Automatic Generation of Peephole Superoptimizers

Sorav Bansal and Alex Aiken

Computer Systems Lab, Stanford University

ASPLOS '06

Introduction

- Peephole optimizers seek to use pattern matching that replace one instruction sequence by another equivalent sequence that is faster.
- This can be achieved by removing redundant *load* and *store* instructions, or optimizing memory utilisation.
- This paper presents a new automatic and systematic approach to building a superoptimizer, which uses offline computation to build rules for instruction replacement.
- The model closely resembles a search engine as far as the storage and retrieval of optimising sequences is concerned.
- A *fingerprinting* mechanism is used to provide each target instruction sequence a unique hash value. Post this step, an equivalence test determines if the sequence may be replaced or not.
- The analysis seeks to demonstrate that limited resources can be used to learn a large number of optimizations, and many such optimizations are an improvement over existing compilers.

Superoptimization: An insight

- Superoptimization involves finding the optimal code sequence for a single sequence if *assembly* instructions that are *loop free*.
- Existing compilers do not usually produce the most optimal sequence: if they do, it is a lucky accident!
- *Massalin et al* explored an approach to superoptimization earlier that found the lowest cost optimal sequence. This used enumeration and equality testing.
- *Denali et al* used transformations to find a set of all possible equivalent sequence. The lowest cost one is then chosen.
- This paper prunes the search space using proprietary techniques that preserve generality of various instruction sequences.

Salient Features and Contributions

- This paper involves a offline generation of the superoptimized versions of many target sequences, by *harvesting* them from a set of training programs.
- To prevent loss of generality, almost 300+ opcodes of the x86 architecture are handled.
- This paper introduces *canonicalization*, a register renaming technique that groups instruction sequences into equivalence classes based on the order of registers they use.
- The output is a set of replacement rules, mapping canonical instruction sequences to their canonical output values.
- These rules may be less generic than hand written rules, but multiple such rules applied together may be able to achieve the same effect.

Some useful Definitions

- A *cost function* is a function capturing the approximate cost of an instruction sequence on a given processor. They may be based on running time or instruction count.
- An *optimal* instruction sequence is one for which no instruction set of lesser cost exists.
- The *equivalence* of two instructions is measured under a context, that involves considering the registers, stack and memory status.
- The algorithm conservatively assumes that all memory and stack locations are live and ignores I/O instructions.
- A *equivalence test* tests two sequences for equality under a context L of live registers. $O \cong_L T$ denotes that under the context L, the two sequences O and L are equivalent.

The Design Approach

- A harvester extracts useful instruction sequences from the training program set.
- The *enumerator* exhaustively enumerates all sequences of a certain length, to see if any one of these sequences can be a suitable optimal replacement for the target sequence.
- The optimizer applies optimizations from a database to the appropriate applications.
- Harvested instruction sequences are assumed to have a single entry point, and must not contain any jump targets of other instructions.
- A harvested sequence may contain jump instructions to other targets, and has multiple exit points.
- The harvester captures the state information at the end of each sequence.

Canonicalization and Fingerprinting

- A canonical sequence of instructions is one that has registers and constants named in the order of their appearance, hat is, in alphabetical order.
- To optimise an instruction sequence, we first canonicalize it, and then search the database for an equivalent optimising sequence. Once found, we restore the original register names.
- The fingerprint of an instruction sequence is its unique index into the hashtable.
- It is computed using *testvectors*, by converting it into binary code and capturing the machine state post execution.
- This method is extremely fast and correctly simulates behavior.
- *Sandboxing* of memory accesses is done by bounding memory accesses to a small memory of size 256 bytes.

The Enumerator

- Enumerated instructions are constrained to have only a certain maximum number of distinct registers and constants, and at most two exit points.
- Direct memory accesses are done via two locations $\mathrm{C0}$ and $\mathrm{C1}$.
- If we use a simple cost function, we observe that all subsequences of a given instruction set must be optimal themselves. Thus, the search space can be pruned by replacing larger instructions with a combination of smaller equivalent ones.
- The fingerprint is computed for the set of live registers, and if a match is found, we proceed to the equivalence test.

Learning an Optimization

- The equivalence test is conducted via two mechanisms, a boolean test and an execution test.
- The execution test involves running the code on test vectors and comparing the machine state post execution.
- A number of pairs of instruction sequences pass this test but fail the boolean test, possibly due to loss of bits during computation. This test does not cover aliasing as well.
- The boolean test involves imposing logical constraints and utilising a SAT solver.
- These constraints record the equivalence of read and write addresses to store aliasing information. Where unavailable, we conservatively assume aliasing.
- The optimization database stores only canonical instructions.

Experimental Results

- The tests were run on a Linux having a Pentium 3.0 Ghz processor and 100 GB local storage.
- The peephole size of instructions was chosen to be 3, with windows of up to length 6 instruction sequences in the experiments.
- A good speed up was observed for kernels on integer arrays.
- When run on SPEC CINT2000 benchmarks, it was observed that the speed up was significant for compute intensive kernels rather than full scale applications.
- A total of 3000 codesize and 2100 runtime optimizations were learnt by the optimizer post training.

Summary and Future Work

- The paper presents a technique to learn optimizations offline from a set of training programs and store them in a database.
- By canonicalizing instructions and grouping them into an equivalence class, the size of the database is significantly reduced.
- Fingerprinting and equivalence checks help retrieve an instruction sequence from the database for replacement.
- Understanding loop-carried dependencies by unrolling loops can be one area of future work.
- Exploring efficient ways to carry out goal directed search (which can detect longer instruction sequences) can be taken up.

Thank You!