A System for Runtime Loop Optimisation in the Jikes RVM

Jisheng Zhao, Dr Ian Rogers, Dr Chris Kirkham
Department of Computer Science, University of Manchester

Key words to describe the work:

JVM, Dynamic Compiler Optimisation, Runtime Check Elimination, Loop Unrolling.

Key Results:

The creation of a new loop analysis phase for the Jikes RVM and three new optimizing compilation phases. Between 0.25% (overall) and 3% (best-case) speed-up of the SPECJVM Client 98 benchmark. An improved array-bound and null check elimination phase that eliminates a greater number of runtime checks than existing analyses. Two new loop optimisation phases that achieve loop unrolling and eliminate redundant branches for FOR loops.

How does the work advance the state-of-the-art?:

The Jikes RVM has a redundant check and loop unrolling optimisation system. The redundant check system looks for known to be true checks and eliminates them. The loop unrolling optimisation replicates loop bodies and their contents. Our new optimisations use loop analysis and explicit checks to remove a greater number of array-bound and null checks through explicit testing, and to unroll loops without replicated test and branches. Our results show the efficacy of runtime loop analysis and runtime checks to improve loop optimisations, and to remove array-bound and null checks.

Motivation (problems addressed):

Dynamic optimising compilers must trade the time spent optimising code with time taken away from running it. With Java the implicit throwing of exceptions can cause poor runtime performance. Standard loop optimisations can also greatly improve performance. A loop analysis phase is desirable in an optimising compiler to generate a number of optimisations, however, the efficacy of such an approach at runtime isn't clear given the potentially large computational cost. This work demonstrates the efficacy of the loop analysis approach, which will be built upon in the future to deliver a range of loop optimisations.

Introduction

The Jikes RVM [1] is a Java Virtual Machine written in the Java programming language. It has two compilers: a baseline compiler that translates the Java bytecode directly into machine code, and an optimizing compiler that translates to machine code going via an intermediate representation and a number of compiler optimisation phases. An adaptive framework uses runtime profiling data to recompile the bytecode using the optimizing compiler to improve the running codes performance. ie.

The Jikes RVM's intermediate representation (IR) is made up of instructions with operands, contained within basic blocks that are part of a control-flow graph. Useful properties of the IR include it being mutable (reducing the cost of modifying instructions), having an initially unlimited number of temporary registers (these are removed as the code is prepared for the target architecture) and for removing the inefficiencies of the stack-based Java bytecode. The Jikes RVM has a static single assignment (SSA) form [3] built upon its core IR. A program is in SSA form if each of its variables has exactly one definition, thus making the IR only contain true data dependencies. The Jikes RVM also captures important control-flow, exception and heap accesses in its IR using guard, exception and heap operands. These simplify analyses of the IR and allow it to be modified in such a way that, as long as the appropriate dependency information is maintained, will be safe for future optimisation phases.

Annotated loop analysis

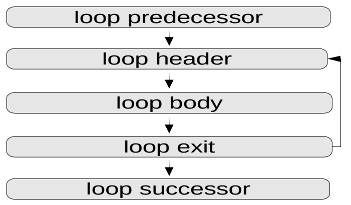

Figure 1 Loop format

We build upon the SSA form by creating an additional representation for loops. Loops are represented in a tree, where loops contained within other loops are represented as sub-trees within the tree. Loops must conform to the structure shown in Figure 1; however, the loop header, body and exit may be the same basic block. Annotations are added to the loop that determine the structure and initial, terminal values of the loop iterator. A loop that has the correct properties we call affine if the initial and terminal values are loop invariant and the instruction to modify the iterator uses a loop invariant stride value. If these values are constant then we call the loop constant.

Array-Bound and Null Check Elimination

Existing analyses to remove array-bound and null checks have relied on adding pi instructions to the IR in an attempt to propagate known test results and eliminate redundant tests [2]. We complement this approach with the addition of a new phase. Our new phase starts at the leaves of the loop tree and creates two identical loops, for that node, with and without checks. To determine the loop to execute at runtime, a dynamic check is created to check that the null and array-bound checks will never be executed. The pi nodes can eliminate redundant checks, and our approach works in all cases whereas the current ABCD algorithm has limitations.

Constant Loop Unrolling

In the case of a constant loop, a loop can be perfectly unrolled, removing all branches and replicating the body of the loop. We determine whether to perform constant loop unrolling by looking at the size of the loop and the number of iterations it executes for. Creating a large number of loop bodies can significantly slow the optimising compiler, so we unroll these loops with a more general approach.

Affine Loop Unrolling

Current loop unrolling in the Jikes RVM replicates loop bodies and doesn't modify tests and the use of iterator values. We improve upon this by creating a loop unrolled a number of times (currently set at 4) and independent loop bodies to take up any spare iterations. We show this in Figure 2.

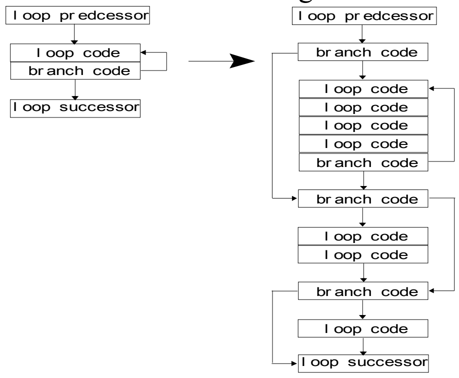

Figure 2 Affine loop unrolling

Result and Discussion

Overall the three optimization phases improve the geometric mean runtime performance of Jikes RVM by 0.25% when running the SPECJVM Client 98 benchmark. Figure 3 shows a break down of the improvements for individual benchmarks.

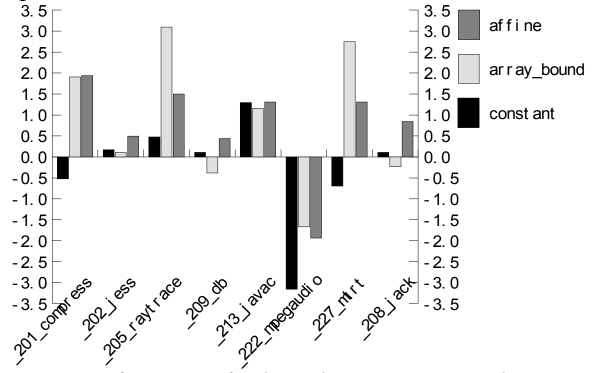

Figure 3 Performance of independent optimisation phases

The results show the optimisations can gain upto a 3% improvement in performance on individual benchmarks. If mpegaudio is ignored the overall improvement from the optimisation phases is 1.195%.

The performance of the optimisations is dependent on applying them when necessary and ensuring the extra compilation overhead is regained. Using profile information we hope to better guide when to perform optimisation. We believe we can also better guide our optimisations by dynamically modifying thresholds for their application. We believe further work is needed to extend the current adaptive system to tackle the complex dynamic optimisation problem.

Summary

We have introduced a new system for loop optimisation performed at runtime within the Jikes RVM. We have demonstrated their performance improvement on the SPECJVM Client 98. There is significant work to be done on the optimisation system structure and on new loop optimisations.

References

[1] The Jalapeño Dynamic Optimizing Compiler for Java, Mical Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarker, Mauricio Serrano, V.C.Sreedhar, and Harini Srinivasan. *1999 ACM Java Grande Conference*, San Francisco, June 12-14, 1999.

[2] ABCD: Eliminating Array Bounds Check on Demand Rastislay Bodik, Rajiv Gupta and Vivek Sarkar. *ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI 2000)*, Vancouver, British Columbia, Canada, June 17-21, 2000.

[3] Advanced Compiler Design Implementation Steven S. Muchnick. Morgan Kaufmann Publishers, 1997.