Optimistic Stack Allocation and Dynamic Heapification for Managed Runtimes

ADITYA ANAND, Indian Institute of Technology Bombay, India
SOLAI ADITHYA, Indian Institute of Technology Mandi, India
SWAPNIL RUSTAGI, Indian Institute of Technology Mandi, India
PRIYAM SETH, Indian Institute of Technology Mandi, India
VIJAY SUNDARESAN, IBM Canada, Canada
DARYL MAIER, IBM Canada, Canada
V. KRISHNA NANDIVADA, Indian Institute of Technology Madras, India
MANAS THAKUR, Indian Institute of Technology Bombay, India

The runtimes of managed object-oriented languages such as Java allocate objects on the heap, and rely on automatic garbage collection (GC) techniques for freeing up unused objects. Most such runtimes also consist of just-in-time (JIT) compilers that optimize memory access and GC times by employing *escape analysis*: an object that does not escape (outlive) its allocating method can be allocated on (and freed up with) the stack frame of the corresponding method. However, in order to minimize the time spent in JIT compilation, the scope of such useful analyses is quite limited, thereby restricting their precision significantly. On the contrary, even though it is feasible to perform precise program analyses statically, it is not possible to use their results in a managed runtime without a closed-world assumption. In this paper, we propose a static+dynamic scheme that allows one to harness the results of a precise static escape analysis for allocating objects on stack, while taking care of both soundness and efficiency concerns in the runtime.

Our scheme comprises of three key ideas. First, using the results of a statically performed escape analysis, it performs *optimistic stack allocation* during JIT compilation. Second, it handles the challenges associated with features that may invalidate the optimism, using a novel idea of *dynamic heapification*. Third, it uses another novel notion of *stack ordering*, again supported by a static analysis, to reduce the overheads associated with the checks that determine the need for heapification. The static and the runtime components of our approach are implemented in the Soot optimization framework and in the tiered infrastructure of the Eclipse OpenJ9 VM, respectively. To evaluate the benefits, we compare our scheme with the existing escape analysis and find that it succeeds in allocating a much larger number of objects on the stack. Furthermore, the enhanced stack allocation leads to a significant reduction in the number of GC cycles and brings decent performance improvements, especially suited for constrained-memory environments.

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: Static analysis, Managed Runtimes, Escape Analysis

Authors' addresses: Aditya Anand, Indian Institute of Technology Bombay, India, adityaanand@cse.iitb.ac.in; Solai Adithya, Indian Institute of Technology Mandi, India, solaiadithya001@gmail.com; Swapnil Rustagi, Indian Institute of Technology Mandi, India, swapnilrustagi97792@gmail.com; Priyam Seth, Indian Institute of Technology Mandi, India, sethpriyam1@gmail.com; Vijay Sundaresan, IBM Canada, Canada, vijaysun@ca.ibm.com; Daryl Maier, IBM Canada, Canada, maier@ca.ibm.com; V. Krishna Nandivada, Indian Institute of Technology Madras, India, nvk@iitm.ac.in; Manas Thakur, Indian Institute of Technology Bombay, India, manas@cse.iitb.ac.in.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 2475-1421/2024/1-ART1
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
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. *Proc. ACM Program. Lang.* 8, PLDI, Article 1 (January 2024), 24 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The managed runtimes of languages such as Java allocate objects on heap and deallocate them using automatic garbage collection. This unburdens the programmer from making intricate allocation-deallocation decisions, and reduces the possibility of harmful memory bugs. On the other hand, memory allocated on stack gets freed up as soon as the activation record of the allocating method is popped out, thus eliminating the need for explicit garbage collection. The ability to allocate objects or their fields on the stack instead of on the heap has long intrigued the community, with researchers coming up with different applications of *escape analysis* to enable the same. For example, the C2 just-in-time (JIT) compiler of the HotSpot VM [17] uses escape analysis to decompose objects into scalar variables on the stack. Similarly, Graal [16] uses a partial-escape analysis [20] to enable scalar replacement in parts of a program when it cannot be performed throughout the program. Nonetheless, there are a large number of scenarios when Java objects cannot be scalarized and need to exist in entirety. The most common case is when an object is passed out as an argument to a call that is too big to be inlined. To overcome these limitations, many prior works, instead of trying to decompose objects, choose to allocate them in their original shape on the stack instead of on the heap; this optimization is called *stack allocation* [4, 6, 24]. However, stack allocation in a managed runtime relies on analyses performed during JIT compilation which, like any JIT analysis, affects the execution time of the program. Consequently, to target efficiency, prior works involving JIT compilers resort to performing imprecise escape analyses that usually enable very few objects to be allocated on the stack. As an example, we found that on average across several benchmark programs of interest, the JIT compiler of the Eclipse OpenJ9 VM [10] was able to stack allocate only 0.16% of the total number of objects.

A promising alternative approach could be to perform an aggressive escape analysis statically – which could potentially identify a much larger number of method-local objects – and use its results to perform stack allocation during JIT compilation. This way, one could offload the task of performing a precise analysis to a stage prior to the VM invocation, and obtain a list of objects to be stack allocated without performing expensive analysis during JIT compilation. However, in presence of dynamic features supported by the language and the runtime, as well as differing libraries between the static analysis and the runtime, such a static+dynamic strategy may go wrong (and lead to erroneous stack allocation). As an instance, it is quite popular in the Java static-analysis community to resolve reflective calls using run-time logs [5]; if the call graph changes during execution from what was assumed statically, an object may get passed to a previously unknown method potentially making it escape. Similarly, many managed runtimes provide features like dynamic classloading and hot-code replacement that allow new code to show up during program execution, thereby again leading to a potentially wrong stack allocation if the object escapes therein. In this paper, we propose a sound and efficient approach of using static-analysis results to perform stack allocation in a Java VM, which is not marred by any dynamism that the language or the runtime may impose during program execution.

Our approach is based on three novel ideas: *optimistic stack allocation*, *dynamic heapification*, and *stack ordering*, supported by corresponding precise static analyses. To begin with, the first idea is for the VM to optimistically allocate a list of objects, generated by a static escape analysis, on the stack frames of the methods it JIT-compiles. This lets us harness the results of a precise escape analysis

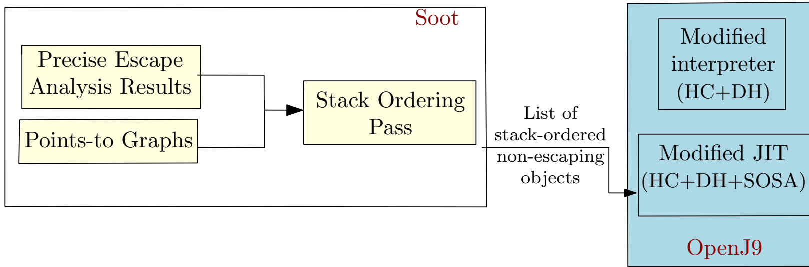


Fig. 1. Block diagram representing our static+dynamic approach for optimistic stack allocation. Abbreviations indicate modifications done in OpenJ9 towards: (i) HC: insertion of heapification checks; (ii) DH: dynamic heapification; (iii) SOSA: stack-ordered stack allocation.

for performing enhanced stack allocation. The second idea is to timely detect, using a run-time check and a “stack walk”, if any objects are allocated on the stack incorrectly by the optimistic scheme; and if so, then to move them onto the heap (along with correction of references). This dynamic heapification allows us to ensure that the optimism does not lead to unsoundness in terms of memory allocation and access. Finally, our third idea uses points-to graphs [24] to statically suggest an “order” for allocating non-escaping objects on the stack during execution. This helps us make a subtle improvement to our heapification checks, in a way that the VM can skip expensive stack walks a majority of times and instead be done with a simple condition-check to determine the requirement of heapification. Note that a scheme like this hinges on two primary requirements: (i) that any chances of an object being incorrectly on the stack be detected (and the repairs be done) before making any dereference – ensuring functional correctness; and (ii) that any run-time checks be as cheap as possible – to ensure efficiency. Our scheme satisfies these requirements across the tiered infrastructure of a production JVM and ensures that the optimism leads to as many benefits as possible (in terms of stack allocation) using the static analysis.

Figure 1 illustrates the key implementation components of our approach. Our static analyses are implemented over the Jimple IR of the Soot analysis framework [23]. In particular, we have (i) a context-, flow- and field-sensitive escape analysis that uses points-to graphs to generate a per-method list of non-escaping objects; and (ii) a stack ordering phase that traverses the points-to graphs to determine efficient stack orders for non-escaping objects. Our run-time components are implemented across the tiered infrastructure of the Eclipse OpenJ9 Java Virtual Machine. In particular, our modified OpenJ9 reads static analysis results and uses them to perform optimistic stack allocation. Our run-time heapification checks (HC) and dynamic heapification (DH) routines are inserted as part of the codegen phases of the JIT compiler as well as in the interpreter, in the form of new opcodes whose semantics enable the required operations. In addition, we have modified the stack allocation routine of the JIT compiler to allocate objects on the activation record of a method as per the (partial) stack order emitted by the static analyzer (SOSA).

We thoroughly evaluate various aspects of our approach over a series of benchmarks from the DaCapo [3] and the SPECjvm [19] suites. Firstly, we find that our approach significantly increases the number of objects allocated on stack (on average, 43%) as well as the number of bytes allocated on stack (on average, 54%). Second, we find that the overhead of the run-time checks is bypassed by the performance improvement brought by additional stack allocation. We experiment further by reducing the heap memory made available to the JVM, and observe that the improvement with our approach translates into fewer GC cycles (on average, 5.3% less), as well as a decent improvement in performance in low-memory environments (on average, 8.8%), for benchmarks with high stack

allocation. Overall, we conclude that our evaluation demonstrates the enhanced precision, coupled with high performance, that our proposed approach could impart to existing JIT pipelines.

Contributions:

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

The rest of the paper is organized as follows. Section 2 illustrates the interesting challenges that our proposed approach handles, with a motivating example. Section 3 describes the key run-time components of our approach: optimistic stack allocation, run-time checks for heapification, and recursive dynamic heapification using stack walks. Later, Section 4 shows how we determine and use stack orders to reduce the number of stack walks towards reducing the overhead of run-time checks. Section 5 states the correctness criterion of our approach, and discusses few design choices that we had to make while implementing it for a full language in an industry VM. Section 6 evaluates the improvements imparted by our approach in terms of stack allocation, performance in constrained memory environments, as well as the contribution of the key ideas in imparting efficiency. Finally, Section 7 presents important related work and Section 8 concludes the paper.

2 MOTIVATING EXAMPLE

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

Consider the Java code snippet shown in Figure 2. The class A has a field g of type B and the class C has a field f of type A. Denoting the abstract object(s) allocated at line l as O_l, we can see that the reference variable x in the method doo points to the object O_5, object O_5's field f points to the object O_7, and object O_7's field g points to the object O_9. The reference variable p too points to the object O_7 (due to the field load at line 8), which is later passed to the method zar of class C (assuming class D is not known during static analysis). As zar does not make the objects pointed-to by its parameters escape, we can allocate the objects O_7 and O_9 on doo's stack frame, as shown in the left-hand side of Figure 3. Prima facie, note that this information required an interprocedural analysis (to incorporate the effect of zar into doo).

Now consider a scenario in which during program execution in the JVM, the method doo is called by passing an object of a dynamically loaded class D (defined at line 20) as the second argument. Here, as the variable r now points to an object of type D, the call site at line 10 would invoke the overridden method zar in D. However, this zar stores the object pointed-to by its parameter p into the field f of its parameter q (at line 22), which in turn points to an object passed to its caller (via the parameter q in doo). As a consequence of this store statement, the lifetime of O_7 becomes

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

Fig. 2. Motivating example to explain various aspects of our run-time scheme.

longer than that of its allocating method doo, and thus O_7 cannot be kept on the stack frame of doo. Furthermore, the object O_9 is reachable from an escaping object O_7 and hence that too cannot be kept on the stack frame of doo. Letting these objects be on the stack is unsound. In order to ensure correctness, we need to first of all detect the erroneous stack allocation before anybody tries to access the corresponding object, and then ensure that the run-time memory layout is modified to avoid future incorrect accesses during the program's remaining execution. Also, as this needs to be performed on-the-fly during program execution, its efficiency is crucial. In the next three subsections, we describe how we perform these operations in context of our motivating example from Figure 2.

2.1 Detecting incorrect stack allocation

Recall that owing to dynamic classloading in the above discussed example, the object O_7 becomes non-stack-allocatable (i.e., escapes) at line 22 in D's zar. Thus, line 22 is the precise program point at which we could detect, just before storing O_7 into the field of a longer lifetime object (the one pointed-to by the parameter q), that we are going to make O_7 escape. In order to detect this occurrence, we insert a check (using a new opcode in the intermediate language of the JIT compiler) immediately before the write barrier associated with the store operation. This "heapification check" compares the lifetimes of the object being stored (say *rhs*) and the base object on the left-hand side of the assignment (say *lhs*), and thus decides whether we could have wrongly allocated *rhs* on the stack. Note that the lifetime-comparison check needs to be performed for any statement that may cause an object to escape, along with the more common store statement discussed above.

2.2 Moving objects and correcting references

Once we have detected that an object is wrongly allocated on the stack, we need to move it onto the heap (we call this operation *heapification*). Moving the object from the stack to the heap is as simple as creating a copy of the object (along with copying the values of its fields) and allocating the new object onto the heap by calling the VM's memory allocation routine. However, there are two more subtle issues to consider. First, the fields of the object being copied may be pointing to other non-primitive objects; for example, the object O_9 pointed-to by the field g of the object O_7. This means the heapification operation needs to be repeatedly performed until it has copied all the reachable objects onto the heap. Hence, once we detect an incorrectly stack allocated object, we invoke a "recursive heapification" routine that allocates all the reachable objects that are currently on the stack onto the heap. Second, there may be stale references to the object being heapified (on

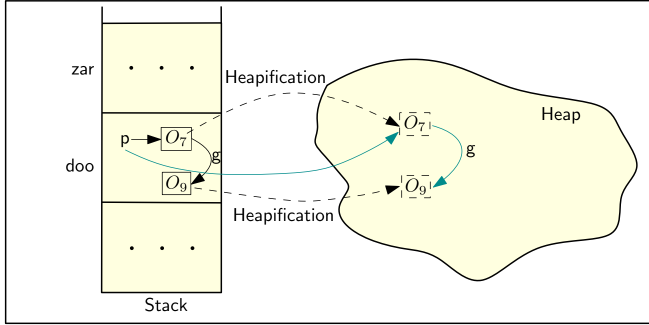


Fig. 3. A snapshot of the run-time stack and the heap after dynamic classloading in Figure 2. Dashed edges represent heapification and green edges represent updated points-to edges post heapification.

stack or in registers); for example, after this heapification is done, there remains a stale reference from the variable p in the method doo to the O_7 on the stack. Such references need to be corrected to point to the right object. We perform this "reference correction" by maintaining a map containing the old and the new addresses of the objects being recursively heapified, traversing the stack frames to find stale references, and then making them point to the right objects from the map. Figure 3 illustrates these operations for the code snippet under consideration.

2.3 Imparting efficiency through stack ordering

The heapification related steps are required as they affect the functional correctness of the optimistic stack allocation scheme. However, it is easy to see that the check and the heapification described above are costly. On the brighter side, it can be expected that the actual heapification operation would have to be invoked rarely. This leaves us with the cost of the heapification check, which is likely to be frequent (at each store to a non-primitive field, for example). Though we have not yet described the check routine completely, it is possible to see that the most expensive operation therein would be to traverse the stack frames until we find the *lhs* and the *rhs* objects, so that we can compare their addresses to in turn compare their lifetimes. We reduce the cost of these heapification checks by proposing another novel idea of ordering objects in stack frames, in a way that their addresses can be compared for the heapification checks, without doing stack walks frequently.

Essentially, while generating the list of stack allocatable objects in the static analysis (represented by the bytecode indices of the corresponding *new* bytecodes in class files), we order the objects according to a topological sort of the points-to graph constructed for performing the escape analysis. The VM is modified to allocate objects on stack in this presented order, and the heapification checks are modified to take advantage of this order, whenever possible. We leave the details for Section 4, but would point out to the reader here that these stack orders bring down the overhead of heapification checks noticeably. Consequently, compared to an implementation of dynamic heapification without stack orders, we see pronounced performance improvements for various benchmark programs considered in our evaluation.

Having motivated the reader towards the interesting challenges that our scheme solves, we now move on to a step-by-step description of the building blocks of our scheme in Sections 3 and 4.

3 OPTIMISTIC STACK ALLOCATION IN ECLIPSE OPENJ9

Eclipse OpenJ9 [10] is a popular open-source JVM implementation, which executes Java Bytecode using a multi-tier interpretation and compilation mechanism. The JIT compiler, Testarossa, uses multiple levels of compilation: noOpt, cold, warm, hot, veryHot and scorching, in order of increasing aggressiveness of the performed optimizations. The hot+ levels of the JIT compiler perform an escape analysis on the code being compiled, whereby the depth of peeking (inside the called methods) varies as the compilation level goes higher. OpenJ9 also implements a pass to allocate eligible objects on stack, based on the results of this escape analysis, as well as a few other constraints. In particular, the VM requires a fixed stack-frame size and hence precludes the stack allocation of variable-sized arrays and objects that escape a loop iteration. The VM also requires stack-allocated objects to be pre-zeroed at method entry and makes the garbage collector aware of these objects by maintaining their offsets in a metadata field with each method. Similar to other Java runtimes, OpenJ9 is also conservative in performing stack allocation (as well as other analyses and optimizations) when dynamic features such as hot-code replacement are allowed by the VM. As we show in Section 6, the imprecision introduced by time constraints during JIT compilation (escape analysis is only partially interprocedural, depending on the peek-depth), as well as the possibility of dynamic features, results in the VM being able to allocate a very small percentage of objects on the stack during program execution. In this section, we propose an approach that uses the results of a statically performed escape analysis to aggressively stack allocate method-local objects in OpenJ9, while being robust to the dynamic features allowed by the runtime.

The next two subsections discuss two major aspects of our scheme. Section 3.1 discusses how the static-analysis results are used in the runtime to optimistically allocate non-escaping objects on the stack. Section 3.2 discusses the dynamic heapification routine, which essentially is the repair that is performed when a stack-allocated object escapes. We discuss stack ordering, which is an optimization to reduce the overhead of the heapification routine, in Section 4.

3.1 Stack allocation using static analysis results

Our approach starts with a statically performed context-, flow- and field-sensitive escape analysis of Java programs in the Soot framework [23]. This analysis uses points-to graphs [24] to build method summaries in terms of the escape status of each abstract object allocated in the program. The summaries are built bottom-up over the call graph (which is generated using the context-insensitive SPARK [13] algorithm implemented in Soot).

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

In an execution enabled with optimistic stack allocation (determined using a command-line argument), we read the supplied .res file when the runtime is being initialized and store the results (say in a map called staticAnalysisNonEscapingMap). This ensures that we incur file-reading overheads only once, in the beginning.

In the existing VM implementation, whenever the JIT compiler picks a hot method for compilation, it traverses the Tree IL corresponding to that method, performs a simple (imprecise) escape analysis over the object nodes therein, and generates a list of candidates deemed suitable for stack allocation.

We top up this list with the objects whose BCIs are present in the list corresponding to the method being compiled in staticAnalysisNonEscapingMap so that those additional objects also get marked for stack allocation, without actually performing a complex analysis during JIT compilation. Later, the stack allocation routine allocates the objects present in this candidates list on the stack (instead of on the heap) when the stack frame of the corresponding method is created during execution.

Example. Consider the code snippet in Figure 2. The local objects O_5 and O_6 in the method doo are passed as arguments to the method bar. The existing escape analysis, unless it is able to peek inside bar, would not be able to stack allocate any of the objects O_5 or O_6 on the stack frame of doo. However, our static-analysis guided optimistic scheme would be able to determine that none of the objects passed to bar escape therein (the store statement inside bar involves objects allocated in the same method, i.e. doo – identifiable with one level of context sensitivity), and consequently mark both O_5 and O_6 in doo for stack allocation. Similarly, our scheme would be able to mark the objects O_7 and O_9 for stack allocation in doo.

3.2 Dynamic heapification

As we shall see in Section 6, our static-analysis based scheme marks a significantly high number of objects for stack allocation during program execution. However, in a language runtime that allows interesting dynamic features such as dynamic classloading (DCL) and hot-code replacement (HCR), which essentially allow arbitrary code changes, an object that was stack allocated may escape during execution. We cannot simply leave such objects allocated on the stack as is, and need a mechanism to repair the memory such that the program behaviour is preserved. The most unique feature of our approach is this timely detection and fallback mechanism for repairing incorrect stack allocation; we describe this next. Also, as a consequence of the presence of a dynamic repair mechanism, we call our static-analysis guided stack allocation *optimistic*.

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

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

(1) Returns of references (bytecode `areturn`).
(2) Reference stores to instance fields, static fields and object arrays (bytecodes `putfield`, `putstatic` and `aastore`, respectively).
(3) Throwing of exceptions (bytecode `athrow`).
(4) Calls to native methods that may lead to reference stores in `sun.misc.Unsafe` (`putObject`, `putObjectOrdered`, `putObjectVolatile` and `compareAndSwapObject`).
(5) JNI APIs used to perform stores in called C/C++ code that may manipulate Java objects (`setObjectField`, `setObjectArrayElement` and `setStaticObjectField`).

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

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

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

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

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

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

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

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

[1]We use this insight in Section 4 to *provide* stack orders and improve the address-comparison checks.

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

Fig. 4. The heapification check performed at store statements ($a.f = b$). lhsReg and rhsReg store the addresses of the base object (pointed-to by a) and that of the object being stored (pointed-to by b), respectively.

3.2.3 The heapification-check algorithm. Figure 4 summarizes the above discussion on heapification checks for store statements (of the form $a.f = b$, where both a and b are reference variables). Say the object pointed-to by a is the "lhs" and the one pointed-to by b is the "rhs". Correspondingly, say lhsReg and rhsReg contain the addresses of the lhs and the rhs objects, respectively. We also assume that stackBaseReg and stackEndReg store the starting and the ending addresses of the stack region of the memory, respectively. The algorithm essentially has three consequences. We do not need any heapification when the rhs object is either outside stack bounds (i.e., already on the heap) or when our address comparison finds that the rhs object has a longer lifetime compared to the lhs object (conditions at lines 1 and 8). We definitely need heapification when we are sure of an escape, which is the when the lhs object is on the heap (condition at line 5). We may or may not need a heapification when both the lhs and the rhs objects are inside the stack bounds but the address comparison fails (line 10), whereby we need to perform a stack walk to determine if the rhs object escapes.

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

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

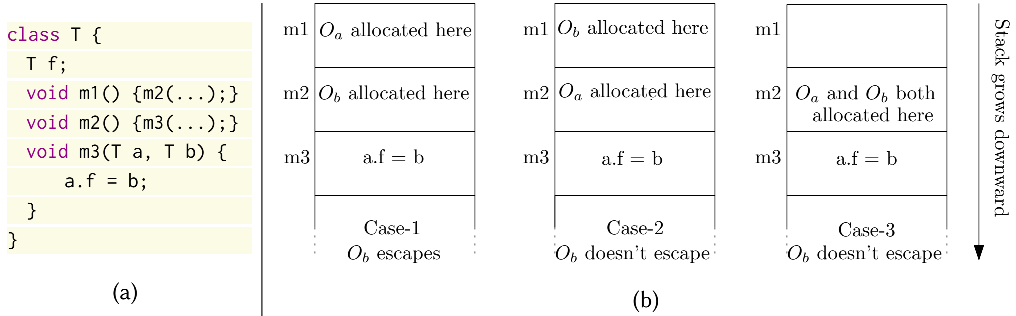


Fig. 5. (a) Example code to explain the scenarios for stack ordering. (b) Possible stack layouts while processing store statements for the example in (a).

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

Example. After dynamic classloading in Figure 2, our heapification check (inserted before the bytecode corresponding to the new store at line 14) would detect an incorrect stack allocation for the object O_7, move it to the heap, and recursively heapify the reachable object O_9. Further, it would update the reference variable p in doo to point to the new O_7 on the heap (see Figure 3).

4 IMPARTING EFFICIENCY THROUGH STACK ORDERING

As discussed in Section 3, when an object is stack-allocated optimistically, there is a possibility that it escapes at run-time either due to unsoundness in the static analysis or due to other dynamic features. Hence, as discussed previously, we perform checks at various statements to find whether such an optimistically stack allocated object is escaping. Recall that the check at a field-store statement is performed by comparing the lifetimes of the lhs and the rhs objects involved therein. If the lhs of the store is a heap-allocated object then the rhs object is escaping by default. However, in the case where both the lhs and the rhs objects are allocated on the stack, three cases arise.

Consider the code snippet shown in Figure 5a. Say there are three methods m1, m2 and m3. Here, m1 calls m2, m2 calls m3, and m3 collects two parameters a and b passed by its caller. Let a be pointing to an object O_a, b be pointing to an object O_b, and the method m3 creates a field reference from O_a to O_b (via the store statement a.f = b). Note that while being in m3, the allocation sites of the objects pointed-to by its parameters, and consequently their lifetimes, are unknown.

In general, the lifetimes of two stack allocated objects involved in a store statement could vary as shown in Figure 5b. The lifetime of the rhs object in a field store could be longer, shorter or equal to that of the lhs object. For the first two scenarios (Case-1 and Case-2), when the lhs and the rhs objects are allocated in different frames, an address comparison between the lhs and the rhs objects is sufficient to determine if the rhs is escaping or not. For Case-3, during the creation of the stack frame, the lhs and the rhs objects could be ordered arbitrarily and an address comparison is not sufficient to conclude whether the rhs object is escaping. Thus, we may have to perform a stack walk to find out if the lhs and the rhs objects are indeed on the same stack frame or not. However,

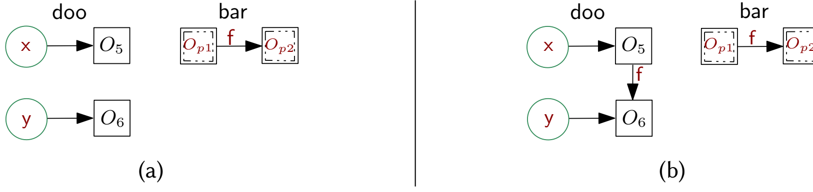


Fig. 6. Points-to graphs for methods doo and bar from Fig. 2 during stack ordering: (a) intraprocedural; (b) interprocedural. Nodes O_{p_1} and O_{p_2} represent the objects pointed-to by parameters p1 and p2, respectively.

even though the previously discussed stack-walk check provides this information precisely, it is expensive as well.

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

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

Example. Figure 6a shows the intraprocedural points-to graphs for methods doo and bar from Figure 2. As can be seen, there is no points-to edge currently between O_5 and O_6 and hence no stack order. However, once we merge the points-to graphs of doo and bar, as shown in Figure 6b, we can infer the stack order between the two objects in doo: O_5 should be placed before O_6.

In the VM, we use the statically provided stack orders to reorder the candidates list (candidates are the objects that have been marked for stack allocation by the VM; see Section 3.1). Afterwards, the stack-creation phase of the JIT compiler carries out allocation according to the reordered candidates list. In a nutshell, these stack orders allow us to allocate the objects in the stack frame deterministically, so that the underlying address comparison underneath heapification checks, which is cheap, is also sufficient whenever possible.

Note that in presence of cycles in the points-to graph, the predetermined stack order may not provide the desired precision. However, if there is a cycle involving n objects, address comparison based on the order of store statements among those objects would still be conclusive for n-1 store statements and fail for only one store statement. This failing store would require a stack walk to determine whether the rhs object needs to be heapified. Observe that this is a significant improvement over requiring a stack walk for all such store statements towards determining the need for heapification. In Section 6, we compare the versions of our implementations with and without stack orders, and observe that they indeed put the nail in the coffin, imparting efficiency to the complete idea of optimistic stack allocation with dynamic heapification.

5 DISCUSSION

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

5.1 Correctness

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

Theorem 5.1. *An object that outlives the scope of its allocating method is guaranteed to be present on the heap before it could be dereferenced.*

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

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

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

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

5.2 Design choices

1. Implementation over existing escape analysis. Recall that we use static-analysis results on top of the existing escape analysis in the JIT compiler (Section 3), though in principle we could have replaced the existing escape analysis altogether. However, we found that the existing analysis was able to handle certain scenarios better than a static analysis; for example, at a call site involving virtual dispatch, sometimes the JIT can speculatively inline one of the targets whereas a static analysis would conservatively take a meet among the escape statuses of the objects passed to possibly multiple methods. Furthermore, the existing analysis is quite fast (due to its imprecise nature) and does not affect compilation time noticeably.

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

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

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

6 EVALUATION

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

The enhancement in stack allocation can be measured in multiple ways: (i) by counting the number of additional allocation sites that our approach allows to be marked for stack allocation – we compute this by simply incrementing a counter during JIT compilation; (ii) by counting the actual number of additional objects allocated on the run-time stack – we do this by inserting a dynamic counter through the codegen phase of the compiler, which gets incremented every time an object is allocated on the stack during program execution; and (iii) by measuring the actual amount of additional memory that gets allocated on the stack – we compute this by adding up the sizes of all the objects allocated on the stack during program execution, again by instrumenting the codegen phase of the compiler. We also present insights from a fine-grained analysis of the features that lead to some of the highest stack allocations. These results are discussed in Section 6.2.

The impact of additional stack allocation itself on a program's execution can also be measured, albeit in more non-trivial ways. We compute the most direct measure – the run-time performance – in a steady state environment for all the benchmarks under consideration, and show that we sometimes improve but never degrade the performance in default settings. However, reckoning that stack allocation assumes more importance in constrained memory environments, we additionally demonstrate reduction in GC cycles (and consequently, performance improvements) by lowering the amount of heap memory supplied to the JVM. These results are discussed in Section 6.3.

Finally, in Section 6.4, in order to establish that even the offline cost of our approach is reasonable, we report the times taken by the Soot-based static analysis as well as the size overheads of the .res files that are to be supplied additionally to the VM during program execution.

6.1 Experimental setup

Here we list down the benchmarks used for our evaluation, the machine setup, and the comparison modes (along with their abbreviations).

Benchmarks and machine. We have used benchmarks from the DaCapo suites 23.10-chopin and 9.12-MRI [3], and SPECjvm 2008 [19]. The benchmarks skipped from these suites are those that cannot be analyzed with the used version of Soot and TamiFlex (error reports available on GitHub). We have performed our experiments on a 12th Gen Intel(R) Core(TM) i7-12700 system with 20 cores and 16 GB RAM, running Ubuntu 22.04.1 LTS. Our static analysis has been written using Soot 3.1 [23], with reflection logs for resolving reflective calls generated using TamiFlex 2.0.3 [5]. Our modifications to the VM (both interpreter and JIT) have been made over the latest version of Eclipse OpenJ9 [10] built with JDK8 (OpenJ9 commit b4cc246, OMR commit 162e6f7, JCL commit 2a5e268); we chose this JDK version because Soot+TamiFlex do not yet fully support newer Java bytecodes (such as invokedynamic), requiring our static analysis to handle the call sites involving them conservatively. However, in order to be able to test our scheme with the newer (and larger) benchmarks from the latest DaCapo suite (23.11-chopin) that cannot be executed with JDK8, we have also ported our implementation to the latest OpenJ9 built with JDK21 (OpenJ9 commit 7c701a7, OMR commit 87019fe, JCL commit 07e5992) and report the results therein.

In all our experiments, we run each benchmark long enough to account for warmup: for DaCapo 9.12-MRI we take the number of required warmup iterations to achieve a steady state from a prior work that used these benchmarks [15], for DaCapo 23.10-chopin we ran each benchmark for 100 warmup iterations, and for SPECjvm benchmarks we use the existing warmup built into its harness.

	Non Optimistic Scheme (BASE)			Optimistic scheme (OPT)		
Bench mark	**Static Counts**	**Dynamic Counts**	**Stack Bytes**	**Static Counts**	**Dynamic Counts**	**Stack Bytes**
biojava	0 (0.00%)	0M (0.00%)	0MB	0 (0.00%)	0M (0.00%)	0MB
graphchi	0 (0.0%)	0M (0.00%)	0MB	32 (4.15%)	506.3M (6.9%)	9184.6MB
h2o	1 (0.02%)	2.6M (0.01%)	42.2MB	2 (0.02%)	3.1M (0.01%)	50.7MB
jme	0 (0.00%)	0M (0.00%)	0MB	0 (0.00%)	0M (0.00%)	0MB
kafka	0 (0.00%)	0M (0.00%)	0MB	0 (0.00%)	0M (0.00%)	0MB
zxing	0 (0.00%)	0M (0.00%)	0MB	2 (0.05%)	0.2M (0.01%)	0.47MB
avrora	9 (0.98%)	0.008M (0.01%)	0.1MB	12 (1.12%)	0.03M (0.04%)	0.9MB
eclipse	7 (0.07%)	8.9M (1.06%)	214MB	31 (0.32%)	9M (1.18%)	252MB
fop	10 (0.15%)	0.04M (0.002%)	1MB	50 (0.77%)	9.8M (0.42%)	161.2MB
h2	61 (2.33%)	29M (0.92%)	523MB	94 (3.87%)	452M (13.92%)	10801MB
luindex	35 (1.35%)	3M (2.39%)	98MB	89 (3.49%)	5M (3.49%)	133MB
lusearch	30 (1.09%)	924M (3.23%)	775MB	78 (3.05%)	928M (7.4%)	1686MB
pmd	89 (1.90%)	52M (7.20%)	1310MB	191 (3.97%)	105M (14.2%)	2465MB
sunflow	114 (10.05%)	1258M (20.08%)	30439MB	161 (13.86%)	1260M (20.11%)	30498MB
compiler	93 (1.73%)	94M (5.50%)	1720MB	137 (2.75%)	105M (6.17%)	2329MB
compress	8 (1.30%)	0.01M (3.29%)	0.2MB	13 (2.22%)	0.01M (3.8%)	0.39MB
fft	3 (0.73%)	12 (0.01%)	0.0002MB	3 (0.8%)	187 (0.16%)	0.003MB
lu	6 (0.83%)	85 (0.08%)	0.002MB	11 (3.30%)	379 (0.3%)	0.009MB
montecarlo	9 (1.50%)	2627 (2.02%)	0.09MB	9 (1.63%)	0.006M (4.47%)	0.4MB
aes	8 (1.03%)	0.01M (2.79%)	0.3MB	16 (2.25%)	0.01M (2.8%)	0.3MB
rsa	16 (1.13%)	0.1M (1.1%)	46MB	35 (3.18%)	7M (4.62%)	170MB
signverify	15 (0.84%)	0.24M (0.86%)	6.8MB	51 (3.10%)	2.1M (7.24%)	49.4MB
Overall	514	2372M	35176MB	1017	3393M	57782MB

Table 1. Stack allocation statistics for various benchmarks with BASE and OPT schemes. The first six benchmarks (biojava–zxing) are from DaCapo 23.10-chopin, the next eight (avrora–sunflow) are from DaCapo 9.12-MRI, and the last eight (compiler–signverify) are from SPECjvm 2008.

Also, we disabled shared-class cache (a feature in OpenJ9 that allows loading of previously compiled code in subsequent runs) so that our experiments are independent of previous executions.

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

6.2 Enhancement in stack allocation

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

6.2.1 Static and dynamic number of allocations. For both the schemes BASE and OPT, the first and the second columns in Table 1 show the number of allocation sites marked for stack allocation during JIT compilation ("Static Counts") and the actual number of objects allocated on stack during program execution ("Dynamic Counts"), respectively. The parentheses in each of the columns contain the percentages of objects identified under each category, relative to the total number of static and dynamic objects, respectively. As can be seen, BASE by default is able to identify and stack allocate very few objects for most benchmarks, sunflow being the only exception where the dynamic counts go up to 20.08%. This is because the existing JIT analysis is imprecise and can work

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

Table 2. Stack allocation statistics with forced compilation of all the methods, for the benchmarks where a longer run leads to noticeable improvements.

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

On the other hand, our scheme OPT is able to improve the stack allocation significantly. The static number of object allocation sites identified for stack allocation, over all the benchmarks, increases by about 98% (from 514 to 1017), and the dynamic number of object allocations on the stack increases by about 43% (from 2372M to 3393M). Note that the improvement in dynamic percentage of stack allocations could be much higher than that in the percentage of allocation sites identified statically; for example, for h2, OPT is able to stack allocate 13.92% of objects on stack even with the static allocation sites being 3.87%. This is because of objects created inside loops, which are identified as stack allocatable by our scheme even if they are passed to other methods, due to the precise interprocedural nature of the results used by OPT. Few notable benchmarks where the improvement is high are the h2, pmd and signverify, wherein OPT increases the dynamic percentage of stack allocation from 0.92%, 7.20% and 0.86%, respectively, to 13.92%, 14.20% and 7.24% of the total number of objects.

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

6.2.2 Simulating longer runs of benchmarks with forced JIT compilation. Noting the near-zero numbers for both BASE and OPT for many of the DaCapo 23.11-chopin benchmarks, we investigated their profiles and found that during their execution with 100 iterations very few methods got JIT-compiled at levels that perform escape analysis and stack allocation in OpenJ9. Observe that some of these benchmarks (such as kafka, which is a real-time streaming system) are indeed supposed to be run for a really long duration of time, which might lead to warmer JIT compilations subsequently; we verified this by running some of these benchmarks for thousands of iterations and noticing that stack allocations started to increase. In order to approximate such runs, and to estimate the maximum improvements our scheme could bring therein, we lowered the JIT compilation threshold of OpenJ9 to one invocation and recomputed the number of stack allocations; Table 2

shows the benchmarks for which either BASE or OPT scheme showed a significant improvement compared to the "default-mode" runs reported in Table 1. As we can see, under this configuration, even BASE performs a few stack allocations for all the benchmarks. Importantly, we see much better improvements with OPT for the benchmarks jme and kafka (where the number of objects allocated on stack increase to 35.6% and 4.37%, respectively). Similarly, we found noticeably higher improvements for the benchmarks fop and h2 in this configuration. With these observations, we believe that our scheme could be used to make case for a stack-allocation pass even at lower levels of compilation, or devices with even lower JIT budget, as it has the potential to improve the number of stack allocations without performing any escape analysis during JIT compilation.

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

First of all, as OPT's escape analysis is performed statically, it analyzes all the methods interprocedurally; whereas BASE performs escape analysis only for methods that are compiled at hot+ levels of compilation. Furthermore, as the peek depth of BASE is quite limited (usually just zero or one, unless all the callees are compiled at high levels of compilation), it marks objects passed as arguments beyond the peek depth as escaping. Even when BASE peeks a callee, it is limited in terms of the analysis information (e.g., use-def information) that it builds for peeked methods as it does not have the necessary compile-time budget to perform these kinds of analyses in the quest for precision, whereas the OPT does not have these kinds of constraints. For example, in the benchmark h2, we found a class RowList whose instances (>2 million in number) are used to store rows from a database. The benchmark code uses rows as the receiver for a call rows.add, which further calls a chain of methods starting with the same receiver. BASE fails to perform interprocedural analysis of the nested calls and marks the object pointed-to by rows as escaping (not the case with OPT). Note that it is possible to make BASE more intricate and capture such objects, which gets evident upon noticing the improvements with forced compilation in Table 2 for h2 (as well as others), but this increases the time spent in JIT compilation (successfully avoided by OPT).

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

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

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

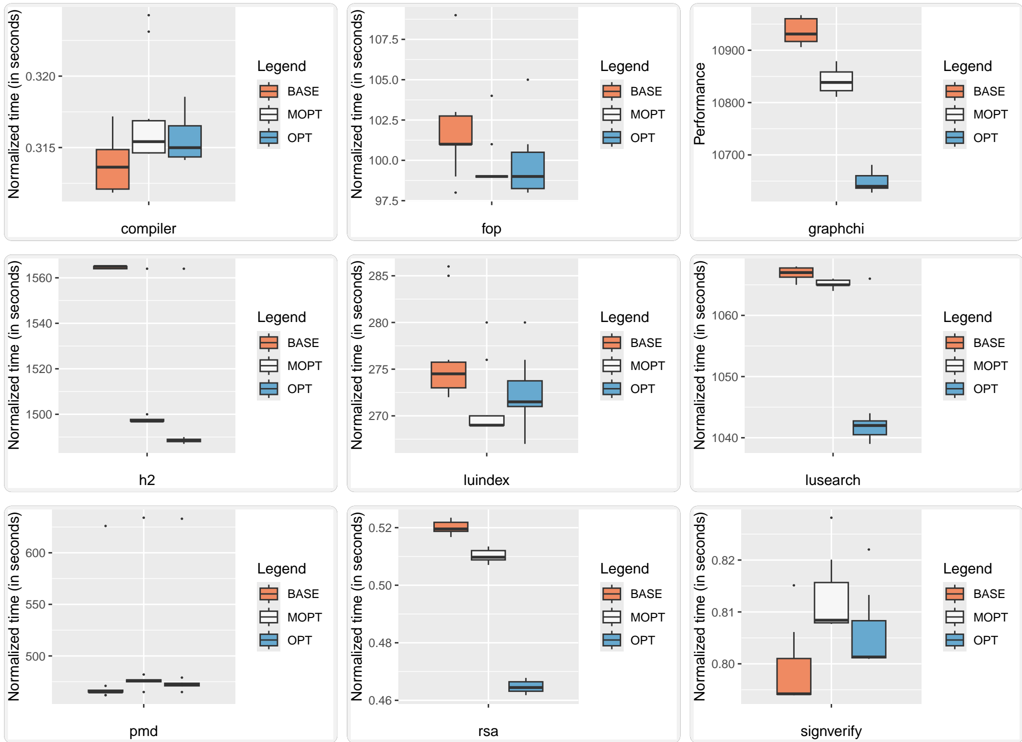


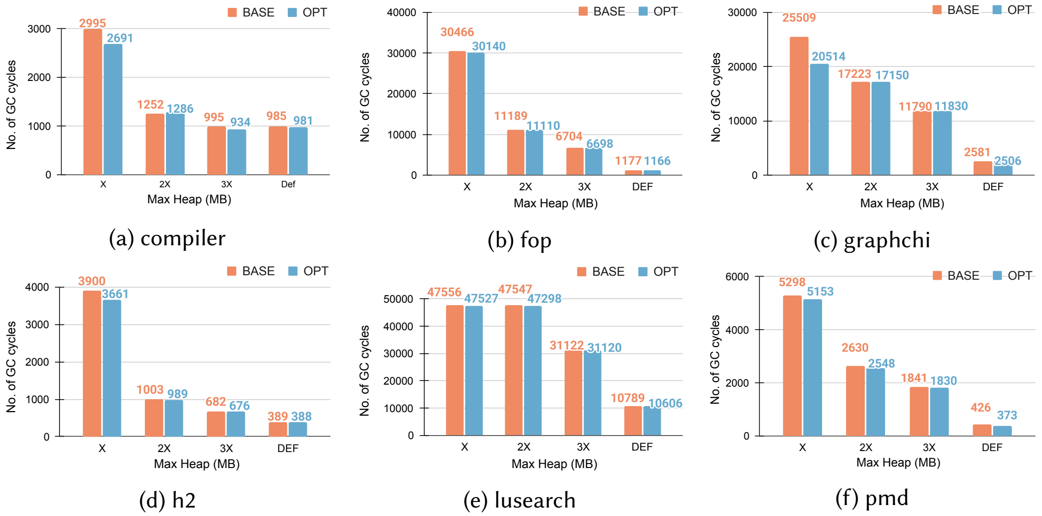
Fig. 7. Performance comparison (default heap size); lower the better. For DaCapo benchmarks the y-axis is time (in msec) and for SPECjvm benchmarks it is the normalized reciprocal of ops/sec.

Overall, we assert that our static+dynamic scheme shows a marked improvement in stack allocation compared to the baseline, and we further measure its impact on performance in Section 6.3.

6.3 Impact of additional stack allocation

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

6.3.1 Impact on performance (default memory). Figure 7 shows the performance box plots over ten iterations in steady state for the three modes outlined in Section 6.1: BASE, MOPT and OPT. We have skipped showing flat results for benchmarks that show a negligible difference across the various modes. For the nine benchmarks in Figure 7, we can sometimes see a small increase in the time taken from BASE to MOPT. This is due to the overhead of heapification checks performed as part of evaluating the potential escape-causing statements as discussed in Section 3.2. On the plus side, with our stack-ordering enabled implementation OPT, we see that the time taken either matches that of BASE (compiler, luindex, pmd) or even improves noticeably (fop, graphchi, h2, lusearch, rsa). Further, though we occasionally see a (very) slight increase in time with OPT, we later show (in Section 6.3.3) that improvements with OPT are much more pronounced once we reduce the currently very high amounts of heap memory supplied to the JVM, for the benchmarks with good stack allocation. Thus, it can be asserted that our proposed scheme is capable of allowing much higher stack allocation without causing performance overheads, and that the performance improves too



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

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

Fig. 8. Number of GC cycles with varied heap sizes (X, 2X, 3X and default heap memory); lower the better.

when the additional stack allocation starts showing benefits (possibly in terms of faster variable accesses and reduced GC cycles; we measure the latter next).

6.3.2 Impact on garbage collection. One of the relevant ways of measuring the impact of stack allocation of objects is to compare the number of garbage collection (GC) cycles between BASE and OPT. To perform this experiment, we vary the amount of maximum heap memory made available to the JVM process (using the `-Xmx` argument) and compute the number of GC cycles (using the `-verbose:gc` argument of OpenJ9) for both BASE and OPT. More the allocation on stack (and conversely, less the allocation on heap), we expect to see fewer GC cycles. Further, lesser the heap memory supplied to the VM, more pronounced should be the impact of allocating bytes on stack rather than on the heap. Now consider the results shown for six benchmarks (chosen based on the object-allocation profile and improvements observed in Section 6.2) in Figure 8. Here, we calculated the minimum heap size (rounded to multiples of 5) required to execute each benchmark (say X) and then computed the number of GC cycles by setting the available heap memory to X, 2X (moderate heap) and 3X (generous heap), as well as the default heap size (DEF), which is 4 GB for our machine.

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

6.3.3 Impact on performance (reduced memory). Motivated by observing the reduction in GC cycles in Section 6.3.2, specially when the memory made available to the JVM was limited to the minimum heap required to execute a given benchmark, we conducted an experiment to compare performance with our scheme under this configuration. Figure 9 shows the obtained box plots for the three modes under consideration for the benchmarks from Section 6.3.2.

As can be seen, we obtain clear performance improvements for five out of six benchmarks with higher stack allocation (compiler, fop, graphchi, h2 and lusearch), both with MOPT and OPT. With 90% confidence, the average performance improvement with MOPT is 4.5% and with OPT it is 8.8%.

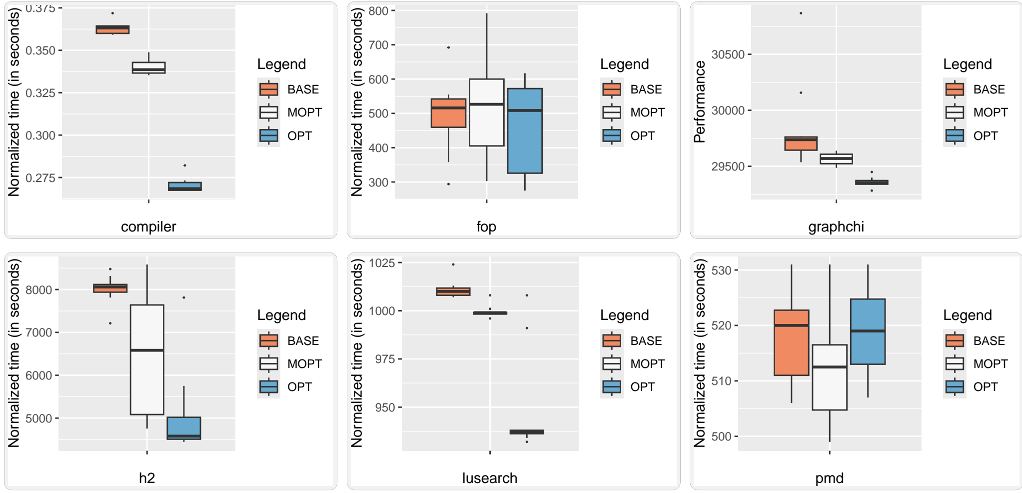


Fig. 9. Performance comparison (minimum heap size for each benchmark); lower the better. For DaCapo benchmarks the y-axis is time (in msec) and for SPECjvm it is the normalized reciprocal of ops/sec.

The highest seen improvement is for h2 (about 18.9% with MOPT and 37% with OPT); this is due to the significantly high increase in stack allocation with our scheme (about 20× more number of stack bytes; see Table 1), as well as the reduction in GC cycles at this configuration (from 3900 to 3661 cycles; see Figure 8) for this benchmark.

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

Overall, with the above two measurements (reduction in GC cycles and performance improvements on lower memory systems), we can assert that our scheme is successful in reducing the amount of garbage collection a Java runtime has to perform during program execution, and that it can allow one to run memory-intensive applications even on lower-memory systems.

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

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

Fig. 10. Time taken by our static analysis along with the .res file overhead.

heapification checks, is indeed effective in scenarios where a dynamic feature may cause optimistic stack allocation to fail occasionally, and is thus practical to be employed in standard runtimes.

6.4 Offline cost

Finally, observe that the advantages of our scheme – both in terms of additional stack allocation and efficient heapification checks – stem from a precise (flow-, field- and 1-level context-sensitive) static escape analysis; in fact, one could apply any of the recent advances in static analysis, for example object and type sensitivity [14, 18], and keep getting better. The advantage of performing analysis of Java Bytecodes statically is that the amount of time spent in gaining additional precision does not affect the execution time of the program. For completion, we now report the times taken, and the sizes of the .res files generated, by our static analysis implementation; see Figure 10.

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

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

7 RELATED WORK

To the best of our knowledge, ours is the first work that uses static analysis to perform stack allocation in a VM while supporting a dynamic repair mechanism involving heapification and stack ordering. Existing works involving JVMs do perform escape analysis and resultant optimizations of various kinds though, and we discuss them in this section.

One of the first extensive proposals to perform escape analysis for Java was implemented in the Jalapeño VM [1, 24], which introduced the points-to escape graph notation for denoting points-to relationships as well as performing reachability-based escape analysis. Another popular abstraction

for performing escape analysis, proposed by Choi et al. [7], is that of connection graphs, which do not maintain points-to relationships directly but allow one to perform check reachability faster. A partially interprocedural version of this approach is used by the C2 compiler of the HotSpot JVM [17] to perform synchronization elision and scalar replacement. The approach proposed by us in this paper for improving the partially interprocedural escape analysis of Eclipse OpenJ9 [10], on the other hand, is context as well as flow sensitive, and uses points-to graphs as they additionally allow us to determine stack orders (which will not be possible with connection graphs).

Kotzmann and Mössenböck [11, 12] proposed an escape analysis that works in presence of dynamic classloading for the C1 compiler of the HotSpot JVM. Essentially, they reallocate objects replaced by scalars if the VM deoptimizes to the interpreter. The more recent GraalVM builds up on this idea and performs scalarization of objects while rematerializing them in branches where they escape [20]. As some of the stack-allocated objects can also be replaced with scalars, it would be interesting to complement our implementation with scalar replacement. Interestingly, the current VM-only versions of these works require additional bookkeeping in order to materialize objects, which can possibly be reduced if (part of) the analysis was performed statically like our approach. We mark this as an interesting future extension.

The closest related works we found are those by Corry [9] and Cleereman et al. [8]. The first one proposes an address check at write barriers to detect objects that live longer than the stack frame being destroyed, but performs stack allocation primarily around loops and by maintaining a separate region for object allocation on stack. The second one proposes to address dynamic classloading that may cause previously non-escaping objects to escape, by modifying the classloader and maintaining special dirty bits on the stack frame, but has no evaluation. Our approach, on the other hand, covers dynamic features including dynamic classloading, and does not require modifications to the classloader, stack management, or to any other traditional routines of the JVM; further, our work is supported by an extensive evaluation over a production JVM.

A recent work proposed a framework [21] for performing expensive analysis statically and using its results during JIT compilation, in context of libraries that may differ between static and dynamic compilation, and another work formalized the idea of staging program analyses across static and JIT compilation [2]. Similar to them, we use bytecode indices to represent static-analysis results that are usable in a JVM. However, adapting their approaches for stack allocation would pose the exact problem that we solve in this paper: timely invalidating the results in presence of dynamic features such as hot-code replacement, dynamic classloading, and even the possibility of different libraries. Nevertheless, we are encouraged by this trend and believe static+dynamic analysis has many further applications that are yet to be explored.

An alternative approach of optimizing memory, popular particularly in the functional programming community [22], is to divide the run-time memory into inferred regions that can lead to faster access and cleanup of objects in the current region (a region may include multiple stack frames). Region inference is more expensive than escape analysis and is not employed by current object-oriented runtimes. However, we feel approaches like ours can be used to improve the efficacy of region-based memory allocation and open up the possibility of experimenting with interesting stack and heap allocation strategies, for popular OO runtimes as well.

8 CONCLUSION

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

Our work picked up an important OO optimization – that of allocating method-local objects on the stack frames of their allocating methods – and showed that using a static escape analysis to optimistically allocate identified objects on stack could improve the precision without thwarting the efficiency. Further, in order to ensure functional correctness in case the static-analysis results do not correspond to the run-time environment, it proposed the ideas of (i) dynamically checking incorrect stack allocations before they could cause an issue; (ii) repairing the memory layout by heapifying escaping objects and correcting their references; and (iii) doing so efficiently by ordering the objects on stack in a statically identified manner.

Our evaluation on a production JVM, across its interpreter and compiler infrastructure, showed that the proposed scheme is capable of bringing the benefits of a precise escape analysis for improving performance in low-memory systems, by reducing memory pressure and correspondingly, the garbage-collection overheads. Further, it was a significant innovation challenge not just to design a scheme that involved multiple levels of abstractions and coding environments, but also to implement it and reach a stage where it is ready to be integrated into an industry virtual machine.

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

REFERENCES

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. 2000. The Jalapeño virtual machine. *IBM Systems Journal* 39, 1 (2000), 211–238. https://doi.org/10.1147/sj.391.0211

[2] 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. https://doi.org/10.1007/978-3-031-22308-2_4

[3] 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.

[4] Bruno Blanchet. 2003. Escape Analysis for JavaTM: Theory and Practice. *ACM Trans. Program. Lang. Syst.* 25, 6 (Nov. 2003), 713–775. https://doi.org/10.1145/945885.945886

[5] 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. https://doi.org/10.1145/1985793.1985827

[6] 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. https://doi.org/10.1145/320384.320386

[7] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. 2002. Efficient and Precise Datarace Detection for Multithreaded Object-oriented Programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. ACM, New York, NY, USA, 258–269. https://doi.org/10.1145/512529.512560

[8] Kevin Cleereman, Michelle Cheatham, and Krishnaprasad Thirunarayan. 2007. Runtime Support of Speculative Optimization for Offline Escape Analysis. In *Proceedings of the International Conference on Software Engineering Research and Practice*. 484–489. https://corescholar.libraries.wright.edu/knoesis/884

[9] Erik Corry. 2006. Optimistic Stack Allocation for Java-like Languages. In *Proceedings of the 5th International Symposium on Memory Management* (Ottawa, Ontario, Canada) *(ISMM '06)*. Association for Computing Machinery, New York, NY, USA, 162–173. https://doi.org/10.1145/1133956.1133978

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

[11] Thomas Kotzmann and Hanspeter Mössenböck. 2005. Escape Analysis in the Context of Dynamic Compilation and Deoptimization. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments* (Chicago, IL, USA) *(VEE '05)*. Association for Computing Machinery, New York, NY, USA, 111–120. https://doi.org/10.1145/1064979.1064996

[12] 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. https://doi.org/10.1109/CGO.2007.34

[13] Ondrej Lhoták and Laurie Hendren. 2003. Scaling Java points-to analysis using SPARK. *International Conference on Compiler Construction* 2622, 153–169. https://doi.org/10.1007/3-540-36579-6_12

[14] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized Object Sensitivity for Points-to Analysis for Java. *ACM Trans. Softw. Eng. Methodol.* 14, 1 (Jan. 2005), 1–41. https://doi.org/10.1145/1044834.1044835

[15] 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.

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

[17] 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.

[18] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick Your Contexts Well: Understanding Object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. ACM, New York, NY, USA, 17–30. https://doi.org/10.1145/1926385.1926390

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

[20] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2018. 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. https://doi.org/10.1145/2544137.2544157

[21] 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. https://doi.org/10.1145/3337794

[22] Mads Tofte and Lars Birkedal. 1998. A Region Inference Algorithm. *ACM Trans. Program. Lang. Syst.* 20, 4 (jul 1998), 724–767. https://doi.org/10.1145/291891.291894

[23] 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. http://dl.acm.org/citation.cfm?id=781995.782008

[24] 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.