CoSSJIT: Combining Static Analysis and Speculation in JIT Compilers

ADITYA ANAND, Indian Institute of Technology Bombay, India
VIJAY SUNDARESAN, IBM Canada Lab, Canada
DARYL MAIER, IBM Canada Lab, Canada
MANAS THAKUR, Indian Institute of Technology Bombay, India

Just-in-time (JIT) compilers typically sacrifice the precision of program analysis for efficiency, but are capable of performing sophisticated speculative optimizations based on run-time profiles to generate code that is specialized to a given execution. On the contrary, ahead-of-time static compilers can often afford precise flow-sensitive interprocedural analysis, but produce conservative results in scenarios where higher precision could be derived from run-time specialization. In this paper, we propose the 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 Java Virtual Machine (JVM).

Our approach of combining static analysis with JIT speculation – named `CoSSJIT` – involves three key contributions. First, we identify the scenarios where a static analysis would make conservative assumptions but a JIT could deliver precision based on run-time speculation. Second, we present the notion of "speculative conditions" and plug them into a static interprocedural dataflow analyzer (whose aim is to identify heap objects that can be allocated on stack), to generate partial results that can be specialized at run-time. Finally, we extend a production JIT compiler to read and enrich static-analysis results with the resolved values of speculative conditions, leading to a practical approach that efficiently combines the best of both worlds. Cherries on the cake: Using `CoSSJIT`, we obtain 5.7× improvement in stack allocation (translating to performance), while building on a system that ensures functional correctness during JIT compilation.

CCS Concepts: • **Theory of computation** → **Program analysis**; • **Software and its engineering** → **Compilers**; **Just-in-time compilers**; *Dynamic analysis*; Object oriented languages.

Additional Key Words and Phrases: Escape analysis, Stack allocation, Speculative optimization

ACM Reference Format:
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. https://doi.org/10.1145/3763149

1 Introduction

Modern managed runtimes have multiple tiers of translation comprising interpreters as well as baseline and optimizing just-in-time (JIT) compilers. Though 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.

Authors' Contact Information: Aditya Anand, Indian Institute of Technology Bombay, Mumbai, India, adityaanand@cse.iitb.ac.in; Vijay Sundaresan, IBM Canada Lab, Markham, Canada, vijaysun@ca.ibm.com; Daryl Maier, IBM Canada Lab, Markham, Canada, maier@ca.ibm.com; Manas Thakur, Indian Institute of Technology Bombay, Mumbai, India, manas@cse.iitb.ac.in.

ACM 2475-1421/2025/10-ART371
https://doi.org/10.1145/3763149

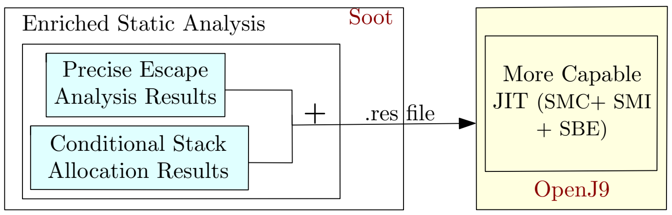

Enriched Static Analysis
Soot
Precise Escape Analysis Results
Conditional Stack Allocation Results
+
.res file
More Capable JIT (SMC+ SMI + SBE)
OpenJ9

Fig. 1. 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 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 [12, 27], to offline analysis of JIT-compiled code [20, 32], to feedback-based approaches that improve results over multiple executions [5]. 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 [32], whereas others employ run-time techniques to detect issues and/or repair the state before they could alter the program's behavior [1, 16].

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, we observe that 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. In this paper, we present the first static+JIT strategy of this kind and implement it in a production setting, to perform an impactful optimization much more aggressively than the state-of-the-art.

The optimization we choose is *stack allocation* [8, 11, 30] – which exploits the property that an object whose lifetime is shorter than that of its allocating method can be allocated on the stack frame of that method (either as an aggregate or decomposed into its fields, called *scalar replacement*). Allocating an object on stack is known to reduce the access time of its fields, often improve cache locality, reduce the need of garbage collection, and consequently improve performance. Owing to these benefits, most OO JIT compilers (for languages such as Java) estimate the lifetimes of objects using *escape analysis* and implement a stack allocation and/or scalar replacement pass. Recent works have also proposed functionally correct usages of static escape analysis for stack allocation in various JITs, making it a perfect candidate for demonstrating our hypothesis stated above.

Our 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. We chose to build the first over an existing formulation of static escape analysis [26], which implements the classic flow-, field- and context-sensitive pointer and escape analysis proposed by Whaley and Rinard [29, 30]. This analysis is known to be more precise than the JIT analyses implemented in popular Java Virtual Machines (JVMs) such as HotSpot, OpenJ9 and GraalVM.

In order to enrich static analysis with the possibility of run-time speculation, our first contribution is the introduction of *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. We add these conditions as special values in the lattice of the underlying 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, our 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. We describe each of these conditions along with examples in Section 4, and have implemented them in the Soot framework [28].

Our second contribution 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, we also have a tunable "speculation threshold" that determines the profile percentage we consider good enough for driving speculative stack allocation. In order to detect and repair incorrect speculative allocations, we utilize an existing notion called *heapification* that uses run-time checks to identify affected objects, and to copy them onto the heap while updating their references. We have implemented the components associated with this contribution in the Testarossa JIT compiler of the Eclipse OpenJ9 VM [14].

We evaluate our approach by comparing it with the baseline escape analysis of the Eclipse OpenJ9 VM over benchmarks from the DaCapo 23.11 [6] and 9.12 [7] suites, as well as from SPECjvm2008 [24]. We find that our approach significantly increases the amount of stack allocation performed by the JVM (5.7×, overall), which translates to decent performance improvements (0.34-20% and on average 6.7% across benchmarks with improved stack allocation, and without any noticeable degradation for the rest). We also study the effects of different kinds of speculative stack allocations, and find that though they are essentially program-dependent, speculative method-inline conditions are the most impactful over different benchmarks. Finally, we measure the overheads of our approach in terms of additional time spent during JIT compilation, and find them to be minimal (0.75%) compared to the huge benefits obtained therein.

Contributions:

- We introduce `CoSSJIT`, an idea that enriches static analysis to account for the possibility of run-time speculations into its results.
- We instantiate our idea for speculatively allocating objects on stack in three scenarios: (i) An object can be allocated on its allocating method's stack 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 an existing idea that heapifies them before being referenced.
- We implement our proposed integration of speculation-enriched static analysis results and speculative stack allocation across the tiered infrastructure of a production JVM.
- We provide 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.

The rest of the paper is organized as follows. Section 2 presents an overview of various concepts and frameworks used in our paper. Section 3 illustrates the interesting challenges that our proposed approach handles, using a motivating example. Section 4 describes the static component of our approach in terms of enriching a static escape analysis to account for possible speculations during JIT compilation. Section 5 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. Section 6 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. Finally, Section 7 presents important related work and Section 8 concludes the paper.

2 Background

2.1 Escape Analysis and Stack Allocation

Escape analysis [11] 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 [18, 25]), 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* [8, 11, 30]). 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 [30] 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.

Another popular optimization enabled by escape analysis (though in fewer scenarios than stack allocation) is *scalar replacement* [18]. Here, an eligible object 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 [11] of the methods being compiled at higher optimization levels of the C2 (Server) JIT compiler [23]. Similarly, the JIT compiler of GraalVM [22] performs scalar replacement aggressively, and *materializes* [25] the fields on the heap in branches where the corresponding object is found to escape.

2.2 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 dataflow 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 if-else conditionals or switch cases, and the receiver types or the methods bound at polymorphic call sites [13].

Based on run-time profiles, JIT compilers can make sophisticated 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. If an assumption is found to be violated during execution, the runtime *deoptimizes* [17], which translates to falling back either to a safer version of compiled code or to interpreting original input code. Typical Java runtimes perform speculative optimizations such as method inlining and 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 (for 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).

2.3 Testarossa JIT and Eclipse OpenJ9

Eclipse OpenJ9 is a popular Java runtime that closely resembles the Oracle HotSpot JVM on traditional workloads and even outperforms it for high-performance applications. On the compilation front, it consists of a sophisticated JIT compiler called Testarossa [14] (TR-JIT), which converts Java Bytecode to a Tree intermediate language (Tree IL) and performs optimizations tiered across multiple optimization levels (cold, warm, hot, very hot, and scorching). At higher optimization levels, it 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 that sometimes can “peek” inside called methods to get an interprocedural flavor. The TR-JIT compiler of OpenJ9 also adapts the idea of partial escape analysis [25] 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 paper can be expected to provide similar improvements therein.

A recent work [1] proposed the idea of run-time checks and dynamic heapification to correctly utilize interprocedural flow-sensitive static-analysis results for enabling stack allocation during JIT compilation in the OpenJ9 VM[1]. Its evaluation showed that it was not only feasible to use static-analysis results for improving stack allocation significantly, but that the increased allocation led to reduced garbage collection and to noticeable improvements in performance. In this paper, we propose a different approach that makes a complementary hypothesis: We claim that there are interesting scenarios where a JIT analysis would be better than a purely static analysis, and that it is possible to design an approach that takes advantage of the best of both the worlds. To demonstrate that both the ideas – using static analysis to improve JIT optimizations, and using JIT speculation to improve static-analysis results – can be applied together, we (i) choose OpenJ9 itself for implementing our approach; and (ii) compute the impact of our work over and above what can be accomplished by the prior art.

[1]Unlike materialization [25] that reconstructs and allocates scalar-replaced objects on heap (in GraalVM), heapification [1] moves stack-allocated objects to heap (in OpenJ9).

```
1  class A {
2     A f;
3     static A global;
4     void foo(A p1) {
5        A x = new A(); // $O_5$
6        A y = new A(); // $O_6$
7        x.f = new A(); // $O_7$
8        A z;
9        if (p1 instanceof A) {
10          z = new B(); // $O_{10}$
11       } else {
12          z = new C(); // $O_{12}$
13          global = y; // $O_6$ 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(); // $O_{22}$
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 */
```

Fig. 2. Motivating example to illustrate various considerations made by CoSSJIT.

3 Motivating Example

Consider the Java code snippet shown in Figure 2, 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 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 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 paper 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 2, 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. A recent work [1] added this notion for guaranteeing functional correctness while utilizing 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 5.

4 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 (link hidden for anonymity) for Java programs, which implements the classic flow-, field- and context-sensitive pointer and escape analysis proposed by Whaley and Rinard [30]. 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 said to *escape*, 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 4.1 for polymorphic call sites, Section 4.2 for conditional branching, and Section 4.3 for method inlining.

4.1 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 2. 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

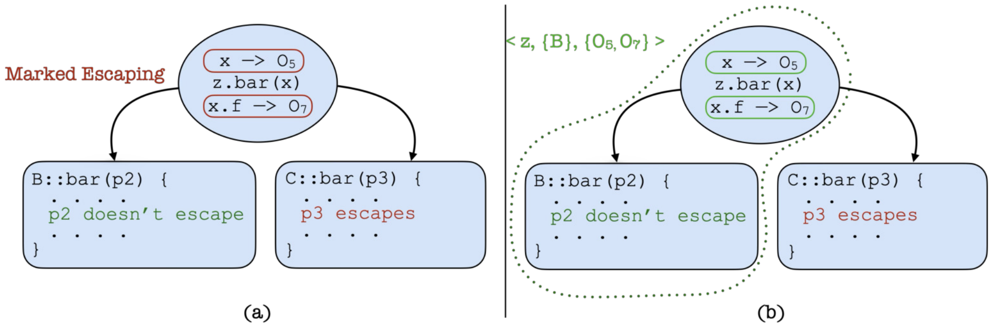


Fig. 3. Handling a polymorphic callsite: (a) Existing static analysis; (b) CoSSJIT's static analysis. Arrows indicate (may) points-to relationships.

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 $\langle z, \ \{B\}, \ \{O_5, O_7\}\rangle$, 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 3 illustrates this case for the given code snippet, contrasting the existing (conventional) approach with ours.

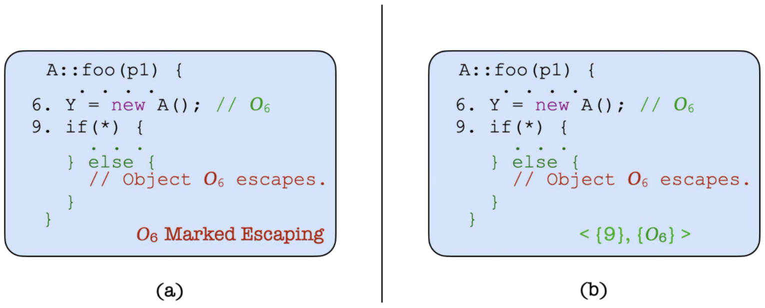
4.2 Handling Conditional Branching

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 2. 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 $\langle\{9\}, \ \{O_6\}\rangle$, where 9 represents the bytecode 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 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



```
A::foo(p1) {
   . . . .
6. Y = new A(); // O6
9. if(*) {
     . . .
   } else {
     // Object O6 escapes.
   }
 }
```

```
A::foo(p1) {
   . . . .
6. Y = new A(); // O6
9. if(*) {
     . . .
   } else {
     // Object O6 escapes.
   }
 }
```

Fig. 4. Handling objects in branches: (a) Existing static analysis; (b) CoSSJIT's static analysis.

the manner described in Section 4.3, we implement the addition of speculative branch-execution conditions as an additional pass over the existing escape analyzer.

4.3 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 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.

Consider the code snippet shown in Figure 5, 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

```
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 { ... }
```

Fig. 5. Extended example from Figure 2 to illustrate benefits of conditional inlining by CoSSJIT.

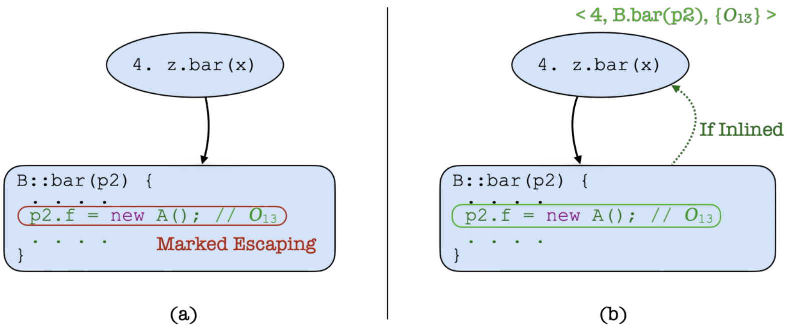


Fig. 6. A callsite with possible inlining: (a) Existing static analysis; (b) CoSSJIT's static analysis.

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 6 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).

4.4 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 such combinations may generate is a well-known problem with 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.

5 Implementation in OpenJ9

Having described the generation of conditional static-analysis results in Section 5, 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 5.1), followed by explaining how we utilize them to resolve the three kinds of speculative conditions, to consequently allocate more objects on stack (Section 5.2). 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.

5.1 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, etc (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).

5.2 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), 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.

5.2.1 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. 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 2, 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.

5.2.2 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, 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

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

```
Data:  $SA_m$: Map from objects to speculative conditions generated by 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.
1  $Candidates_m$= Set of all objects identified by JIT in method m.
2  foreach $O \in Candidates_m$ do
3      if $O$ is unconditionally non-escaping then
4          Mark the object $O$ for stack allocation.
5      else
6          markedNonEscaping = false.
7          conditionalAllocation = true.
8          if $O \in SAm$ then
               // 1. Check if $O$ is likely to escape at polymorphic callsite.
9              foreach Callsite $c \in SA_m[O]$ do
10                 if $CH_m[c] \not\subseteq SA_m[O][c]$ then
11                     if $\sum_{\forall t \in SA_m[O][c]} (CP_m(t) \leq ST)$ then
12                         conditionalAllocation = false. // break
13             if conditionalAllocation then
14                 Mark the object $O$ for stack allocation.
15                 markedNonEscaping = true.
               // 2. Check if $O$ is likely to escape in branches.
16             if $\neg$conditionalAllocation then
17                 conditionalAllocation = true.
18                 foreach Branchresult $br \in SA_m[O]$ do
19                     if $\sum_{\forall b \in SA_m[O][br]} (BP_m(b) \leq ST)$ then
20                         conditionalAllocation = false. // break
21             if conditionalAllocation then
22                 Mark the object $O$ for stack allocation.
```

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

5.2.3 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

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

```
Data: $SA_m$: Map from objects to speculative conditions generated by 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.
```

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 4.3), 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 2, 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 object O_{22} for stack allocation.

5.3 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.

6 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?

6.1 Experimental Setup and Benchmarks

Our static analysis is implemented on top of the Soot framework version 3.1 [28], with TamiFlex version 2.0.3 [9] 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 [6] as well as from SPECjvm2008 [24]. 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 [7], and hence we added them as corresponding replacements. The first column of Table 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 6.2 answers RQ1 and RQ2, Section 6.3 answers RQ3, and Section 6.4 answers RQ4.

Table 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

6.2 Increase in Stack Allocation

6.2.1 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 5). 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 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 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 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

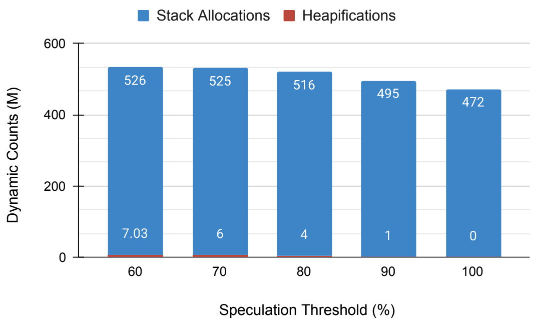


Fig. 7. Number of stack allocations and heapifications (in millions) at different values of ST (Speculation Threshold) for the benchmark h2.

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 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 6.3.

Value of speculation threshold. Recall from Section 5 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 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 seconds, we decided to play with the value of ST for benchmarks in which we observed more than a million heapifications; Figure 7 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.

6.2.2 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,

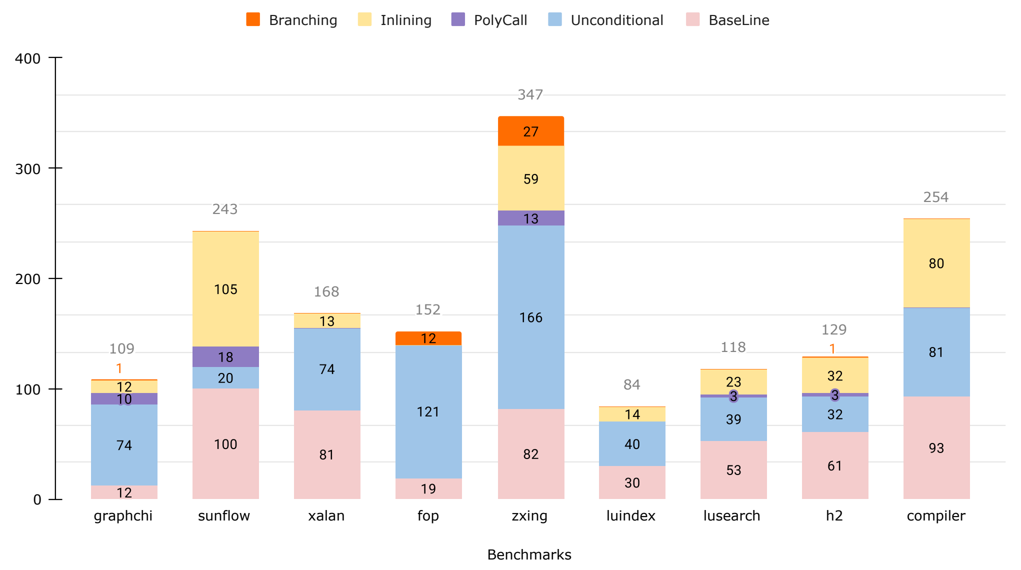


Fig. 8. Distribution of the contributions made by different speculative conditions, in terms of the number of objects marked for stack allocation during JIT compilation.

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 8 shows the static counts for each of these modes, totaling to the count reported under `CoSSJIT` in Table 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 8). 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 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.

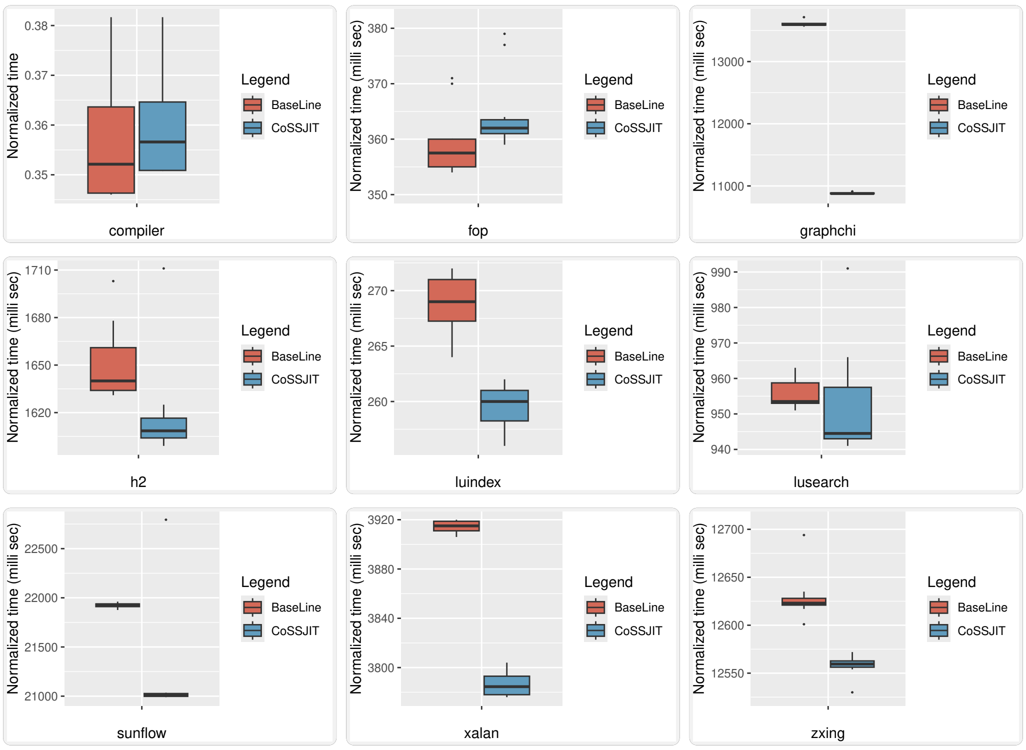


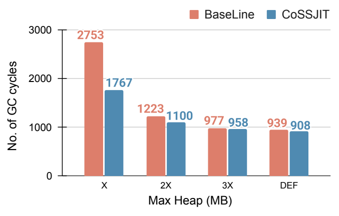
Fig. 9. Performance comparison between BaseLine and CoSSJIT; lower the better. For DaCapo benchmarks the y-axis is time (in milli sec) and for SPECjvm benchmarks it is the normalized reciprocal of ops/sec.

6.3 Impact of Stack Allocation

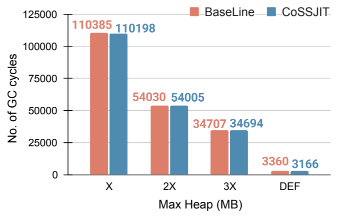
6.3.1 Impact on Performance. Given the importance of stack allocation in reducing object allocation and access times, as well as garbage collection, we measured the impact of additional stack allocation achieved by CoSSJIT, over the baseline OpenJ9 VM. While measuring performances in JITted systems can be non-trivial (variations due to non-determinism in order and number of compilations, method inlines, etc), 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 [21] for DaCapo 9.12-MRI and fixed at 100 for DaCapo 23.11-chopin based on approximations done by [1]) 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 9 uses box plots to report the performance[2] for the nine benchmarks with high stack allocation from Section 6.2 (the performance metric is normalized across the two benchmark suites). For space, the results for the remaining benchmarks are added to Appendix A [3].

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

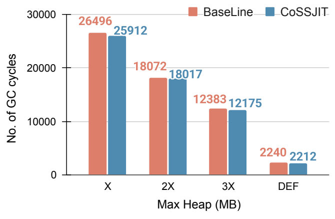
[2]For drawing box plots, we use multiple iterations at the approximated steady state within a given run, instead of picking a particular iteration from multiple runs. As we pin the programs to specific cores and run with a constant workload, we do not observe any noticeable difference in the range of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



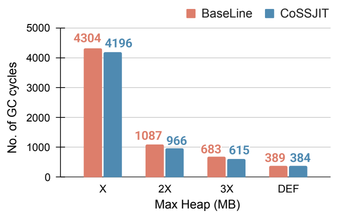
(a) compiler



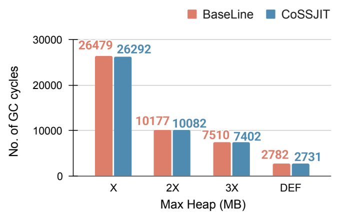
(b) fop



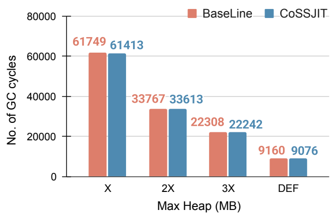
(c) graphchi



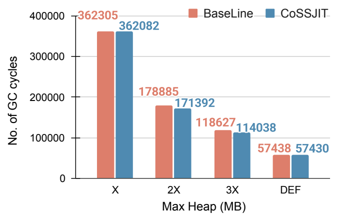
(d) h2



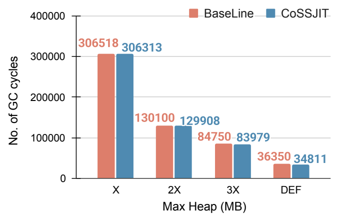
(e) luindex



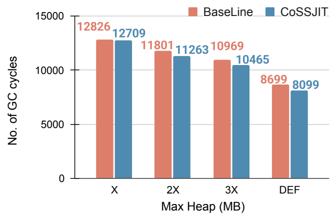
(f) lusearch



(g) sunflow



(h) xalan



(i) zxing

Fig. 10. Number of GC cycles with varied heap sizes (X, 2X, 3X and default heap memory); lower the better.

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 9. 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.

6.3.2 *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 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 10 for the results of this experiment.

Table 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 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 9. For completion, we have added the corresponding charts to Appendix B [3].

6.4 Additional Overheads

6.4.1 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 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 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 4.4 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.

6.4.2 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 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"). 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 a reduction in the 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)[3].

6.4.3 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 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 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 11 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 6.2. We can see that by default, both HotSpot and OpenJ9 mark a

[3]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.

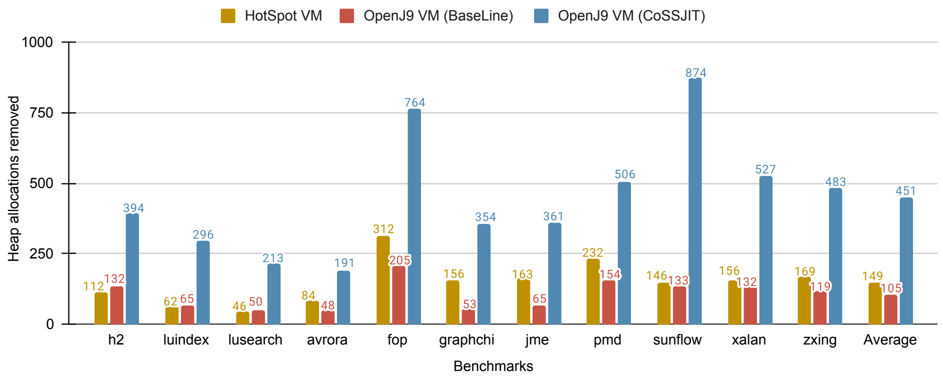


Fig. 11. Number of heap allocations removed in HotSpot VM, OpenJ9 (BaseLine) and OpenJ9 (CoSSJIT)

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. For completion, we also measured the performance of the two VMs, and found them to be very comparable (modulo the differences in their mode of operation such as the heuristics used for compiling a method, the optimizations at different compilation levels, and so on); see Appendix C [3] for details.

7 Related Work

In general, we are not aware of any practical attempts that combine the best of AOT static analysis and JIT speculation. However, challenges in imparting efficiency to precise JIT analyses are well known and a good chunk of recent research has focussed on mitigating them. In this section, we present related work on this topic, with respect to approaches that range from partial to feedback-driven static analysis, to incremental and offline analysis of JITted code.

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 [12] to statically generate *conditional dataflow values* that can be resolved during JIT compilation [4, 27]. 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 [31]. 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 though the latter approach allows offline analysis of code produced in JITted executions (e.g. [10]), 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 [20]. 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 [16]. Similarly, another work proposed timely detection and repair of incorrect stack allocations during execution [1]. In this paper, 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 implementation 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 [25] 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 [19]), a recent work proposed the notion of *eventual soundness*, which iteratively obtains sound results for Java pointer analysis [5]. Such feedback-driven ideas are gaining traction in the JavaScript community as well [15], 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.

8 Conclusion

In this paper, we 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, 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.

Acknowledgments

This research was partially supported by IBM through the CAS project 1156. Also, we would like to thank the anonymous reviewers for helping us improve the paper.

Data Availability Statement

The artifact for this paper, including source code, benchmarks and scripts, is available publicly [2].

References

[1] Aditya Anand, Solai Adithya, Swapnil Rustagi, Priyam Seth, Vijay Sundaresan, Daryl Maier, V. Krishna Nandivada, and Manas Thakur. 2024. Optimistic Stack Allocation and Dynamic Heapification for Managed Runtimes. 8, PLDI (2024). doi:10.1145/3656389

[2] Aditya Anand, Vijay Sundaresan, Daryl Maier, and Manas Thakur. 2025. *Artifact of CoSSJIT: Combining Static Analysis and Speculation in JIT Compilers.* https://doi.org/10.5281/zenodo.15762175

[3] Aditya Anand, Vijay Sundaresan, Daryl Maier, and Manas Thakur. 2025. CoSSJIT: Combining Static Analysis and Speculation in JIT Compilers (Supplementary Material). *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 371 (June 2025), 31 pages. doi:10.1145/3763149

[4] Aditya Anand and Manas Thakur. 2022. Principles of Staged Static+Dynamic Partial Analysis. In *Static Analysis*, Gagandeep Singh and Caterina Urban (Eds.). Springer Nature Switzerland, Cham, 44–73. doi:10.1007/978-3-031-22308-2_4

[5] Osbert Bastani, Rahul Sharma, Lazaro Clapp, Saswat Anand, and Alex Aiken. 2019. Eventually Sound Points-To Analysis with Specifications. In *33rd European Conference on Object-Oriented Programming (ECOOP 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 134)*, Alastair F. Donaldson (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 11:1–11:28. doi:10.4230/LIPIcs.ECOOP.2019.11

[6] Stephen M Blackburn, Zixian Cai, Rui Chen, Xi Yang, John Zhang, and John Zigman. 2025. 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. doi:10.1145/3669940.3707217

[7] 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. 2006. 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* (Portland, Oregon, USA) *(OOPSLA '06)*. ACM, New York, NY, USA, 169–190. doi:10.1145/1167473.1167488

[8] Bruno Blanchet. 2003. Escape analysis for JavaTM: Theory and practice. *ACM Trans. Program. Lang. Syst.* 25, 6 (Nov. 2003), 713–775. doi:10.1145/945885.945886

[9] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders. https://github.com/secure-software-engineering/tamiflex. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. ACM, New York, NY, USA, 241–250. doi:10.1145/1985793.1985827

[10] Rodrigo Bruno, Vojin Jovanovic, Christian Wimmer, and Gustavo Alonso. 2021. Compiler-Assisted Object Inlining with Value Fields. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 128–141. doi:10.1145/3453483.3454034

[11] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. 1999. Escape Analysis for Java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (Denver, Colorado, USA) *(OOPSLA '99)*. ACM, New York, NY, USA, 1–19. doi:10.1145/320384.320386

[12] Barthélémy Dagenais and Laurie Hendren. 2008. Enabling Static Analysis for Partial Java Programs. *SIGPLAN Not.* 43, 10 (Oct. 2008), 313–328. doi:10.1145/1449955.1449790

[13] Gilles Duboscq. 2016. *Combining Speculative Optimizations with Flexible Scheduling of Side-effects.* Ph. D. Dissertation. Linz, Austria. https://ssw.jku.at/Teaching/PhDTheses/Duboscq/index.html

[14] Eclipse Foundation. 2023. *Eclipse OpenJ9.* https://www.eclipse.org/openj9/

[15] Mafalda Ferreira, Miguel Monteiro, Tiago Brito, Miguel E. Coimbra, Nuno Santos, Limin Jia, and José Fragoso Santos. 2024. Efficient Static Vulnerability Analysis for JavaScript with Multiversion Dependency Graphs. *Proc. ACM Program. Lang.* 8, PLDI, Article 164 (June 2024), 25 pages. doi:10.1145/3656394

[16] Shashin Halalingaiah, Vijay Sundaresan, Daryl Maier, and V. Krishna Nandivada. 2024. The ART of Sharing Points-to Analysis: Reusing Points-to Analysis Results Safely and Efficiently. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 363 (Oct. 2024), 27 pages. doi:10.1145/3689803

[17] Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation* (San Francisco, California, USA) *(PLDI '92)*. Association for Computing Machinery, New York, NY, USA, 32–43. doi:10.1145/143095.143114

[18] Thomas Kotzmann and Hanspeter Mössenböck. 2007. Run-Time Support for Optimizations Based on Escape Analysis. In *International Symposium on Code Generation and Optimization (CGO'07)*. 49–60. doi:10.1109/CGO.2007.34

[19] 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. 2015. In Defense of Soundiness: A Manifesto. *Commun. ACM* 58, 2 (Jan. 2015), 44–46. doi:10.1145/2644805

[20] Meetesh Kalpesh Mehta, Sebastián Krynski, Hugo Musso Gualandi, Manas Thakur, and Jan Vitek. 2023. Reusing Just-in-Time Compiled Code. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 263 (Oct. 2023), 22 pages. doi:10.1145/3622839

[21] Erick Ochoa, Cijie Xia, Karim Ali, Andrew Craik, and José Nelson Amaral. 2021. U Can't Inline This!. In *Proceedings of the 31st Annual International Conference on Computer Science and Software Engineering* (Toronto, Canada) *(CASCON '21)*. IBM Corp., USA, 173–182. doi:10.5555/3507788.3507812

[22] OpenJDK Graal. 2023. GraalVM. https://www.graalvm.org

[23] Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java HotspotTM Server Compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1* (Monterey, California) *(JVM'01)*. USENIX Association, USA, 1. https://www.usenix.org/legacy/events/jvm01/full_papers/paleczny/paleczny.pdf

[24] SPEC. 2008. *SPECjvm 2008*. https://www.spec.org/jvm2008/

[25] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Partial Escape Analysis and Scalar Replacement for Java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Orlando, FL, USA) *(CGO '14)*. Association for Computing Machinery, New York, NY, USA, 165–174. doi:10.1145/2544137.2544157

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

[27] Manas Thakur and V. Krishna Nandivada. 2019. PYE: A Framework for Precise-Yet-Efficient Just-In-Time Analyses for Java Programs. *ACM Trans. Program. Lang. Syst.* 41, 3, Article 16 (July 2019), 37 pages. doi:10.1145/3337794

[28] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research* (Mississauga, Ontario, Canada) *(CASCON '99)*. IBM Press, 13–23. https://dl.acm.org/citation.cfm?id=781995.782008

[29] Frédéric Vivien and Martin Rinard. 2001. Incrementalized Pointer and Escape Analysis. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation* (Snowbird, Utah, USA) *(PLDI '01)*. Association for Computing Machinery, New York, NY, USA, 35–46. doi:10.1145/378795.378804

[30] John Whaley and Martin Rinard. 1999. Compositional Pointer and Escape Analysis for Java Programs. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (Denver, Colorado, USA) *(OOPSLA '99)*. ACM, New York, NY, USA, 187–206. doi:10.1145/320385.320400

[31] Christian Wimmer. 2021. 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* (Chicago, IL, USA) *(VMIL 2021)*. Association for Computing Machinery, New York, NY, USA, 3. doi:10.1145/3486606.3488075

[32] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. 2019. Initialize Once, Start Fast: Application Initialization at Build Time. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 184 (Oct. 2019), 29 pages. doi:10.1145/3360610

Received 2025-03-26; accepted 2025-08-12