ESCAPE ANALYSIS FOR MULTI-THREADED JAVALI

Andrei Pârvu, Sebastian Wicki

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

Describe in concise words what you do, why you do it (not necessarily in this order), and the main result. The abstract has to be self-contained and readable for a person in the general area. You should write the abstract last.

1. INTRODUCTION

The basis of this project is an extended version of Javali with support for basic multi-threading, similar to Java's built-in `Thread` class. Additionally, we also added support for synchronization though locks. Having those additional language features also give more margin for potential optimizations, which we will discuss in the following paragraphs.

Motivation. Escape analysis can determine if an object reference escapes a certain dynamic scope, for example the scope of a thread or the scope of a stack frame. This information can be useful for a class of optimizations. In this project, we explored two optimizations: Stack allocation of objects and synchronization elision. Since Javali does not support modules or other forms of dynamic code loading, we can perform a whole-program escape analysis, typically yielding more precision.

In standard Javali, all non-primitive objects are allocated on the heap. The existing implementation even leaks those objects, as there is currently on garbage collection. Using escape analysis, the compiler can infer that certain objects will never outlive the current stack frame, thus those objects could be allocated on the stack. This has the advantage that the object will be cleaned up automatically once the function returns. Additionally, allocating data on the stack is typically faster than calling `malloc()` for heap allocation. This is especially true in multi-threaded programs, as the implementation of `malloc()` internally needs to synchronize as well, although modern implementations such as `TCMalloc` [1] optimize this case heavily.

In this paper we provide an extended escape analysis for stack allocation that even allows stack allocation for objects that are shared with other threads, under the condition that the thread does not outlive the current function. This is often the case in the fork-join programming model, where a function forks multiple child threads to work on in parallel on some data, and then waits for them to join.

As a second optimization, we implement synchronization removal on objects that never escape the current thread, i.e. that are never shared among multiple threads thus do not to be locked in order to be accessed safely.

This optimization is useful for classes that are is written to be thread-safe, i.e. use synchronization to ensure correctness, but are actually never used in a multi-threaded setting, which leads to unnecessary overhead. Library classes such as Java's `Vector` or `Hashtable` class are examples of thread-safe library classes that were later replaced with non-synchronized alternatives (namely `ArrayList` and `HashMap`) for better performance.

Related work. There exist numerous papers on escape analysis for Java. The most cited one, which is also used by the Java HotSpot virtual machine [2], is the work by Choi et al [3]. They model the relationship of object in a so called *connection graph*, where objects are represented as nodes, and directed edges represent the relationship among the objects. Nodes are marked with *ArgEscape* to indicate that they escape its method via arguments, *GlobalEscape* is used for nodes that could be accessed globally or by another thread, and *NoEscape* is used for nodes that do not escape.

A similar graph-based approach was published at the same time in the same journal by Whaley and Rinard [4]. They also use a directed graph, called the points-to graph, to represent objects. The escape information is determined by the type of edges between objects: An object is said to escape its method when it is reachable via an outside edge.

Both approaches are flow-sensitive, they process the nodes of the control flow graph individually and merge the resulting graphs where needed. Both approaches have a intraprocedural phase where each method is processed individually, and a interprocedural phase where the graphs are merged at call sites to yield more precise information.

A survey by [**?**] shows that the approach by Whaley and Rinard is more precise than the algorithm by Choi et all, they are able to stack allocate more objects and remove more synchronization.

Next, you have to give a brief overview of related work.

For a report like this, anywhere between 2 and 8 references. Briefly explain what they do. In the end contrast to what you do to make now precisely clear what your contribution is.

2. BACKGROUND

In this section we briefly present the programming interface and semantics of our threading extension. In the second part, we formally introduce escape analysis.

Threading and Synchronization in Javali. The interface for multi-threading in Javali is heavily inspired by Java. In order to create a new thread, the programmer has to create a new class that inherits from the built-in `Thread` class and overwrite the `run()` method to provide a thread entry point. The `Thread` class contains the following methods:

`run()`	Thread entry point. To be overwritten by clients.
`start()`	Starts a new thread. Like in Java it is an error to call this method more than once on the same receiver.
`join()`	Blocks the caller until the thread exits. It is an error to call this on a thread that was never started or was already joined.

Similar to Java, our extension to Javali has a built-in lock and conditional variable within every object. A notable difference to Java however is that there is no special syntax for locking, the built-in `Object` just provides the following methods.

`lock()`	Locks the receiver object. If the object was already locked by another thread, the current thread is blocked until the object is unlocked. The lock is re-entrant in the sense that a thread can call this method on objects on which it already own the lock.
`unlock()`	Unlocks the object. Also re-entrant in the sense that in order to fully unlock an object, the lock owner needs to call `unlock()` as many times as `lock()` was called. It is an error to call `unlock()` without owning the lock.
`wait()`	Causes the calling thread to wait until another thread calls `notify()` on the object.
`notify()`	Wakes up another thread waiting on this object.

Escape Analysis. For stack allocation we needed to implement an algorithm which can determine if a certain variable (or field) escapes the method in which it is allocated. If the object does not escape, then it can be allocated on the stack. Otherwise, we must distinguish between other cases:

- the object is referenced by a parameter - in this case it might be allocated on the stack of the caller (TODO: we currently don't do this in the code)
- the object is returned - then it might either be allocated on the stack of the caller of freed by the caller
- the object is referenced by 'this'
- the object is passed to another thread

In order to achieve these goals we decided to implement and expand the escape-analysis algorithm presented in [4].

3. PROPOSED METHOD

Points-to-escape graph. The algorithm shown in [4] constructs, for each method, a graph in which nodes represent allocated objects and edges represent references between objects. The nodes with in-degree zero are either locally declared variables, method parameters, or the 'this' object. Edge between nodes can either be labeled with the name of a field, or ... TODO. As described in section 2, each node can have one of the following labels: REF_PARAM, THREAD, ARRAY, THIS and ESCAPED.
An example program is shown in Listing 1. with the associated escape-graph shown in Figure 1. One can observe that the nodes have been classified as following:

- Node 5, corresponding to t.arg has labels THREAD and REF_PARAM, because t is a parameter and also a thread object.
- Node 4, corresponding to e has only label REF_PARAM
- Node 14 and 1 have the THIS label

Listing 1. Example program

```
class Custom extends Thread {
    Box arg;
    void run() {
        write(arg.val); writeln();
    }
}

class Box {
    int val;
}

class Main {
    Box b;
    void escapeOne(Custom t, Box e) {
        write(b.val); writeln();
        t.arg = e;
        t.start();

        t.join();
    }

    void main() {
        Custom t;
        Box b, e;
```

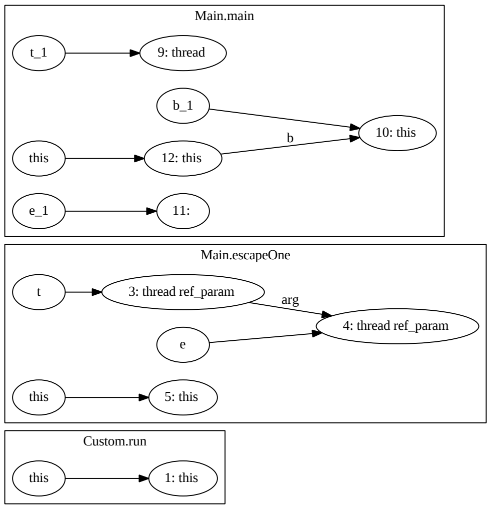

Fig. 1. Points-to-escape graph

```
        b = new Box();
        e = new Box();
        b.val = 23;
        e.val = 42;

        this.b = b;

        t = new Custom();
        escapeOne(t, e);
    }
}
```

Building the graph for a method.. The escape graph is built in an iterative manner using control-flow analysis. First of all each basic block is processed, in a bread-first search manner, and all its instructions are added to a basic-block local escape-graph. Secondly, the escape graph of each basic block is merged with the escape graphs of its predecessors and the process is repeated until no more changes occur. Finally, after the graph is completed, a depth-first search is applied on it, in order to spread the labels of the nodes to the set of nodes it references.

Analyzing a method call.. The first approach we had to analyzing method calls was to map the parameters of the given call to the corresponding nodes in the escape graph of the callee and copy the already computed labels from the callee to the caller. Unfortunately, this solution misses cases in which different fields of parameters are assigned to each other in the callee (one example is shown in Listing 2). In order to address this problem we decided to recurse into the callee and construct it's escape graph given the parameter nodes of the caller. Although this increases the overall complexity of our solution, it provides a more accurate analysis.

Listing 2. Field parameter assignment

```
void main() {
  F f1, f2;
  f1 = new F();
  f1.next = new F();

  f2 = new F();
  f2.next = new F();

  d1(f1, f2);

  this.f = f1;
}

void d1(F f1, F f2) {
  f1.next = f2.next;
}
```

4. EXPERIMENTAL RESULTS

Experimental setup. For the experiments we used two different machines. The first one, running Ubuntu 14.04 with a i5-3317U, 1.70GHz processor and 4Gb RAM.

We tried to analyze three different aspects of our project: firstly, the running time improvement of just allocating the unescaped objects on the stack. Secondly, we wanted to see the time benefit of working with stack-allocated objects instead of heap-allocated ones. And thirdly, we tried to observe the how lock-removal affects single-threaded programs.

Results. For the first experiment, we just allocated various objects without using them afterwards, in order to see how the running time differs for both cases. TODO: run the actual experiment :)

In the second experiment, we wrote a basic matrix multiplication program with the intent of observing how execution time changes when using stack allocation. The results can be seen in Figure 2. It can be observed that the stack allocation improves running time with almost 20%, with the obvious draw-back that more stack memory is used.

For the third experiment, we implemented a hash map, which simulates string hashing (because Javali doesn't have strings we used ASCII arrays). The hash map is implemented to be thread-safe, so we ran the experiment with one thread and lock removal. Results can be seen in Figure 3; with lock removal, running time is approximately 10% faster than the unanalyzed version.

5. CONCLUSIONS

Here you need to summarize what you did and why this is important. *Do not take the abstract* and put it in the past tense. Remember, now the reader has (hopefully) read the report, so it is a very different situation from the abstract. Try to highlight important results and say the things you really want to get across such as high-level statements (e.g.,

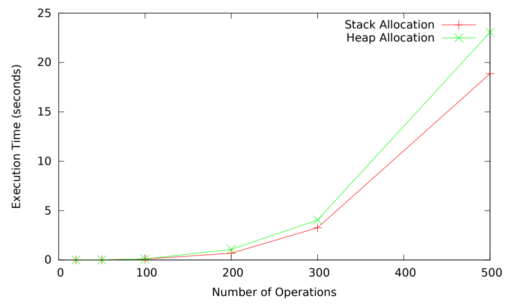

Fig. 2. Running times matrix multiplication

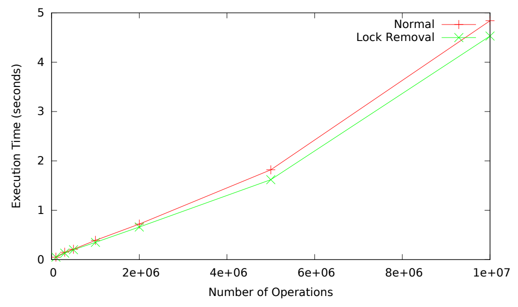

Fig. 3. Running times hash map

we believe that .... is the right approach to .... Even though we only considered x, the .... technique should be applicable ....) You can also formulate next steps if you want. Be brief. After the conclusions there are only the references.

6. REFERENCES

[1] Sanjay Ghemawat and Paul Menage, "TC-Malloc: Thread-caching malloc," `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`, 2009, Accessed: 2015-06-02.

[2] "Java HotSpot^{TM} virtual machine performance enhancements," `https://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html#escapeAnalysis`, Accessed: 2015-06-02.

[3] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C Sreedhar, and Sam Midkiff, "Escape analysis for Java," *Acm Sigplan Notices*, vol. 34, no. 10, pp. 1–19, 1999.

[4] John Whaley and Martin Rinard, "Compositional pointer and escape analysis for java programs," *ACM Sigplan Notices*, vol. 34, no. 10, pp. 187–206, 1999.

[5] Andrew C King, *Removing Garbage Collector Synchronisation*, Ph.D. thesis, University of Kent at Canterbury, 2004.