CS 6004: PA4 Report

Shiv Kiran - 200050019
Karthikeya - 200050084

Spring '24

Contents

1 Intro

For the sake of simplicity we've assumed the following about the input programs

- Control flow graph is a DAG
- Constructors are simple and are only intended for object creation (i.e. no object escapes it, it has no side effects and virtual invokes). Thus are not transformed in any way.
- No try-catch blocks

Here's our github repo. To reproduce the results, first copy our `BytecodeInterpreter.hpp` (present at the top level dir) to appropriate location in openj9 dir, and then run `./run.sh`. This scripts runs soot on all the testcases with and without our optimization and stores class files in `./build` dir and jimple files in `./jimpleOutput` dir. It also compares the application output with the expected ones and reports if there's any mismatch - which would indicate that our transformation is semantically changing the program somehow.

2 Transformations performed

2.1 Method Inlining at Monomorphic callsites

We used Soot's Variable Type Analysis (VTA) to construct the call graph. Using this CG, we identify monomorphic virtual-callsites (i.e virtual invokes) and inline appropriate methods. During method inlining, we utilize its transformed body rather than the original one. Note that we process methods in cg-topologically sorted order. We recognize that this approach may not be scalable due to the potential exponential increase in code size resulting from nested inlining. However, our primary objective remains to minimize runtime as much as possible.

2.2 Scalar-replacement

We do this replacement after method-inlining because we might be able to scalar-replace more objects. Conditions for an object to be scalar replaceable are: The object doesn't escape and there are no units in which you need the object to be present in its entirety. We've used PA2's intra-procedural escape analysis to determine the escape status of the objects. For all expressions like below, we mark all the reachable concrete-objects from `a` to be non-scalar-replaceable.

- `a instance of T`
- `a == b`
- `a == null`

- `(T) a`
- `foo(a)`

3 Performance Stats

3.1 Testcases

We've written two testcases `./allTCs/heavy.java` and `/allTCs/sanityChecks.java`.

In the first one we wrote a loop which runs for many iterations and makes some virtual calls and heap allocations (new calls) in every iteration. This testcase is mainly to demonstrate that we are able to get the execution time down by a substantial amount (exact numbers are show in the next sub-sections).
In the second testcase, we've included some corner cases to demonstrate the correctness of our transformation. Like, inlining functions with multiple return statements, complex branching etc. and different cases where scalar replacement isn't possible.

3.2 Virtual Invoke Calls

Results for `sanityCheck.java`

With Optimization:

- Test.Main: 22

Without Optimization:

- Test.Main: 6
- Test.sanityCheckVI: 23
- Test.sanityCheckSR: 11

Results for n = 100 in `heavy.java`

With Optimization:

- Test.main: 4

Without Optimization:

- Test.main: 8
- Test.heavySRs: 202
- Test.heavyVIs: 201

Results for n = 1000 in `heavy.java`

With Optimization:

- Test.main: 4

Without Optimization:

- Test.main: 8
- Test.heavySRs: 2002
- Test.heavyVIs: 2001

3.3 `new()` Calls

For sanityChecks.java

- Number of new calls with optimization: 8
- Number of new calls without optimization: 17

For heavy.java

- n = 100. With optimization = 2, Without optimization = 107
- n = 1000. With optimization = 2, Without optimization = 1007

3.4 Time Analysis

Table 1: Timing Results for Various no. of Iterations in heavy.java

Iterations	With Optimization		Without Optimization	
	User	Sys	User	Sys
1000	0.050s	0.015s	0.035s	0.032s
10000	0.056s	0.016s	0.062s	0.014s
100000	0.065s	0.025s	0.096s	0.008s
1000000	0.146s	0.018s	0.281s	0.031s
10000000	0.844s	0.056s	1.840s	0.017s
100000000	9.943s	0.044s	18.122s	0.060s