DuckTracks: Path-based Object Allocation Tracking

[Work in Progress]

Stefan Fitzek[1] Philipp Lengauer[2] Hanspeter Mössenböck[2]

[1]Christian Doppler Laboratory MEVSS
Johannes Kepler University Linz, Austria
stefan.fitzek@jku.at

[2]Institute for System Software
Johannes Kepler University Linz, Austria
philipp.lengauer@jku.at

ABSTRACT

Efficiently tracking an application's object allocations is of interest for areas such as memory leak detection or memory usage optimization. The state-of-the-art approach of instrumenting every allocation site with a counter introduces considerable overhead. This makes allocation tracking in a production environment unattractive. Our approach reduces this overhead by instrumenting control flow paths instead of allocation sites and dynamically determining the hot path through a method. Our ultimate goal is to reduce the amount of required counter increments by such a degree that using it in production environments becomes feasible. We present an implementation of our approach for the Java HotSpot Virtual Machine. First measurements already show a reduction of required increments of up to 30% compared to the state of the art.

Keywords

Allocation Tracking; Allocation Site; Hot Path; Memory Monitoring; Path; Control Flow

1. INTRODUCTION

Detecting and localizing memory leaks, reducing the amount of required garbage collections, or identifying possible code optimizations regarding memory behavior count among the areas that require, or benefit from, allocation tracking. The general approach used by state-of-the-art allocation trackers is to instrument every allocation site with a counter. The drawback of this approach is that the counter overheads ruin the advantage of Java's fast allocations. Although it provides accurate results, the introduced run-time overhead makes this approach infeasible in a production environment. State-of-the-art memory monitoring software such as ElephantTracks sometimes introduces run-time overhead exceeding 1000% [7], although it has to be said that they track some additional events like object deaths, too.

We introduce DuckTracks, an allocation tracking approach that uses path information to reduce the amount of counter increments, and, therefore, the introduced overhead required to track allocations. We define a path as a sequence of connected blocks in the control flow graph of a method. All blocks and edges in the control flow graph belong to exactly one path.

Every method usually has multiple paths which the control flow can take during execution. One of these paths will be taken more often than the others. We call this the *hot path*. By identifying this path and by instrumenting it with a counter representing its allocations we can track the allocations of a method with just a single counter increment in most cases. To correctly track allocations during executions that diverge from the hot path, we have to instrument all diverging paths as well. Their counters may add additional allocations or remove allocations which were already counted, but will not happen due to the control flow diverging from the expected path.

Correctly identifying the hot path is best done by analyzing the path usage of a method while it is executed, which can be done by gathering a path profile during a data gathering run and then basing the instrumentation on this profile during the actual run. This, however, assumes that the data gathering run is representative for the actual method usage and would require tailoring data gathering runs to profiled applications. We therefore chose a more dynamic approach. By initially choosing paths based on heuristics, DuckTracks gathers hot path information during the actual run. It then reinstruments methods based on this information, if necessary.

To the best of our knowledge, DuckTracks is the first monitoring tool that tracks allocations pathwise in order to reduce the amount of required counter increments. Furthermore, these paths are not statically determined and fixed, but dynamically adjusted based on data derived from our allocation counters. This enables DuckTracks to adapt itself to the behavior of the monitored application.

2. APPROACH

Our approach dynamically instruments the methods of every class at load time, using functionality provided by the Java Virtual Machine Tool Interface (*JVMTI*) and the Java Native Interface (*JNI*). The instrumentation process is divided into six steps, as shown in Figure 1.

The first step is acquiring the method's bytecode which is trivial due to existing JVMTI functionality. We then construct the bytecode's control flow graph.

ICPE'17, April 22-26, 2017, L'Aquila, Italy

DOI: http://dx.doi.org/10.1145/3030207.3030235

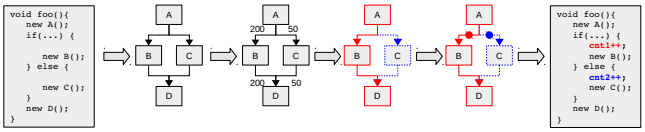

Figure 1: From original to instrumented bytecode

In the next step, we mark every edge of this control flow graph with its execution frequency. Naturally, when a class is initially loaded no edge frequency data exists. Therefore, we additionally instrument every method with an entry counter. These entry counters trigger a reload of the class after a certain number of method calls, which allows us to reinstrument its methods. During reinstrumentation we have access to the counter values of the previous instrumentation, which gives us the traversal frequencies of all instrumented edges. We then propagate these values through the control flow graph. This propagation is currently rather rudimentary, and improving it will be part of our future work.

The frequency-marked control flow graph is then used to create the paths through the method. We continue by determining the ideal counter position for each of these paths and then calculate the counter semantics, i.e., which allocations are measured by a specific counter and which other allocations have to be corrected. Finally, the method's bytecode is instrumented with these counters and the instrumented class is loaded.

Although constructing the control flow graph and instrumenting the bytecode are comprehensive topics by themselves, the path building and counter calculation steps constitute the core of our contributions. We will therefore cover these two aspects in more detail, starting with how paths are built.

2.1 Path Building

We define a path as a sequence of connected blocks in the control flow graph of a method. The left side of Figure 2 shows a trivial method with two such paths and purely sequential control flow. A path can either start at a block without predecessors, such as the continuous (red) path at the method entry block A, or by branching off from another path, as the dashed (blue) path. The block from which a path branches off is called its *branching block*, e.g., block A is the dashed path's *branching block*. Similarly, a path can end by leaving the method, indicated by ending with a block without successors, such as block E, or by joining another path, i.e., the path's last block is connected to a block that belongs to another path. The block at which the path joins is called its *joining block*, e.g., block E is the dashed path's *joining block*.

2.1.1 Handling Sequential Paths

Path building starts with the hot path, which starts at the method entry block. We build it by incrementally appending the most likely successor of the current tail block. If the control flow graph is not frequency marked, e.g., when the class is initially loaded, the most likely successor is picked based on static heuristics. Otherwise, we pick the successor with the highest edge frequency. The picked successor becomes the path's new tail block. This process repeats until we reach a method exit, which ends the path.

The resulting path may contain several *branching points*, i.e., blocks with more than one successor where additional paths branch off. We now create each of these side paths. Unlike the initial path, these paths may select successors that already belong to other paths. This leads to a join: The path's *joining block* is set accordingly, which ends the path. The result of this process can be seen on the left side of Figure 2, with the dashed path branching off from block A and rejoining at block E.

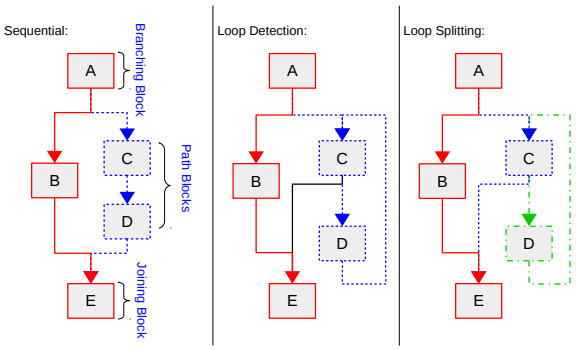

Figure 2: Sequential and looping path structure

2.1.2 Handling Loops

Loops are detected during path building when a path under construction, called the *current path*, tries to append a block that is already part of this path. We call this block the *header block*. Figure 2 shows this situation in the middle graph: the dashed (blue) path tries to append block C as the successor of block D, but block C is already part of the dashed path. Therefore, block C is identified as the *header block* of a loop.

We handle loops by splitting off the path segment after the *header block* and turning it into a separate path. The resulting path's *branching block* and its *joining block* are identical, i.e., the *header block*. This has been done in the rightmost graph of Figure 2, resulting in the dashed-dotted (green) path which branches off from block C. Construction of the dashed path then continues at the *header block* by picking another successor, which leads to the dashed path joining the continuous (red) path at block E in our example.

If the *header block* has no other successors, then the *current path* ends with the *header block*. As a rule, the *current path* cannot join any loop path that has been split off from the *current path* itself. After the *current path* has been constructed, the *branching points* in any of the split off loop paths are handled in addition to the path's own *branching points*.

No further knowledge of loop structure or specialized handling is required. Loops are therefore simply paths that branch off from some block and rejoin at the same block.

2.2 Counter Construction

We now construct a counter for each path. These counters are defined by two attributes: their location and their semantics, i.e., the set of allocation sites that are affected by a specific counter. A counter's semantics is influenced by its own location as well as by the locations of other counters. We therefore determine the locations of all counters before calculating their semantics.

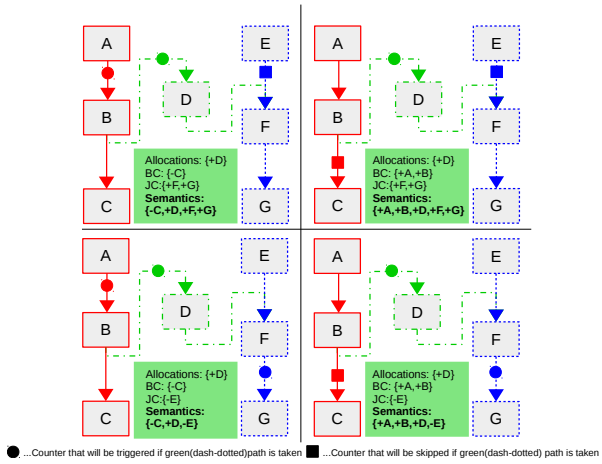

Figure 3: Counter semantics as dictated by location

2.2.1 Counter Locations

Counters are placed on edges in the control flow graph. We want to place counters such that as few of them as possible are triggered during any given method execution. To achieve this, we introduce the terms *flow* and *share count*. A *flow* is a sequence of blocks from the method entry to the method exit, i.e., a complete control flow through the method during a particular execution. The *share count* of an edge specifies how many *flows* share this edge. Since every edge is owned by exactly one path, the *share count* is at least one. It is increased by every path that branches off after the edge's end block and every path that joins before the edge's start block.

The chance of incrementing multiple counters during a method execution rises for every counter placed on an edge with a *share count* greater than one. We therefore aim to place counters at the least shared edge of every path.

2.2.2 Counter Semantics

A counter's semantics is the set of allocation sites that the counter represents. It consists of three components: the *allocations*, the *branching correction* and the *joining correction*.

A path's *allocations* represent the allocation sites of the path itself and are therefore the set of all allocation sites in its blocks. The *branching correction* and the *joining correction* represent allocations before and after the path. Their values depend on the counter placement in the path from which the *current path* branches off and the counter placement in the path that the *current path* joins. Figure 3 shows the four possible counter placement scenarios and their effect on the *branching correction* and the *joining correction*. In this example, we are interested in the value of the dash-dotted (green) path, which branches off from the continuous (red) path and joins the dashed (blue) path.

If the continuous path's counter is placed before the *branching block B*, it will have been incremented already when the dash-dotted path is entered. The dash-dotted path therefore requires a *branching correction* (BC) that subtracts the allocations on the continuous path after the *branching block B* since they have been counted but will not be traversed. This is the case in the two graphs on the left side of Figure 3.

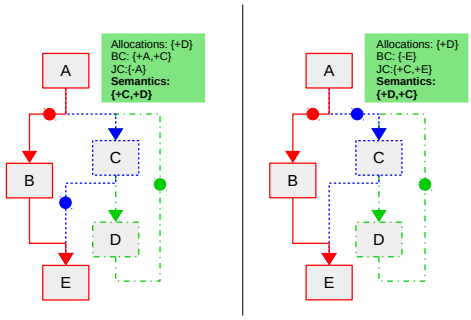

Figure 4: Loop counter semantics as dictated by location

On the other hand, if the continuous counter is placed after the *branching block B*, it will not have been incremented yet when the dash-dotted path is entered. The dash-dotted path's *branching correction* therefore needs to add all allocations that happen before and in the *branching block B* since they have been traversed but were not counted. This can be seen in the two graphs on the right side of Figure 3.

The same principle applies when calculating the *joining correction* (JC) of the dash-dotted path. If the dashed path's counter will be skipped, i.e., if it is placed before the *joining block F* the *joining correction* needs to add all allocations of the dashed path after and in the *joining block F*. This case holds in the two upper graphs in Figure 3. If the dashed path's counter will be incremented, i.e., if it is placed after the *joining block F*, the *joining correction* needs to subtract all allocations of the dashed path that happen before the *joining block F*. This is the case in the two lower graphs in Figure 3.

This example assumes, for simplicity's sake, that both the continuous path and the dashed path do not branch off from, or join, another path. If, for example, the dashed or continuous path were to join a fourth path, then this fourth path would also affect the dash-dotted path's corrections, as would every other path in this join chain until a path is reached that exits the method. The dashed and continuous paths' branching chains affect the dash-dotted path's corrections in the same way. This will be explained in more detail in future work.

The actual counter semantics is the sum of these three components. Consequently, parts of the *branching correction* and the *joining correction* may cancel each other out. Loops are a good example of this. Since a loop's *branching block* and *joining block* are the same, namely the *header block*, both corrections will be calculated based on the same path segment. The result will be that the parts of the corrections that handle blocks before and after the *header block* cancel each other out, leaving only the allocations of the *header block* itself. The resulting counter semantics for the loop will therefore include the loop's *allocations* and the allocations of the *header block*, which together represents a single loop iteration. This is shown in Figure 4: the placement of the dashed (blue) path's counter changes the dash-dotted (green) path's *branch correction* and *join correction*, but the resulting counter semantics remains the same.

When this step is completed, we know the location and semantics of every path's counter. A path's counter semantics may be empty, meaning that traversing the path has no

effect on the allocation total. We eliminate such counters.

After placing the counters into the bytecode, they will accurately count the number of allocations caused by every method execution. Since we identified the hot path and constructed the counters based on it, only a single counter increment will be needed most of the time.

3. FIRST RESULTS

We tested our prototype implementation on various benchmarks of the DaCapo, DaCapo Scala and SPECjvm suites. These preliminary tests showed a reduction of required counter increments of up to 30% compared to counting allocations individually. We measured a run-time overhead of 20-30%. Section 5 will present some of our ideas to further improve these first results.

4. RELATED WORK

There is ample work on tracking allocations, resulting in varying levels of overhead and precision. Some approaches, such as AntTracks [6], modify a VM and thus benefit from having full access to VM functionality. This allows for low overhead as well as for tracking events that cannot be accessed from outside the VM. The price for this level of access is a lack of portability due to the required VM modifications.

Agent-based approaches such as ElephantTracks [7] implement tracking through instrumentation. These approaches cause significantly more overhead and have less access to VM events. They are, however, not restricted to a single VM. DuckTracks falls into this category and attempts to reduce the overhead penalty to a level that makes its usage feasible in a production environment.

Creating and utilizing path information is a well-researched area, too. The seminal work of Ball and Larus [1, 2] describes an algorithm for creating path traces in purely sequential programs. It has been built upon in a variety of subsequent work, including additions for loop handling such as D'Elia et al. [4], or selective profiling such as Vaswanie et al. [8]. The main difference between the Ball/Larus algorithm and our approach is that they consider paths as complete sequences of blocks from method entry to method exit, which requires counters that ensure that every such sequence is correctly tracked. Since we are only interested in tracking allocations, we can omit all counters for path sequences that do not change the number of allocated objects. Also, their counter placement is optimized to reduce the number of placed counters while still delivering a complete path trace. The goal of our counter placement is to minimize the number of counter increments during execution.

Hot path information is extensively used in areas such as compilation and code optimization, as shown in the work of, e.g., Bebenita et.al [3] or Gupta et al. [5]. However, to the best of our knowledge, we are the first to use hot path information to achieve more efficient allocation tracking.

5. FUTURE WORK

We currently work on improving edge frequency propagation by exploiting dominance relationships between blocks as well as marking edges with frequency ranges if precise marking is not possible. Since it would by trivial to extend DuckTracks to allow tracking of other attributes such as method calls, eventually adding such attributes remains a possibility for future work. A more immediate improvement is the addition of counter inlining. This involves removing the hot path counter from a method and adding its semantics to the counters of every calling method. This would reduce the amount of required counter increments even further. We also plan to handle exceptions by instrumenting catch blocks and using the stack trace to correct allocations.

6. CONCLUSIONS

We improved upon the state-of-the-art approach for allocation tracking by tracking per path instead of per allocation site. We further refined our approach by constructing paths according to their execution probabilities, thereby using hot path information to great effect.

Preliminary results already show that our approach reduces the amount of necessary increments by up to 30% compared to counting allocations individually, with further optimizations under way.

Reducing the amount of required increments brings us one step closer to realizing an allocation tracker that is suited for production use. This is achieved without the need for extra runs to gather profiling information due to DuckTrack's self-optimizing, adaptive nature.

7. ACKNOWLEDGMENTS

This work was supported by the Christian Doppler Forschungsgesellschaft and by Dynatrace Austria.

8. REFERENCES

[1] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.*, 16(4), July 1994.

[2] T. Ball and J. R. Larus. Efficient path profiling. In *Proc. of the 29th Annual ACM/IEEE Int'l. Symposium on Microarchitecture*, Washington, DC, USA, 1996.

[3] M. Bebenita, M. Chang, G. Wagner, A. Gal, C. Wimmer, and M. Franz. Trace-based compilation in execution environments without interpreters. In *Proc. of the 8th Int'l. Conf. on the Principles and Practice of Programming in Java*, New York, NY, USA, 2010.

[4] D. C. D'Elia and C. Demetrescu. Ball-larus path profiling across multiple loop iterations. In *Proc. of the 2013 ACM SIGPLAN Int'l. Conf. on Object Oriented Programming Systems Languages & Applications*, New York, NY, USA, 2013.

[5] R. Gupta, D. A. Berson, and J. Z. Fang. Resource-sensitive profile-directed data flow analysis for code optimization. In *Proc. of the 30th Annual ACM/IEEE Int'l. Symp. on Microarchitecture*, Washington, DC, USA, 1997.

[6] P. Lengauer, V. Bitto, and H. Mössenböck. Accurate and efficient object tracing for java applications. In *Proc. of the 6th ACM/SPEC Int'l. Conf. on Performance Engineering*, New York, NY, USA, 2015.

[7] N. P. Ricci, S. Z. Guyer, and J. E. B. Moss. Elephant tracks: Portable production of complete and precise gc traces. In *Proc. of the 2013 Int'l. Symp. on Memory Management*, New York, NY, USA, 2013.

[8] K. Vaswani, A. V. Nori, and T. M. Chilimbi. Preferential path profiling: Compactly numbering interesting paths. In *Proc. of the 34th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, New York, NY, USA, 2007.