Automatic Generation of Peephole Superoptimizers

Sorav Bansal and Alex Aiken

Computer Systems Lab
Stanford University
{sbansal, aiken}@cs.stanford.edu

Abstract

Peephole optimizers are typically constructed using human-written pattern matching rules, an approach that requires expertise and time, as well as being less than systematic at exploiting all opportunities for optimization. We explore fully automatic construction of peephole optimizers using brute force superoptimization. While the optimizations discovered by our automatic system may be less general than human-written counterparts, our approach has the potential to automatically learn a database of thousands to millions of optimizations, in contrast to the hundreds found in current peephole optimizers. We show experimentally that our optimizer is able to exploit performance opportunities not found by existing compilers; in particular, we show speedups from 1.7 to a factor of 10 on some compute intensive kernels over a conventional optimizing compiler.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors

General Terms Algorithms, Performance, Design, Reliability

Keywords Superoptimization, Peephole Optimization, Code Selection

1. Introduction

Peephole optimizers are pattern matching systems that replace one sequence of instructions by another equivalent, but faster, sequence of instructions. The optimizations are usually expressed as parameterized replacement rules, so that, for example,

```
mov r1, r2; mov r2, r1 => mov r1, r2
```

expresses that if the value of register `r1` is copied to register `r2`, then the following instruction `mov r2,r1` is useless and can be deleted. Today, peephole optimization rules are hand-written, relying on the experience and insight of experts in the machine architecture to recognize and codify the important rules.

In this paper we explore a different approach to building peephole optimizers that is both completely automatic and more systematic. The basic idea is to use *superoptimization* techniques (described further below) to automatically discover replacement rules that are optimizations; this computation is done off-line. The optimizations are then organized into a lookup table, mapping original sequences to their optimized counterparts. Optimization of a compiler's generated code can then be done as efficiently as a normal peephole optimizer, simply using the precomputed rules.

This architecture, where optimizations are computed off-line and then presented as an indexed structure for efficient lookup, is much closer to a search engine database than to a traditional optimizer. Unlike standard compilers where every user has a copy of the entire system, search engines have so much data that it is more efficient to keep the data at a central site and provide access to users over a network. We believe it is possible to build a peephole optimizer using our approach with many millions of learned optimizations, and at that scale the most efficient deployment may also be as a network-based search engine. Our goals in this paper are considerably more modest, focusing on showing that an automatically constructed peephole optimizer is possible and, even with limited resources (i.e., a single machine) and learning hundreds to thousands of useful optimizations, such an optimizer can find significant speedups that standard optimizers miss.

The classical meaning of *superoptimization* [10] is to find the optimal code sequence for a single, loop-free assembly sequence of instructions, which we call the *target sequence*. As noted in later work [8], the term superoptimization is an oxymoron: If a program has been optimized—meaning it is optimal—then what can it mean to be superoptimized? The terminology problem lies in the need to distinguish superoptimization from garden variety *optimization* as that term is normally used; compiler optimizations are really just code improvers and it is an accident if a conventional optimizer produces an optimal program. However, for brevity, we will often refer to our own system as an optimizer rather than as a superoptimizer.

There have been two approaches to superoptimization explored in the past. The first, used in Massalin's original paper [10], simply enumerates sequences of instructions of increasing length or cost, testing each for equality with the target sequence; the lowest cost equivalent sequence found is the optimal one. The second approach, pursued in Denali, constrains the search space to a set of equality-preserving transformations expressed by the system designer. For a given target sequence, a structure representing all possible equivalent sequences under the transformation rules is searched for the lowest cost equivalent sequence [8]. A common point of view in both approaches is that superoptimization is something that is expensive, potentially requiring many hours of computation to optimize a single target instruction sequence, and that the main application is as an aid to human performance experts in speeding up the occasional critical inner loop.

Our work differs from this previous work in a number of ways, beginning with the goal. Our main interest is in creating a peephole superoptimizer that is fast enough and systematic enough to be worth using in every compilation. We are also interested in investigating, to what extent the considerable human labor needed

ASPLOS'06 October 21–25,2006, San Jose, California, USA
Copyright © 2006 ACM 1-59593-451-0/06/0010. . . $5.00

to write an optimizer can be automated. To this end, we make the following contributions:

- We superoptimize many target sequences (potentially millions) simultaneously in a first, off-line phase. The target sequences are extracted, or *harvested*, from a *training set* of programs. The idea is that the important sequences to optimize are the ones emitted by compilers; we simply take all instruction sequences up to a given length from a representative collection of existing binaries as our training set.
- Because we aim to be applicable to general binaries, our prototype implementation handles nearly all of the 300+ opcodes of the x86 architecture; previous efforts have focused on a much smaller set of register-to-register operations. In particular, we present the first techniques for correctly inferring superoptimizations involving memory accesses and branches, as well as the first approach that takes the context (e.g., the set of live variables) of an instruction sequence into account.
- A key problem in superoptimization is spending as little time as possible considering instruction sequences that cannot be optimal versions of target sequences. We introduce a new technique, *canonicalization*, based on the observation that having once considered a sequence, we need never consider a sequence that is equal up to consistent renaming of registers and symbolic constants. We show that canonicalization dramatically reduces the search space for our system.
- The output of our system is a set of replacement rules. Each rule gives a source (canonical) instruction sequence and the resulting optimized (canonical) instruction sequence. Thus, these rules can be indexed and used as efficiently as the rules in a standard peephole optimizer. The rules we discover may be less general than rules written by humans—i.e., it may require multiple rules discovered by the superoptimizer to cover the same functionality as a single rule written in a more general form. However, a peephole superoptimizer can compensate for less general rules by automatically discovering many more rules than are written for normal peephole optimizers.
- We report experimental results on a number of kernels where our system achieves speedups of between 1.7 and a factor of 10 over code already optimized by a standard compiler. The improvements show that even mature compilers do not come close to the best possible code in at least some relatively simple situations.

We begin with an overview of our system's design (Section 2) followed by a detailed discussion of each of the major components (Sections 3-5). We then present experimental results (Sections 6-7), discuss related work (Section 8), and conclude (Section 9).

2. Design of the Optimizer

We begin by defining a few terms that we use throughout the paper. An *instruction* is an opcode together with some valid operands. For example, on a machine with eight registers `r0` through `r7`, the increment opcode (`inc`) generates eight unique instructions, each operating on a different register. A potential problem arises with opcodes that take immediate operands, as they generate a huge number of instructions. We restrict immediate operands to a small set of constants and symbolic constants; in this way, we ensure opcodes with immediate operands generate only a small number of distinct instructions.

A *cost function* captures the approximate cost of an instruction sequence on a particular processor. We use different cost functions for different purposes; e.g., running time to optimize speed, instruction byte count to optimize the size of a binary. An instruction sequence is *optimal* if no equivalent sequence of lower cost exists. Equivalence of two instruction sequences is defined under a *context*, which is a subset of the machine state that is live beyond the instruction sequences themselves. Since we ignore I/O instructions, the machine state for our purposes consists of registers, stack and memory. The context of a target instruction sequence can potentially include registers, memory locations and stack locations live at the program point where the instruction sequence ends. However, for implementation simplicity, we currently conservatively assume memory and stack locations are always live. The context of an instruction sequence is thus reduced to the set of registers live on exit from the sequence.

An *equivalence test* $\cong_L$ tests two instruction sequences for equivalence under the context (set of live registers) L. For a target sequence T and a cost function c, we are interested in finding a minimum cost instruction sequence O such that

$$(O \cong_L T)$$

Unlike previous efforts, our superoptimizer computes the optimal instruction sequences for several different target sequences simultaneously. Moreover, once an optimization is found, it is saved in an indexed *optimization database* so that the expensive work done to compute the optimizations need never be repeated again. Thus, the database represents all the optimizations acquired by running the superoptimizer. Once computed, these optimizations can be used to optimize any number of programs.

Our optimizer is structured in three parts:

- The *harvester* extracts target instruction sequences from the training applications. The target instruction sequences are the ones we seek to optimize.
- The *enumerator* exhaustively enumerates all possible candidate instruction sequences up to a certain length, checking if each candidate sequence is an optimal replacement for any of the target instruction sequences.
- The optimizer applies the *optimization database*, an index of all discovered optimizations, to applications.

There are two key challenges for our approach. First, we must reduce the search space of the enumerator as much as possible (Section 4.2). Second, we need a very efficient test for equivalence of two instruction sequences (Section 5.1).

A flowchart of the superoptimizer is shown in Figure 1. We discuss the components shown in the flowchart in the following sections.

3. Harvesting Target Instruction Sequences

The first step in creating a superoptimizer using our approach is to obtain target instruction sequences from a representative set of applications. These *harvested* instruction sequences form the corpus used to train the optimizer. Not all instruction sequences are harvestable in our current implementation. A harvestable instruction sequence I must have a single entry point—no instruction in I (except the first instruction) should be a jump target of any instruction outside of I. To enforce this constraint, we identify all jump targets of direct-jump instructions in the binary executable. Also, we identify all instructions starting at addresses pointed to by symbol names since these instructions are possible targets of indirect jump instructions. Any such instructions should not be a part of a harvested instruction sequence I (except possibly being the first instruction in I). Notice that a harvested instruction sequence can have multiple exits since we allow jump instructions in the sequence.

When the harvester extracts instruction sequences from a binary, it also records the set of registers live at the end of the se-

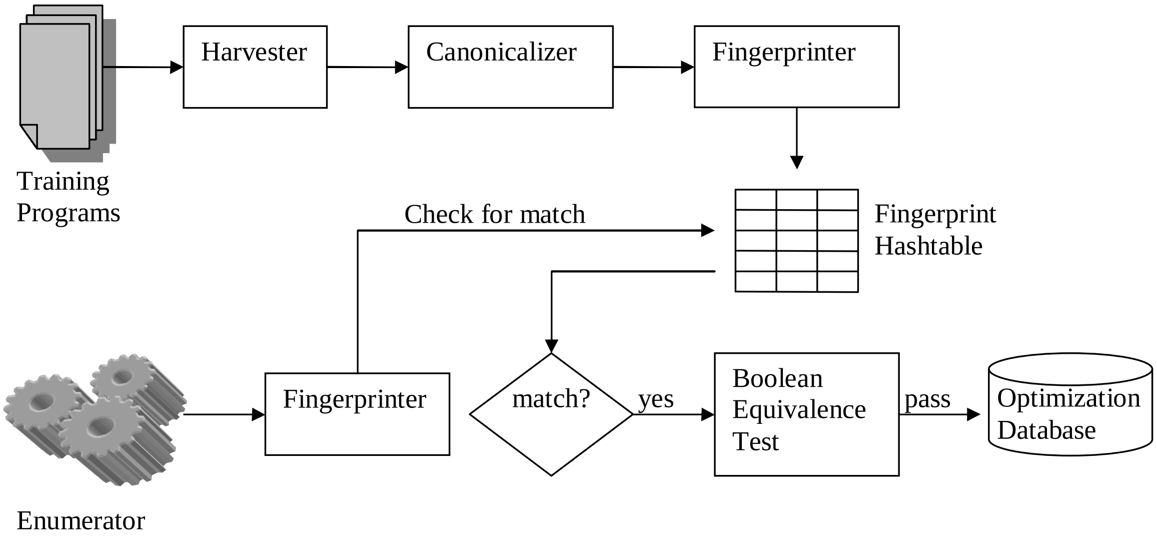

Figure 1. Flowchart of the superoptimizer.

quence; this context is used in determining equivalence as discussed in Section 2.

3.1 Canonicalization

All well-formed instruction sequences are valid candidates for optimization, but many sequences are just transformations of each other under renamings of registers and immediate operands. For example, on a machine with eight registers, an instruction `mov r1, r0` has $8 * 7 = 56$ equivalent versions with different register names. To reduce wasted effort, one would like to eliminate all unnecessary instruction sequences that are mere renamings of others—a process we call *canonicalization*.

An instruction sequence is *canonical* if its registers and constants are named in the order of their appearance in the instruction sequence i.e., the first register used is always `r0`, the second distinct register used is always `r1`, and so on. Similarly, the first constant used in a canonical instruction sequence is (the symbolic constant) `c0`, the second distinct constant `c1`, and so on.

An instruction sequence is *canonicalized* by renaming registers and constants. An optimization that applies to a sequence is also valid for its canonicalization (with registers and constants suitably renamed). Hence, we store only canonical forms of instruction sequences in our optimization database. Optimizing an instruction sequence I requires first canonicalizing I to $\theta(I)$, where θ is the canonical renaming of registers and symbolic constants of I. We then search the database for a sequence R equivalent to $\theta(I)$, and then "uncanonicalize" R to $\theta^{-1}(R)$ so that the registers and constants have their original names as in I. The sequence $\theta^{-1}(R)$ then replaces I in the application.

Dealing with only canonical instruction sequences dramatically reduces the size of the corpus of target instruction sequences. Figure 2 plots the number of unique harvested instruction sequences before and after canonicalization. At short instruction sequence lengths, there are many fewer unique canonical instruction sequences than the number of unique harvested sequences. At longer lengths, the number of harvested instruction sequences decreases because fewer sequences meet the harvester's constraints.

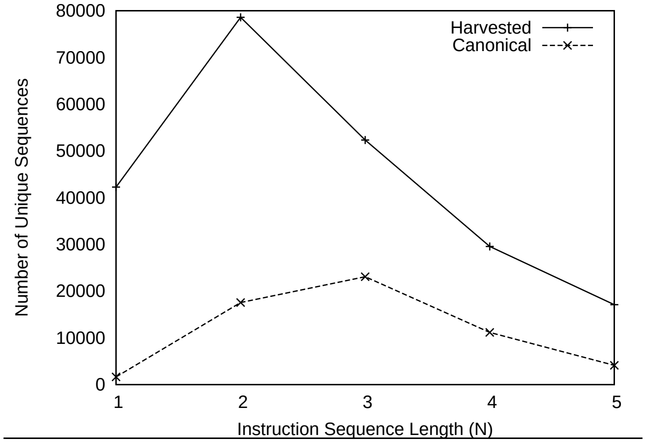

Figure 2. The number of unique harvested instruction sequences in SPEC CINT2000 benchmarks, before and after canonicalization.

3.2 Fingerprinting

The most common operation in our off-line computation of optimizations is determining whether an instruction sequence I is equivalent to any target instruction sequence. We execute I on test machine states and then compute a hash of the result, which we call I's *fingerprint*. The fingerprint is the index into a hashtable; each bucket holds the target instruction sequences, if any, with that fingerprint. The most important properties of the fingerprint are that it is very fast to compute and results in at most a small set of target sequences that might be equivalent to I.

We have found it sufficient to use two pseudo-random machine states called *testvectors* to compute fingerprints.[1] The instruction sequence is first converted into an executable binary form. The machine is loaded with a testvector and control is transferred to the instruction sequence. The machine state (the contents of registers,

[1] Each bit in the two testvectors is set randomly, but the same testvectors are used for fingerprinting every instruction sequence.

Original Instruction Sequence

```
        inc r1
        xchg (r2), r1
```

With Memory Sandboxing Instructions

```
        inc r1
          mov r2, r7
          and $0xff, r7
          add $membase, r7
        xchg (r7), r1
```

Figure 3. The memory array M starts at address `membase` and is $2^8 = 256$ bytes long. Every memory access is prepended with three instructions ensuring the memory access is contained within M. In this example, a temporary register `r7` was used to perform this function.

status bits, and memory—see below) is recorded after the instruction sequence finishes execution. This process is repeated for both testvectors and a hash is then computed on the machine states that were obtained.

Executing the instruction sequence on the bare machine has three advantages. First, it is extremely fast. Second, it eliminates sources of error due to incorrect simulation of instructions. And third, machine counters can be used to estimate the time spent in executing the instruction sequence on hardware, providing hints for shaping the time-based cost function.

While executing the instruction sequence directly on hardware is good, it presents its own set of challenges. In particular, we must isolate the state of our system from any side-effects of the instruction sequence. We save all registers before executing the instruction sequence and restore them after the execution is finished. We *sandbox* all memory and stack references by adding extra instructions to the executed code. Both memory and stack accesses are constrained to small regions of memory in the address space of the superoptimizer. The memory is approximated by a small array M of size 2^s starting at a memory address membase. Each instruction performing a memory access is then prepended with instructions ensuring that the memory access does not fall outside M. A similar approach is taken for stack references. Figure 3 shows the sandboxing instructions used for the x86 instruction set. Note that this strategy for handling memory references preserves the property that if two instructions sequences are equivalent they result in the same machine state on any testvector and therefore have the same fingerprint. We have found that a sandboxed memory of size $M = 256$ bytes is sufficient for minimizing fingerprint collisions between inequivalent instruction sequences.

The function used to hash the machine states obtained after the execution of the instruction sequences on the testvectors must have some special properties to ensure minimal collisions. First, it should be asymmetric with respect to different memory locations and registers, which is necessary to distinguish between instruction sequences performing identical operations at two different locations. Second, it should not be based on a single operator (like `xor`); otherwise, there are likely to be many collisions on instruction sequences using that particular operator. We employ a combination of `xor` and weighted-add operations to compute the hash of the machine state. To handle context correctly, when fingerprinting a target sequence the hash function includes only the live registers; the values of the dead registers are discarded.

Finally, the full structure of the fingerprint hashtable is more elaborate than we have described so far. For each target instruction sequence I, the hashtable records I and the fingerprint not only for the canonicalization of I, but also for all of I's different register and symbolic constant renamings. This, as we describe in Section 4.2, helps us in reducing the search space of the enumerator. Hence, an instruction sequence using r distinct registers and c distinct constants can generate at most $r! * c!$ fingerprints. Typically $r \leq 5$ and $c \leq 2$, so the blow-up is upper-bounded by 240. In practice, we find that the blow-up is around 18. The fingerprint hashtable is indexed by an instruction sequence's fingerprint and set of live registers.

In summary, the fingerprint hashtable maps a fingerprint and set of live registers to a set of instruction sequences with the same fingerprint under that context. This table forms the corpus of instruction sequences that we wish to superoptimize.

4. Enumerator

The enumerator simply enumerates all possible, unique instruction sequences. We discuss the enumerable instruction set, techniques to reduce the search space, and the search for useful optimizations in the following subsections.

4.1 Enumerable Instruction Set

Instruction sequences are enumerated from a subset of all instructions. At most one branch instruction is allowed in an instruction sequence. For the branch instruction, a canonical target is defined which represents an exit point outside of I. Hence, an enumerated instruction sequence is allowed to have at most two different exits: the straight-line exit point in the code, and the exit defined by the branch instruction. Notice that while an enumerated instruction sequence can have at most one branch instruction, a target instruction sequence could have more branches; many optimizations eliminate or reduce the number of branches in the target sequence.

To bound the search space, we restrict the maximum number of distinct registers and constants that can appear in an enumerable instruction sequence. For instructions using a restricted subset of registers, only that subset is considered during enumeration. For constants we allow the numbers `0` and `1`, the symbolic constants `c0` and `c1`, and addition or subtraction where the first argument is a symbolic constant and the second argument is a symbolic constant or 1. Allowing addition and subtraction of constants enables discovery of local constant-folding optimizations. Constant-folding optimizations involving more than two constants are captured by repeated application of optimizations to a code sequence.

We also restrict the number of distinct registers used in an enumerated instruction sequence. The number of registers used by instruction sequences varies greatly. We profiled some CPU-intensive applications to gauge this distribution (see Figure 4) and observed that more than 50% of harvested instruction sequences of length 8 use fewer than 4 machine registers. Thus, we decided to allow at most 4 distinct registers in an enumerated instruction sequence. Again, notice that a target instruction sequence can use more registers than the corresponding optimal instruction sequence. In fact, many optimizations produced by the superoptimizer eliminate redundant registers.

The number of indirect memory accesses in an instruction sequence is constrained by the number of registers allowed, since an indirect memory access dereferences a register. For direct memory accesses, we allow at most two distinct direct memory addresses (`c0` and `c1`). Because we use the symbolic constants `c0` and `c1` as both values of immediate operands and memory addresses, we capture optimizations involving the transformation of indirect memory accesses to direct memory accesses.

Figure 5 shows examples of opcodes of different types and the instructions generated by them.

4.2 Reducing the Search Space

Once the enumerable instruction set is fixed, the enumerator's search space is exponential in the length of the instruction se-

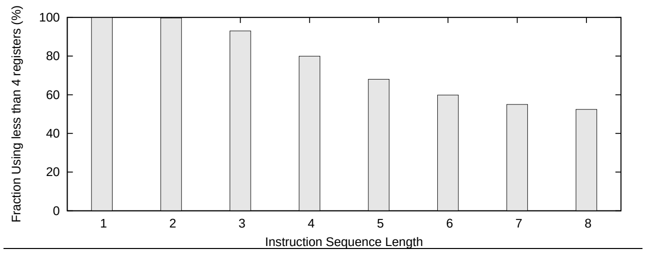

Figure 4. Pattern of register usage of harvested instruction sequences in SPEC CINT2000 benchmarks.

`not` <*register*>

`not` r0
`not` r1
`not` r2
`not` r3

`dec` <*memory location*>

`dec` (r0)
`dec` (r1)
`dec` (r2)
`dec` (r3)
`dec` (c0)
`dec` (c1)

`add` <*mem-indirect*>, <*immediate*>

`add` (r0), 0
`add` (r0), 1
`add` (r0), c0
`add` (r0), c1
`add` (r0), c0+1
`add` (r0), c0-1
`add` (r0), c0+c1
`add` (r0), c0-c1
and repetition of the above for `r1`, `r2`, . . .

Figure 5. Examples of instructions generated by opcodes taking different operand-types in the x86 instruction set.

quence. We use two techniques to reduce the size of the search space.

- We enumerate only canonical instruction sequences. While this decision reduces the size of the enumerated set of sequences, it does cause a blow-up in the size of the fingerprint hashtable (recall Section 3.2).
- We prune the search space by identifying and eliminating instructions that are functionally equivalent to other cheaper instructions.

  For simple cost functions, it is possible to further prune the search space by observing that all subsequences of a length n instruction sequence must be optimal—if any subsequence is not optimal, then it can be replaced by a cheaper sequence and hence the sequence is not optimal. This is always true when we are optimizing for codesize, since the cost function is simply the sum of individual instruction lengths. For runtime optimizations, this is not true in general. In our experiments, we employed this aggressive pruning strategy only when optimizing for codesize. Pruning the search space at smaller instruction sequence lengths provides a significant benefit for longer instruction sequences. This idea was first proposed by Massalin [10]. We currently check that all subsequences of length 2 are optimal using a table that lists all length 2 optimal sequences, when optimizing for codesize.

Table 1 lists the size of the set of enumerated instruction sequences with and without canonicalization and pruning. While canonicalization provides the biggest reduction, the effect is cumulative and using both techniques we achieve over 50x improvement in the size of the search space for instruction sequences of length 3 on the x86 architecture. In Table 1, the reduction due to pruning is only due to elimination of single instructions that are equivalent to other single instructions. For the codesize cost function, where we can employ the more aggressive pruning strategy, we get a total improvement of 60x (20% more) in the size of the search space at length 3.

Length	Original Search Space	After Canonicalization	After Pruning	Reduction Factor
1	5,453	997	644	8.5
2	29 m	2.49 m	1.2 m	24.7
3	162.1 b	8.6 b	3.11 b	52.1

Table 1. The size of the search space for x86 instruction sequences of length 1 to 3. The last column shows the reduction in search space achieved through pruning and canonicalization.

Many of the enumerated sequences are redundant and it is tempting to avoid enumerating them by placing checks in the enumerator. For example, it is possible to check for instruction sequences of the form {`mov r0, r1; mov r0, r1`} and avoid fingerprinting them. However, such checks in the inner loop of the enumerator result in an overall slowdown. In the interest of speed, we let the system weed out such special cases automatically through fingerprinting and equivalence checks.

The enumerator stores enumerable instructions in a table with information about the registers and constants used to help the enumerator generate only canonical instruction sequences. The table is sorted in an order to make enumeration fast. Using the fast fingerprint technique, about $500,000$ instruction sequences per second can be enumerated and fingerprinted on a single processor.

4.3 Searching the Fingerprint Hashtable

Each enumerated instruction sequence is fingerprinted as described in Section 3.2. The fingerprint is computed for all possible sets of live registers. The fingerprint value and the corresponding set of live registers is then used to look up any matching target instruction sequence in the fingerprint hashtable. If there is a match, we have found a candidate optimization and proceed with the equivalence test described in Section 5.1. If there is no match in the fingerprint hashtable, the enumerated instruction sequence is simply discarded.

Recall that while we enumerate only canonical instruction sequences, the fingerprint hashtable contains instruction sequences in both canonical and non-canonical forms. This is important, because it is possible to optimize a canonical instruction sequence with an non-canonical instruction sequence and vice-versa. For example, a canonical length 2 instruction sequence T {`mov r0, r1; mov r1, r2`} can be optimized using a non-canonical length 1 instruction sequence O {`mov r0, r2`} (assuming `r1` is not live). To catch this optimization, we keep all renamings of T in the fingerprint hashtable and enumerate only the canonical version of O. In this example, the non-canonical renaming of T {`mov r0, r2; mov r2, r1`} in the fingerprint hashtable is optimized by the canonical enumerated sequence {`mov r0, r1`}.

5. Learning an Optimization

Once a match is found in the fingerprint hashtable for an enumerated instruction sequence, an equivalence test is performed. If the target instruction sequence and the candidate instruction sequence are found to be equivalent, and the cost of the candidate instruction sequence is lower than the target (or a previously discovered optimization for that target), the optimization is stored in the optimization database. Each of these steps is described in the following subsections.

5.1 Equivalence Test

The equivalence test proceeds in two steps—a fast but incomplete execution test and a slower but exact boolean test.

5.1.1 Execution Test

Our fast execution test is similar to fingerprinting. We run the two sequences over a set of testvectors and observe if they yield the same output on each test. In our experiments, we use a total of 18 testvectors: one is all zeros, one is all ones and in the remaining 16, each bit is set randomly.

Contrary to Massalin's experience [10], we found a number of pairs of instruction sequences that passed the execution test and failed the boolean test.[2] This situation arises due to a variety of reasons, almost all involving loss of bits during the computation. For example, an equality comparison of two computed registers on the testvectors is likely to always return false. Similarly, memory addresses are almost never aliased by execution tests, while a boolean deterministic test catches all inconsistencies due to the possibility of memory aliasing.

5.1.2 Boolean Test

The boolean verification test represents an instruction sequence by a boolean formula and expresses the equivalence relation as a satisfiability constraint. The satisfiability constraint is tested using a SAT solver.

A machine state is represented by a finite set of registers and a model of the full memory and stack. Registers are represented as bitvectors. Memory is modeled by a map from address expressions to data bits. The first use of a memory location is encoded by fresh boolean variables representing the data bits at that address. Boolean clauses are used to encode the relationship between the data bits and address bits. e.g., for a sequence performing two memory reads at addresses $addr_1$ and $addr_2$, and returning data bytes $data_1$ and $data_2$ respectively, the following clause captures their aliasing relationship:

$$(addr_1 = addr_2) \Rightarrow (data_1 = data_2)$$

All memory writes are stored in a table in order of their occurrence. For a memory read occurring after memory writes, the read-address needs to be compared with the address expressions of the writes. Each read-access R is checked for address-equivalence with each of the preceding write accesses W_i in decreasing order of i, where W_i is the i'th write access by the instruction sequence. The following clause encodes this relationship between the data of the read access $data_R$ and the data of one of the preceding write accesses $data_{W_i}$.

$$\bigvee_{j \geq i} (addr_R \neq addr_{W_j}) \wedge addr_R = addr_{W_i} \Rightarrow data_R = data_{W_i}$$

For each pair of memory accesses, a boolean clause is generated to capture the possibility of their address expressions aliasing with each other. Where information is not available, we conservatively assume that two memory addresses may alias. The equivalence of two memory states is checked by reading the bits at each address location for both states and checking them for boolean equivalence. The model of the stack is identical to that of memory, with additional bits representing the stack and frame pointers.

Instructions are encoded as boolean circuits transforming an input machine state to an output machine state. Branch instructions are handled by predicating the execution of instructions on the true and false paths with the branch condition or its negation. The program counter is modeled to indicate if a branch to a target outside the instruction sequence was taken. The input state is shared between the two the instruction sequences being checked for equivalence. Two instruction sequences are equivalent iff the registers, memory and stack expressions obtained in the final state are equivalent. The equivalence relation of the output machine states is expressed as a satisfiability constraint before giving it to the SAT solver.

5.2 Optimization Database

The optimization database records all optimizations discovered by the superoptimizer. The database is indexed by the original instruction sequence (in its canonical form) and the set of live registers, and returns the corresponding optimal sequence if one exists. Because instruction sequences stored in the fingerprint hashtable need not be canonical, they must be canonicalized (and their optimal versions renamed) before storing them in the optimization database.

The operation of optimizing a binary executable is fast: it involves only harvesting a target sequence, canonicalizing it, and searching the indexed optimization database. Multiple optimization passes are performed on the executable until no further optimizations are found.

6. Experimental Results

Our implementation of the optimizer is written in C++ and O'Caml [9]. We use the Diablo link-time rewriting framework [3, 12] to compute liveness information for an x86 executable binary. We use zChaff [11, 13] as our backend SAT solver because of its performance and incremental SAT solving capabilities. It took around two weeks to write formulas modeling the opcodes of the Intel Pentium instruction set for the boolean test. We compared our optimizer on executables compiled using `gcc` version 3.2.3. The default optimization level used was `-O2`.

Our experiments were done using a Linux machine with a single Intel Pentium 3.0GHz processor and 100 Gigabytes local storage. We limited the peephole size to instruction sequences of length 3, which were not too time consuming to enumerate. Given more resources, we can easily scale the system to length 4 instruction sequences, which we believe, would produce even better results. Going beyond length 4 instruction sequences requires additional techniques to further reduce the search space of the enumerator. Although, we enumerate only up to length 3 instruction sequences, we optimized windows of up to length 6 instruction sequences in our experiments.

We use two different cost functions, one capturing runtime and the other codesize. The codesize cost function simply considers the size of the executable binary code of a sequence as its cost. The runtime cost function is more involved. It first takes into account the number of memory accesses and branch instructions in the sequence. Then, the approximate cycle costs of the instruction are considered, as obtained from the technical manuals on Intel architectures. In case of a tie, the number of registers used and the code length are used as tie-breakers. [3]

[2] Massalin did not implement a complete test, relying on humans to confirm that candidate optimizations that passed an execution test were correct in all circumstances.

[3] We tried using Pentium performance counters to estimate the runtime of an instruction sequence. In our experience, that was not useful for short

In our first set of experiments, we took some kernels operating on arrays of integer elements. All the kernels were written in C. A description of each of the kernels is given in Table 3. These kernels were compiled using architecture specific (`-march=pentium4`, `-mmmx` and `-msse`) optimization options in gcc, with the loops unrolled 8 times.

Figure 6 plots the runtime improvements our superoptimizer obtained in the different kernels over gcc. We achieved improvements of between 1.7 and 10 times over already-optimized code. Some (but not all) of the large improvements in running time are because the superoptimizer finds clever ways to use the SIMD (single instruction multiple data) instructions available in the Intel architecture. The problem of emitting efficient SIMD code has confounded compiler-authors for many years; gcc at least does not appear to attempt to use SIMD instructions. Most code involving the use of complex instructions is currently hand-coded by expert assembly programmers. Our results show that an automatically generated optimizer is at least a partial solution to this problem.

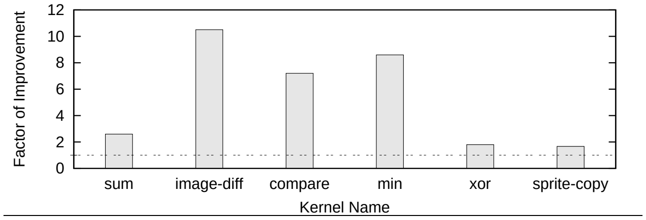

Figure 6. Speedups for the kernels in Table 3.

Next, we applied the superoptimizer to applications from the SPEC CINT2000 benchmarks [7]. The number of optimizations performed and the corresponding improvements over `gcc` are shown in Table 4. As one would expect, the improvements are much less dramatic for full applications than for compute-intensive kernels. We found speedups of 1-5% with these improvements, though we found that speedup varied across different runs and machine configurations. We saw improvements in code size of 1-6% over executables already optimized for size using `-Os`.

We also ran our optimizer on SPEC executables compiled using the architecture specific Intel C++ compiler `icc` [1]. For the SPEC benchmarks, the speedups obtained on `icc` optimized executables were less than 1%, but we found that the codesize of these executables reduced by 2.5-4% with no performance penalty. On the kernels, our optimizer achieved speedups over `icc` comparable to the results with `gcc`.

A sample of some interesting optimizations performed on binaries that had been already optimized using gcc are given in Table 5. The system found a range of optimizations, from ones that are well-known (constant folding, redundant load elimination, strength-reduction) to very architecture specific optimizations (the use of the `xchg` instruction to swap registers, and various uses of the SIMD instructions). We discuss two discovered optimizations in detail. In Example 1 of Table 5, the superoptimizer finds a three-instruction sequence to compute the sum of eight unsigned byte integers using the 64 bit registers available on the x86 platform. It first zeros out one of the 64 bit registers (`mm0`) by subtracting it from itself. It then uses the `psadbw` instruction, which computes the sum of absolute differences of two 64-bit values. Since one of the registers in this sequence is zero, this amounts to the computation of the sum of the eight bytes in the other operand. The third instruction then stores the computed sum to the memory location `sum`. In Example 5, the destination (register `esi`) is intended to be zeroed out only if the comparison flag in the machine is set; here gcc produces clever code to avoid a branch instruction. The target sequence emitted by gcc reads the flag to a register `eax`, decrements it (causing it to be either 0 or -1) and then computes the bitwise-and of `eax` and `esi`. Since -1 is represented by all 1s in two's complement, this effectively sets `esi` to zero only if the comparison flag was set. The superoptimizer proposes the use of a simple conditional-move `cmov` instruction to achieve the same result.

A total of around 3000 codesize optimizations and 2100 runtime optimizations were learnt after training the optimizer on a diverse set of integer programs. One metric of importance is the frequency of use of these optimizations. We found a tremendous amount of re-use. Table 2 presents a profile of the optimizations that were applied to the SPEC integer benchmarks. Five optimizations were used more than 1,000 times each; in total over 600 distinct optimizations were used at least once each on these benchmarks. To further study the re-use of optimizations, we trained the optimizer on one set of executables and optimized another set of executables. We found that most optimizations are captured even though the executable being optimized was not a part of the training set. For example, 97% of the optimizations were captured when we ran the optimizer on the popular internet browser `firefox` after training it only on the SPEC benchmarks.

Frequency Of Use	Number of Optimizations	Number of Applications
> 1000	8	18679
$201 - 1000$	7	4098
$51 - 200$	33	2823
$11 - 50$	82	1737
$1 - 10$	474	1256

Table 2. Profile of the number of optimizations and the number of times they were applied on SPEC CINT2000 benchmarks.

The process of optimizing a full binary using the optimization database is very fast, completing in less than two seconds on these benchmarks. A prototype of our system is available online at [2].

7. Discussion

In this section we show in detail how our system optimizes a simple loop; the purpose is to illustrate what our techniques can, and cannot, do using a small but fairly realistic example. Consider the following C program to traverse a linked list of integers, multiplying each element by 2:

```
struct node
{
  int val;
  struct node *next;
};

void traverse (struct node *head)
{
  while (head)
  {
    head->val *= 2;
    head = head->next;
  }
}
```

The following assembly code is generated by gcc without optimizations for the loop body of `traverse()` (`eax`, `edx` are machine registers, `ebp` is the register holding the frame pointer).

```
1 : movl 8(%ebp), %edx    #edx  := head
```

sequences due to the large variance in the numbers obtained across different runs.

Kernel Name	Description	Pseudo-code
sum	Calculate the sum of unsigned byte-integers in an array	sum += a[i]
image-diff	Calculate the sum of absolute differences of image pixels	sum += ABS (a[i] - b[i])
comparison	Compare each element of two arrays	c[i] = (a[i] $<$ b[i]) ? c0 : c1
min	Find the minimum of each element of two arrays	c[i] = (a[i] $<$ b[i]) ? a[i] : b[i]
xor	Computes exclusive-OR over two arrays	c[i] = b[i] $\oplus$ a[i]
sprite-copy	Rendering sprite graphics (Game Programming)	c[i] = (a[i] $==$ 0) ? b[i] : a[i]

Table 3. Superoptimized kernels, operating on arrays of 4 million elements.

		Runtime		Codesize	
Program	Description	Number of Optimizations	Instructions Eliminated	Number of Optimizations	Codesize Improvement
gzip	Data Compression Utility	621	4.16%	402	3.95%
mcf	Minimum Cost Network Flow Solver	381	3.73%	335	5.86%
crafty	Chess Program	1074	2.19%	758	1.71%
bzip2	Data Compression Utility	396	4.11%	301	4.58%
gcc	C compiler	10326	2.44%	2996	1.12%
parser	Natural Language Processing	1123	3.84%	582	3.06%
twolf	Place and Route Simulator	1125	2.17%	619	1.47%

Table 4. Results of running the optimizer on SPEC CINT2000 benchmark applications. The runtime improvements are shown over 'gcc -O2' optimization. The codesize improvements are shown over 'gcc -Os'.

	Description	Target Sequence	Optimal Sequence	Live Registers

1.	Sum of byte-integers in an array	sum += a[i]
		sum += a[i+1]
		. . .
		sum += a[i+7]
			psubb %mm0, %mm0
			psadbw &a[i], %mm0
			movd %mm0, sum
				sum

2.	eax $\leftarrow$ ecx - eax - 1	sub %eax, %ecx mov %ecx, %eax dec %eax	notl %eax add %ecx, %eax	eax
3.	Elimination of Branch Instructions	sub %eax, %ecx test %ecx, %ecx je .END mov %edx, %ebx .END:	sub %eax, %ecx cmovne %edx, %ebx	eax, ecx, edx, ebx
4.	Swap two Registers	mov %eax, %ecx mov %edx, %eax mov %ecx, %edx	xchg %eax, %edx	eax, edx
5.	Use of Conditional Move Instruction	setg %al movzbl %al, %eax dec %eax and %eax, %esi	mov $0, %eax cmovg %eax, %esi	esi
6.	Constant Folding	mov $8, %eax sub %ecx, %eax dec %eax	mov $7, %eax sub %ecx, %eax	eax, ecx
7.	Elimination of Redundant Loads	mov %eax, -20(%ebp) mov -20(%ebp), %ecx	mov %eax, -20(%ebp) mov %eax, %ecx	ecx

Table 5. Examples of runtime optimizations performed by the superoptimizer on gcc-optimized executables.

```
2 : movl 8(%ebp), %eax    #eax  := head
3 : movl (%eax), %eax     #eax  := head->val
4 : sall %eax             #left-shift eax by 1
5 : movl %eax, (%edx)     #head->val := eax
6 : movl 8(%ebp), %eax    #eax  := head
7 : movl 4(%eax), %eax    #eax  := head->next
8 : movl %eax, 8(%ebp)    #head := eax
9 : cmpl $0, 8(%ebp)      #head == null?
```

The superoptimizer first replaces instruction 2 with

$2'$:`movl %edx, %eax`

and instruction 9 with

$9'$:`cmpl $0, %eax`

eliminating two redundant loads. Then, the instruction sequence $2', 3, 4, 5$ is replaced with a single instruction

$3'$:`sall (%edx)`

taking advantage of the fact that `eax` is not live at the end of instruction 5. It is inferred that locations `8(%ebp)` and `(%edx)` in instructions 1 and $3'$ cannot alias with each other by comparing the types of instruction operands. Hence, in the third step, the instruction sequence $1, 3', 6$ is replaced by the sequence $1, 3', 6'$ with

$6'$:`movl %edx, %eax`

eliminating another redundant load. Instructions $6'$ and 7 are replaced by

$7'$:`movl 4(%edx), %eax`

eliminating a register copy and finally the use of register `eax` is eliminated in instructions $7'$, 8 and $9'$ by replacing it with `edx` in all three instructions. After these optimizations, the assembly code is:

```
1 : movl 8(%ebp), %edx   #edx  := head
3': sall (%edx)          #left-shift head->val by 1
7': movl 4(%edx), %edx   #edx  := head->next
8': movl %edx, 8(%ebp)   #head := edx
9': cmpl $0x0, %edx      #edx == null?
```

A standard optimizing compiler produces the following code (`eax` holds the value of `head` before entering the loop body):

```
1 : sall (%eax)          #left-shift head->val by 1
2 : movl 4(%eax), %eax   #eax  := head->next
3 : testl %eax, %eax     #eax == null?
```

In this example, our automatically generated optimizer performs all but one of the optimizations performed by a standard optimizing compiler. The optimization that is missed involves the iteration variable (instructions 1 and 8). Because dataflow analysis gives the standard compiler a global view of the loop's behavior across all iterations, the standard compiler can cache the iteration variable (`head`) in a register avoiding loads and stores at loop boundaries. Our rule-based system cannot currently find this optimization because it does not understand loop-carried dependencies. Unrolling the loop a few times would mitigate this limitation since the intermediate loads can still be eliminated by pattern-matching on short sequences of instructions.

8. Related Work

Superoptimization of code sequences was first proposed nearly 20 years ago, but we are aware of just three efforts that have developed the idea. Massalin first described an exhaustive-search based approach to discover short optimal programs [10]. We have adopted the same basic approach to searching (enumerating) instruction sequences, with the addition of simultaneously optimizing many target sequences and reducing the search space using canonicalized instruction sequences. While Massalin was interested in computing optimal programs for mathematical functions (e.g. `signum`), we compute optimal versions of any instruction sequence (up to a certain length) found in commonly executed code.

Massalin's work reported on the optimization of relatively long sequences (12 instructions), at least compared to ours. To achieve such lengths it was necessary to restrict the enumerable instructions to a very small set of 10-15 hand-chosen opcodes. We deal with roughly 300 opcodes, and so the number of instruction sequences for us grows much more rapidly with length.

The GNU Superoptimizer (GSO) [6] learns optimizations involving elimination of branch instructions for the RS/6000 processor, for later use with the GNU C Compiler (GCC). They use exhaustive search to find the fastest straight-line code computing a goal function. In particular, they find optimal versions of the computation of comparison operators (`A` *rel-op* `B`). This work is perhaps the closest to ours in its goals; we are both interested in learning peephole optimizations. GSO has a large manual component, as a user is required to specify the goal function. Our approach is completely automatic. While GSO has been used to learn a few tens of optimizations, our system has learned thousands and there is no reason the algorithms should not scale to millions of optimizations. GSO also only optimizes register-register operations where the output and inputs of the goal functions are assumed to be in specific registers; we optimize nearly arbitrary sequences.

Another interesting approach to superoptimization is proposed in a system called Denali [8]. Denali requires a set of axioms expressed in first order logic, capturing mathematical operators and the instruction set of the architecture. For example, an axiom could express the fact that integer addition is associative, or that the `leftshift` instruction multiplies its operand by 2. The system then proceeds by matching the program constructs with the corresponding axioms to find all possible ways to compute a goal function and formulates a satisfiability constraint, the solution to which expresses the fastest among all possible equivalent instruction sequences. Unlike our approach, Denali uses goal-directed search, allowing it to find much longer sequences than we can currently generate using exhaustive search. However, Denali has two drawbacks that led us to prefer exhaustive search. First, Denali is dependent on having enough rules (axioms) to cover all interesting cases; we didn't want to rule out optimizations simply because we hadn't thought of them. Second, it is unclear how this approach can be used to optimize several instruction sequences simultaneously; we gain significant efficiency by amortizing the cost of a single exhaustive enumeration of instruction sequences over the optimization of many target sequences.

Peephole optimizers, apart from their typical use in the final optimization pass, have also been used to perform code selection at link time to generate highly portable compilers [5, 4]. In these systems, peephole optimization through pattern-matching is a primary method to perform code optimization. For example, the "very portable optimizer" (VPO) in [4] uses peephole optimization to reduce the volume of intermediate code by a factor of two to three. These systems share our goal of automatically and systematically discovering peephole optimizations. The primary differences with our work are that our equivalence test based on SAT is more general (able to detect more equivalent sequences) and works for longer sequences than previous systems. Discovering each optimization is also more expensive in our approach; however, by partitioning the work into an off-line learning phase that computes a database of optimizations and an actual optimization phase that simply looks up transformations in the database, our optimization phase can be as fast or faster than traditional peephole optimizers.

9. Conclusions

We have described the construction of a system to automatically generate a peephole superoptimizer for a target architecture. The system is currently capable of automatically learning thousands of peephole optimization rules, each replacing the target sequence with the corresponding optimal sequence. In the future, we are interested in extending this approach to longer instruction sequences.

References

[1] Intel C++ Compiler 9.0. Software available at `http://www.intel.com/software/products/compilers/clin`.

[2] Superoptimizer prototype. Available on the web at `http://cs.stanford.edu/~sbansal/superoptimizer/`.

[3] B. Anckaert, F. Vandeputte, B. D. Bus, B. D. Sutter, and K. D. Bosschere. Link-time optimization of ia64 binaries. In *Proceedings of the 10th International Euro-par Conference*, pages 211–220, 2004.

[4] M. E. Benitez and J. W. Davidson. A portable global optimizer and linker. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 329–338, 1988.

[5] J. Davidson and C. Fraser. Code selection through object code optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(4):505–526, 1984.

[6] T. Granlund and R. Kenner. Eliminating branches using a superoptimizer and the gnu C compiler. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, volume 27, pages 341–352, San Francisco, CA, June 1992.

[7] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millenium. *IEEE Computer*, 33(7):28–35, July 2000.

[8] R. Joshi, G. Nelson, and K. H. Randall. Denali: A goal-directed superoptimizer. In *Proceedings of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation*, pages 304–314, Berlin, Germany, June 2002.

[9] X. Leroy, D. Doligez, J. Garrigue, and J. Vouillon. The Objective Caml system. Software and documentation available at `http://caml.inria.fr`.

[10] H. Massalin. Superoptimizer: A look at the smallest program. In *Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pages 122–126, 1987.

[11] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.

[12] L. V. Put, D. Chanet, B. D. Bus, B. D. Sutter, and K. D. Bosschere. Diablo: a reliable, retargetable and extensible link-time rewriting framework. In *Proceedings of the 2005 IEEE International Symposium on Signal Processing and Information Technology*, pages 7–12, 2005.

[13] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.