

Runtime Support of Speculative Optimization for Offline Escape Analysis

Kevin Cleereman, Michelle Cheatham
Air Force Research Laboratory
2241 Avionics Circle
Wright-Patterson AFB, OH, 45433-7334
(937-904-9090), (937-904-9011)

{kevin.cleereman, michelle.cheatham}@wpafb.af.mil

Krishnaprasad Thirunarayan
Wright State University
3640 Colonel Glenn Hwy
Dayton, OH 45435
(937-775-5109)
t.k.prasad@wright.edu

Abstract

Escape analysis can improve the speed and memory efficiency of garbage collected languages by allocating objects to the call stack, but an offline analysis will potentially interfere with dynamic class loading and an online analysis must sacrifice precision for speed. We describe a technique that permits the safe use of aggressive, speculative offline escape analysis in programs potentially loading classes that violate the analysis results.

Keywords

Escape Analysis, Dynamic Class Loading, Speculative Optimization, Automatic Refactoring

1. INTRODUCTION

Escape analysis is used to determine when it is safe to allocate an object on the call stack instead of the heap, thereby reducing synchronization and garbage collection overhead. However, escape analysis typically requires a conservative whole program analysis, which complicates its use in environments supporting dynamic class loading (as in Java) or dynamic class generation/modification (as in CLOS), and which rules out the use of powerful speculative pointer analyses ([6]) to power the escape analysis. A conservative online analysis ([7]) can tolerate dynamic class loading, but sacrifices precision for analysis speed and does not provide a framework for speculative escape analysis. We present an approach for supporting aggressive, speculative offline escape analysis in environments with dynamic class loading that incurs negligible overhead in the case that the resulting optimizations are sound, and that incurs low overhead in the case that the resulting optimizations prove to be unsafe.

2. STACK ALLOCATION AND DYNAMIC CLASS LOADING

Stack allocation can be implemented explicitly through type annotations inserted by the programmer (e.g., using a `SCOPED` keyword to indicate that a reference is scoped to the allocating procedure, or using a `REGMALLOC` keyword to indicate the memory region that defines the scope of a reference) or implicitly through escape analysis. Explicit annotations simplify program compilation, however, the need to insert scoping annotations places an additional

burden on the programmer which will hinder program development, and the typing rules for the scoping annotations must be conservative to facilitate fast type-checking. For example, in the Real-Time Specification for Java ([3]), scoping annotations are flow-and-context-insensitive, whereas escape analyses may be flow-sensitive and/or context-sensitive. Such scoping annotations also hinder software evolution, because although a reference may not need to escape its scope in the current project iteration there is rarely any guarantee that it will not need to escape its scope in future project iterations, such as via the inclusion of new subclasses and overriding methods. (Type annotations would, of course, be appropriate when the correctness of the program hinges on a reference not escaping its scope. Nevertheless, it would be preferable to be able to employ arbitrary escape analyses to ensure type (or typestate) safety, rather than relying on conservative typing rules.) Thus, although verified software with conservative scope annotations is guaranteed to be type-safe even in the face of dynamic class loading, this safety comes at a significant cost to the programmer by requiring additional annotations and hindering code reuse.

Escape analysis ([2], [4], [9], [11]) provides a more flexible solution to the problem of determining which objects may be stack allocated. It places no additional burden on the programmer, and by doing away with conservative typing rules the analysis will potentially be able to allocate more objects to the stack. Unfortunately, dynamic class loading presents a problem in the absence of type annotations. Escape analysis ought not to interfere with semantically valid changes that are made to a program, even if those changes violate the results of earlier escape analyses. For example, it would violate the software designer's specifications were an escape analysis to allocate an object to the stack and subsequently reject any dynamically loaded class that attempted to treat the object as heap-allocated. However, if the language runtime does not reject the dynamically loaded class that violates the inferred scoping rules, then we will potentially create dangling references when the stack-allocated object is popped along with its enclosing stack frame. Corry ([5]) presents a technique that prevents dangling references when a stack-allocated object escapes (his approach targets optimistic stack allocation, which is a more general problem than dynamic class

loading and speculative optimization), but requires potentially expensive runtime stack inspection. (The technique we describe in this paper avoids runtime stack inspection, but requires a potentially expensive write barrier and partial heap trace.)

In this paper we make no assumptions about which objects may be stack-allocated, in particular we do not assume that the size of stack-allocated objects is precisely known or even bounded at compile-time. In the case that a stack-allocated reference points to a heap-allocated object (e.g., a stack-allocated `ArrayList` reference would need to point to a heap-allocated `ArrayList` object unless a reasonable bound for the array's size could be computed at compile-time) we assume that the language runtime will handle the task of finalizing the heap object when the stack frame is popped. We assume that it is possible to determine if an object is stack-allocated by inspecting its address (as this simplifies the write barrier), therefore the language runtime will likely require a special heap (the `RegionHeap`) for lexically scoped objects that do not fit on the stack.

Thus, each thread has a thread-local call stack and a thread-local `RegionHeap` for lexically scoped objects with unbounded size. When a stack frame is popped, then objects in the `RegionHeap` within the frame's lexical scope are also deleted. In addition, the `RegionHeap` may be garbage collected, but note that the `Heap` may not hold references to objects on the `RegionHeap`, so garbage collecting the `RegionHeap` is considerably faster than garbage collecting the entire `Heap`. Finally, the shaded areas of the stacks contain `Immortal` objects that exist for the lifetime of the thread. We do not assume which offline escape analysis is responsible for segregating stack allocated objects from heap allocated objects – our runtime monitoring is aimed at (possibly speculative) escape analyses that do not account for dynamic class loading (and therefore isn't needed if the non-speculative escape analysis being used already accounts for dynamic class loading).

3. ESCAPING STACK REFERENCES

Offline escape analysis can violate memory safety in the presence of dynamic class loading by producing dangling references. Consider the classes `Source`, `ArraySink`, and `Sink` (Figure 1, Figure 2, Figure 3).

```
Class Source {
    private ArrayList <Sink> x;
    void Main(ArraySink z) {
        z.compareAll(x);
    }
}
```

Figure 1

```
Class ArraySink {
    void compareAll(ArrayList <Sink> x) {
        ListIterator l1 = x.ListIterator(0);
        ListIterator l2 = x.ListIterator(1);
        while(l2.hasNext()) {
            l1.next().Compare(l2.next());
        }
    }
}
```

Figure 2

```
Class Sink {
    boolean Compare(Sink b) {
        return (b == this);
    }
}
```

Figure 3

References to `ArrayList <Sink> x` in class `Source` never escape the scope of `Source`, therefore instances of `Source` with the inlined `ArrayList` may be stack allocated independently of whether the `ArraySink z` parameter is allocated to the stack or the heap. However, this analysis relies on the semantics of both `ArraySink` and `Sink`, either of which may change in an environment supporting dynamic class loading or modification. For example, a dynamically loaded class `ArrayEscape` inheriting from `ArraySink` (Figure 4) can violate the escape semantics (Figure 5) inferred from the super `ArraySink` class.

```
Class ArrayEscape extends ArraySink {
    ArrayList <Sink> escape;
    void compareAll(ArrayList <Sink> x) {
        escape.add(x);
        ...
    }
}
```

Figure 4

```
ArrayEscape f2 = new ArrayEscape();
(new Source()).Main(f2);
System.print(f2.escape);
```

Figure 5

If a `Main` invocation on a `Source` object receives an `ArrayEscape` reference instead of an `ArraySink` reference as in the code sample above, then `ArrayList <Sink> x` will escape the scope of `Source` and cannot be stack allocated (unless we can guarantee that `ArrayEscape` references are always stack allocated

within the scope of the `Source` object). In an environment that does not support dynamic class loading this does not pose any problem, because a whole program analysis will reveal that `ArrayList <Sink> x` may escape in some contexts. However, if `ArrayEscape` is dynamically loaded, then the escape analysis on `Source`, `ArraySink`, and `Sink` that resulted in `ArrayList <Sink> x` being stack allocated will produce incorrect program semantics. Specifically, `ArrayEscape` will contain a dangling reference to the `ArrayList <Sink> x` object that is deallocated when `Source` is popped from the call stack.

We can solve this problem by immediately evacuating to the heap all stack-allocated objects that will potentially escape as soon as the `ArrayEscape` class is loaded. However, this will not only require runtime stack inspection to locate all potentially escaping objects to be copied onto the heap, but it will *also* require us to either leave redirects for the references pointing to the new locations of the potentially escaping objects (if the language runtime even supports redirects for stack-allocated objects), or else it will require us to perform additional runtime stack inspection to immediately update the call stack references to the potentially escaping objects. We propose an incremental approach instead.

A dynamically loaded class can also violate the inferred synchronization semantics ([1], [4]) by allowing inferred thread-local references to escape the allocating thread, potentially creating data races. We do not address this problem in this paper.

4. STACK MONITORS

Aggressive escape analysis can support dynamic class loading, as well as speculative optimizations, by using *stack monitors* to eliminate dangling references. This incurs $O(n)$ overhead whenever an escaping reference is popped from the stack (where n is the number of potentially capturing objects that point to the escaping object), and incurs $O(1)$ overhead when a dynamically-loaded class captures a stack-allocated object. We will refer to a stack-allocated object that will potentially escape its scope as an *escaping object*, and we will refer to the heap-allocated object causing the scope violation as the *capturing object*.

A `StackMonitor` (Figure 6) is a remembered-set that tracks a stack frame containing escaping objects, and maintains a list of all capturing objects causing scope violations. Each stack frame has an associated uninitialized `StackMonitor` object. When a heap-allocated object first captures an object allocated to the stack frame, the frame will call `NewRegister` to initialize the `StackMonitor`. Subsequent capturing objects in the same frame are registered with the `OldRegister`

method. When the stack frame is popped, the finalizer calls the `Unregister` method to copy reachable stack objects to the heap. The monitor stores weak references to the capturing objects to avoid interfering with the garbage collector.

```
Class StackMonitor {
    LinkedList <WeakReference> registry;
    boolean dirtybit = false;

    inline void Register(WeakReference w) {
        if(dirtybit) OldRegister(w)
        else NewRegister(w)
    }

    void NewRegister (WeakReference w) {
        dirtybit = true;
        registry = new LinkedList();
        registry.add(w);
    }

    void OldRegister (WeakReference w) {
        registry.add(w);
    }

    static void Unregister () {
        ListIterator <WeakReference> itr =
            registry.listIterator();
        while(itr.hasNext()) {
            Object temp = itr.next().get();
            System.Update(temp, thisFrame);
        }
    }
}
```

Figure 6

`System.Update(Object, Frame)` traverses the fields in `temp`, and moves any referenced objects onto the heap (leaving a redirect on the stack) if the reachable object is on the `thisFrame`. As noted in the write barrier discussion below, `System.Update` must also `Register` any reachable stack-allocated objects that are not in the current frame.

To accommodate the `StackMonitor` functionality, stack frames must store a `dirtybit` to track whether any of their stack allocated objects have escaped. When an object first escapes the frame's scope, the system sets the `dirtybit` and initializes the `StackMonitor` using `NewRegister`. Subsequently, an inlined `dirtybit` check results in calls to `OldRegister`. When popping the stack frame, a `StackFinalizer` checks the `dirtybit` and `UnRegisters` the frame if necessary.

The class loader is responsible for modifying the capturing objects so that they correctly register escaping references.

For example, after refactoring the `ArrayEscape` class by inserting a write barrier (Figure 7), whenever a method assigns to `escape`, the object first performs an inlined test to see if the escaping object's stack frame is higher than the currently captured stack frame (assuming that the call stack is contiguous and grows from low addresses to high addresses). `Frame1.higher(Frame2)` returns false if `Frame1`'s address is less-than or equal to `Frame2`, or if either `Frame1` or `Frame2` is the `nullFrame`. This prevents the capturing object from being registered multiple times for the same stack frame. `x.Frame()` returns the object's stack frame, or the `nullFrame` if the object is heap-allocated. However, it will be necessary for `System.Update` to register objects that capture stack allocated objects on lower frames than the one being popped. For example, if objects on `Frame1` and `Frame2` are both captured by `CapturingObject`, with `Frame2` higher than `Frame1`, then `CapturingObject` may not be registered with `Frame1`, but this is safe because there cannot be any dangling references to `Frame1` so long as `Frame2` has not yet been popped; therefore, registering `CapturingObject` to `Frame1` can be delayed until `Frame2` is popped (at which point `System.Update` is responsible for updating the `StackMonitors`). As an alternative, `CapturingObject` could maintain a hashtable of all captured stack frames, but this would make the inlined write barrier much more expensive; checking membership in a hash table may be a constant-time operation, but it is still much more costly than a simple arithmetic comparison. Another alternative is to associate one `escapeFrame` per capturing field; this simplifies the `System.Update` method and maintains the simplicity of the arithmetic comparison, but also incurs a linear space overhead.

```

Class ArrayEscape extends ArraySink {
    WeakReference w =
        new WeakReference(this);
    Frame escapeFrame = nullFrame;
    ArrayList <Sink> escape;
    void compareAll(ArrayList <Sink> x) {
        if(x.Frame().higher(escapeFrame)) {
            escapeFrame = x.Frame();
            x.Frame().Register(w);
        }
        escape.add(x)
        ...
    }
}

```

Figure 7

Lastly, the runtime system may refactor classes whose stack allocated objects escape their scope because of the semantics of a dynamically loaded class. However, this is

not strictly necessary, as it may be cheaper to continue stack allocating the escaping objects if they are rarely captured by objects in the dynamically loaded class.

5. PROGRAM FLOW

For this section we assume that a dynamically loaded class has violated the scoping rules of stack-allocated objects, and that all classes that stack-allocated escaping objects have been refactored so that they allocate to the heap. The state of a call stack is tracked with a `CorruptStack` field that holds the address of the highest stack frame that has escaping references (as before, we assume that the stack is contiguous and grows from low addresses to high addresses). If the `CorruptStack` address is lower than the top of the `Immortal` stack segment then there are no escaping references on the stack. (If the classes allocating escaping objects are not refactored (i.e., if they continue to allocate the escaping objects to the stack) then the program will never return to the steady state and it will not be necessary to maintain the `CorruptStack` field.)

5.1 Steady State

This is the default state of the program. The `CorruptStack` address is lower than the `Immortal` stack segment. Stack-allocated objects do not escape their scope, and lexically scoped regions referenced from the stack can be immediately deallocated when their referencing frame is popped. `StackMonitors` are not initialized and `System.Update` is not invoked at stack finalization.

5.2 Dynamic Class Load

A new class that violates the program's escape semantics is loaded. The language runtime refactors the new class so that its write barrier will register escaping references with a stack frame's `StackMonitor`. The classes that allocate escaping references to the stack are refactored so that they allocate the references to the heap instead, and the new class definitions are loaded. Each thread's current stack pointer is saved in the thread-local `CorruptStack` field.

5.3 Corrupted State

The initialized `StackMonitors` continue tracking all escaping references. When a frame is popped from the call stack, the `CorruptStack` field is assigned `Min(CorruptStack, CurrentFrame)`, i.e., the minimum of the current frame's address and the value of the `CorruptStack`. Frames that are above the `CorruptStack` address do not need to `UnRegister` capturing objects, because the loader has refactored all new objects and methods to conform to the program's new escape semantics, i.e., new objects and methods will not produce escaping references. Note: In the case that refactoring eliminates the possibility that new stack frames

may contain escaping objects, the `x.Frame()` method (Figure 7) should return a `nullFrame` if the frame's address is higher than `CorruptStack`.

5.4 Returning to Steady State

When a thread's `CorruptStack` field reaches the top of the `Immortal` segment of its call stack then there are no more escaping references in that thread. The thread's `StackMonitors` are uninitialized, and the thread can now use the dynamically loaded class as it was originally defined (i.e., the inlined registry tests and calls to `StackMonitor` are removed).

6. OPTIMIZATIONS

Some inferred scoping rules cannot be violated by dynamically loaded classes, e.g., if a `final` method does not allow parameters to escape (Figure 8) then a dynamically loaded class cannot override the inferred scoping rule (Figure 9). Although a dynamically loaded class may redefine the `Sink` class and thereby change the scoping of the `Compare` method, this redefinition will not change the scoping of any references currently on the stack, and there will be no danger of dangling references being created.

```
Class Sink {
    final boolean Compare(Sink b) {
        return (b == this);
    }
}
```

Figure 8

```
Class EscapeSink extends Sink {
    boolean Compare(Sink b) {
        // illegal method override
    }
}
```

Figure 9

The compiler may also be able to determine that the circumstances under which a class is dynamically loaded cannot possibly create dangling references to stack-allocated objects, e.g., if classes are only dynamically loaded from methods that are always within the `Immortal` segment of the call stack then the compiler may infer that all escaping references have already been popped when a class loader is called. In addition, if the runtime system performs a fast escape analysis ([7]) when a class is dynamically loaded, then this will potentially reduce the amount of refactoring that needs to occur and will consequently reduce the number of calls to `System.StackFinalizer(Frame)` (if runtime escape analysis determines that the `Frame` cannot have any

escaping references then the `StackFinalizer` method call can be replaced with a `nop`) and the number of references that need to be registered with the `StackMonitor`. (Ideally, the compiler would save all of its compositional escape analysis information so that the runtime system can perform a more accurate analysis of the dynamically loaded class.)

The runtime system can also use dynamic escape analysis to hasten a thread's return to the steady state. For example, if the runtime system determines that only the escape semantics for method `M` have changed as a result of dynamically loading a class, then for the purposes of returning to a steady state the address of the `Immortal` stack segment can be changed to point to the first instance of method `M` on the call stack, i.e., when the last instance of method `M` is popped from the stack then the thread returns to a steady state. However, this optimization requires stack inspection with cost $O(n)$ where n is the size of the call stack, and so it will probably only be effective if performed in a background thread. The exception is when the compiler has already determined that method `M` is *never* called from a thread, and in that case the thread in question does not enter a corrupted state.

7. CONCLUSIONS AND FUTURE RESEARCH

Escape analysis provides a viable means of safely allocating data to the stack, reducing the load on the garbage collector and thus decreasing latency and increasing throughput. We provide a means of employing speculative escape analysis in an environment supporting dynamic class loading/generation. There is a negligible performance penalty for the case that stack allocated data does not escape its scope. There is an $O(1)$ performance penalty for the case that a dynamically loaded class assigns an escaping reference to a capturing object; if the current frame's `StackMonitor` is already initialized, then this write barrier will have the low cost of an arithmetic comparison and possibly a call to the `LinkedList`'s `add` method. When an escaping reference is popped from the call stack then the object must be reallocated to the heap, after which the objects that have registered with the `StackMonitor` must be updated. In contrast to our approach, Corry ([5]) presents a technique that scans and updates the stack when an object escapes its scope, which eliminates the `StackMonitor` write barrier at the cost of a potentially expensive stack scan. Our intuition is that Corry's method is superior when scope violations are frequent (in which case the write barrier is frequently invoked and the `StackMonitor` will register a considerable number of objects) and that ours is superior when scope violations are rare (in which case the cost of the write barrier is made up for by the amortized evacuation cost). Unfortunately, establishing the values of "frequent"

and “rare” in this context will be difficult due to the dearth of benchmarks that use dynamic class loading / modification; however, we hope to be able to experimentally compare the benefits of optimistic stack allocation with speculative escape analysis (though as noted in Section 5, the program may never return to a steady state in this context).

Although our `StackMonitors` are easily extended to handle lexically scoped region-based memory management ([8]) by treating a region as a stack frame, we plan to investigate whether we can also apply `StackMonitors` to region-based memory management approaches that do not use lexical scopes ([10]).

Escape analysis (and other analyses) also permit the removal of unnecessary synchronization code by detecting thread-local objects ([1], [4]). We would like to extend our runtime refactoring and monitoring technique to handle escaping references that are no longer thread-local due to the semantics of a dynamically loaded class.

8. REFERENCES

- [1] Aldrich, J., Sirer, E.G., Chambers, C., Eggers, S.J. Comprehensive synchronization elimination for Java. *Science of Computer Programming*. Volume 47 (2003), pp. 91-120.
- [2] Blanchet, B. Escape analysis for JavaTM: Theory and practice. *ACM Transactions on Programming Languages and Systems (TOPLAS)*. Volume 25 , Issue 6 (November 2003), pp. 713-775.
- [3] Bollella, G., Brosgol, B., Furr, S., Hardin, D., Dibble, P., Gosling, J., Turnbull, M. *The Real-Time Specification for JavaTM*. Addison Wesley, 2000.
- [4] Choi, J-D, Gupta, M., Serrano, M.J., Sreedhar, V.C., Midkiff, S.P. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*. Volume 25, Issue 6 (November 2003), pp. 876-910.
- [5] Corry, E. 2006. Optimistic stack allocation for java-like languages. In *Proceedings of the 2006 international Symposium on Memory Management* (Ottawa, Ontario, Canada, June 10 - 11, 2006). ISMM '06. ACM Press, New York, NY, 162-173.
- [6] Da Silva, J. and Steffan, J. G. 2006. A probabilistic pointer analysis for speculative optimizations. *SIGPLAN Not.* 41, 11 (Nov. 2006), 416-425.
- [7] Hirzel, M., Dincklage, D., Diwan, A., and Hind, M.. Fast online pointer analysis. Technical report, IBM Research RC23638, June 2005.
- [8] Tofte, M., Talpin, J-P. Region-Based Memory Management. *Information and Computation*. Volume 132, Issue 2 (February 1997), pp. 109 - 176.
- [9] Vivien, F., Rinard, M. Incrementalized pointer and escape analysis. *ACM SIGPLAN Notices*. Volume 36, Issue 5 (May 2001), pp. 35-46.
- [10] Walker, D., Crary, K., Morrisett, G. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems (TOPLAS)*. Volume 22, Issue 4 (July 2000), pp. 701-771.
- [11] Whaley, J., Rinard, M. Compositional Pointer and Escape Analysis for Java Programs. *ACM SIGPLAN Notices*. Volume 34, Issue 10 (October 1999), pp. 187-206.