analysis in the Context of Dynamic Compilation and Deoptimization. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, page 111–120, New York, NY, USA, 2005. Association for Computing Machinery.

[41] Thomas Kotzmann and Hanspeter Mössenböck. Run-Time Support for Optimizations Based on Escape Analysis. In *International Symposium on Code Generation and Optimization (CGO'07)*, pages 49–60, 2007.

[42] Florian Latifi. Practical Second Futamura Projection: Partial Evaluation for High-Performance Language Interpreters. In *Proceedings Companion of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, SPLASH Companion 2019, page 29–31, New York, NY, USA, 2019. Association for Computing Machinery.

[43] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In Defense of Soundiness: A Manifesto. *Commun. ACM*, 58(2):44–46, January 2015.

[44] Stefan Marr and Stéphane Ducasse. Tracing vs. Partial Evaluation: Comparing Meta-Compilation Approaches for Self-Optimizing Interpreters. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, page 821–839, New York, NY, USA, 2015. Association for Computing Machinery.

[45] Meetesh Kalpesh Mehta, Sebastián Krynski, Hugo Musso Gualandi, Manas Thakur, and Jan Vitek. Reusing Just-in-Time Compiled Code. *Proc. ACM Program. Lang.*, 7(OOPSLA2), October 2023.

[46] Leandro T. C. Melo, Rodrigo G. Ribeiro, Marcus R. de Araújo, and Fernando Magno Quintão Pereira. Inference of Static Semantics for Incomplete C Programs. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.

[47] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized Object Sensitivity for Points-to Analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, January 2005.

[48] Alon Mishne, Sharon Shoham, and Eran Yahav. Typestate-Based Semantic Code Search over Partial Programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, page 997–1016, New York, NY, USA, 2012. Association for Computing Machinery.

[49] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[50] Erick Ochoa, Cijie Xia, Karim Ali, Andrew Craik, and José Nelson Amaral. U Can't Inline This! In *Proceedings of the 31st Annual International Conference on Computer Science and Software Engineering*, CASCON '21, page 173–182, USA, 2021. IBM Corp.

[51] OpenJDK Graal. GraalVM, 2023.

[52] Michael Paleczny, Christopher Vick, and Cliff Click. The Java Hotspot Server Compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, page 1, USA, 2001. USENIX Association.

[53] Michael Paleczny, Christopher Vick, and Cliff Click. The Java HotspotTM Server Compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, page 1, USA, 2001. USENIX Association.

[54] Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. Type-Directed Completion of Partial Expressions. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, page 275–286, New York, NY, USA, 2012. Association for Computing Machinery.

[55] Saverio Perugini and Brandon Williams. Revisiting the Futamura Projections: A Diagrammatic Approach. *Theoretical and Applied Informatics*, 28:15–32, 12 2017.

[56] Girish Maskeri Rama, Raghavan Komondoor, and Himanshu Sharma. Refinement in Object-sensitivity Points-to Analysis via Slicing. *Proc. ACM Program. Lang.*, 2(OOPSLA):142:1–142:27, October 2018.

[57] Rishi Sharma, Shreyansh Kulshreshtha, and Manas Thakur. ZS3: Marrying Static Analyzers and Constraint Solvers to Parallelize Loops in Managed Runtimes. In *Proceedings of the 32nd Annual International Conference on Computer Science and Software Engineering*, CASCON '22, page 213–220, USA, 2022. IBM Corp.

[58] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick Your Contexts Well: Understanding Object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 17–30, New York, NY, USA, 2011. ACM.

[59] SPEC. SPECjvm 2008, 2008.

[60] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. Partial escape analysis and scalar replacement for java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, page 165–174, New York, NY, USA, 2018. Association for Computing Machinery.

[61] Giancarlo Succi and Raymond Wong. The Application of JavaCC to Develop a C/C++ Preprocessor. *ACM SIGAPP Applied Computing Review*, 7:11–18, 09 1999.

[62] Nikhil T R, Dheeraj Yadav, Aditya Anand, and Manas Thakur. Stava. `https://github.com/CompL-Research/Stava-Speculative`, 2025.

[63] Hao Tang, Xiaoyin Wang, Lingming Zhang, Bing Xie, Lu Zhang, and Hong Mei. Summary-Based Context-Sensitive Data-Dependence Analysis in Presence of Callbacks. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, New York, NY, USA, 2015. Association for Computing Machinery.

[64] Manas Thakur and V. Krishna Nandivada. PYE: A Framework for Precise-Yet-Efficient Just-In-Time Analyses for Java Programs. *ACM Trans. Program. Lang. Syst.*, 41(3):16:1–16:37, July 2019.

[65] Mads Tofte and Lars Birkedal. A Region Inference Algorithm. *ACM Trans. Program. Lang. Syst.*, 20(4):724–767, July 1998.

[66] Akshay Utture and Jens Palsberg. Fast and Precise Application Code Analysis using a Partial Library. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 934–945, New York, NY, USA, 2022. Association for Computing Machinery.

[67] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–23. IBM Press, 1999.

[68] Frédéric Vivien and Martin Rinard. Incrementalized Pointer and Escape Analysis. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, page 35–46, New York, NY, USA, 2001. Association for Computing Machinery.

[69] Frédéric Vivien and Martin Rinard. Incrementalized Pointer and Escape Analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '01, pages 35–46, New York, NY, USA, 2001. ACM.

[70] John Whaley and Martin Rinard. Compositional Pointer and Escape Analysis for Java Programs. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, pages 187–206, New York, NY, USA, 1999. ACM.

[71] Christian Wimmer. GraalVM Native Image: Large-Scale Static Analysis for Java (keynote). In *Proceedings of the 13th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, VMIL 2021, page 3, New York, NY, USA, 2021. Association for Computing Machinery.

[72] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. Initialize Once, Start Fast: Application Initialization at Build Time. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[73] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. Practical Partial Evaluation for High-Performance Dynamic Language Runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 662–676, New York, NY, USA, 2017. Association for Computing Machinery.

[74] Hao Zhong and Xiaoyin Wang. Boosting Complete-Code Tool for Partial Program. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, page 671–681. IEEE Press, 2017.