Using Tracing and Dynamic Slicing to Tune Compilers

James R. Larus
Satish Chandra

Technical Report #1174

August 1993

Using Tracing and Dynamic Slicing to Tune Compilers

James R. Larus and Satish Chandra*
larus@cs.wisc.edu
Computer Sciences Department
University of Wisconsin–Madison
1210 West Dayton Street
Madison, WI 53706, USA

August 23, 1993

Abstract

Performance tuning improves a compiler's performance by detecting errors and missed opportunities in its analysis, optimization, and code generation stages. Normally, a compiler's author tunes it by examining the generated code to find suboptimal code sequences. This paper describes a collection of tools, called *compiler auditors*, that assist a compiler writer by partially mechanizing the process of finding suboptimal code sequences. Although these code sequences do not always exhibit compiler bugs, they frequently illustrate problems in a compiler. Experiments show that auditors effectively find suboptimal code, even in a high-quality, commercial compiler.

After writing a high-quality compiler, its authors improve it with the time-consuming and tedious process of examining generated assembly code to find inefficient code sequences that could run faster or consume less space. These sequences direct a compiler writer's attention to places in the compiler at which improved analysis, optimization, or code generation would result in better code. Typically a compiler writer finds this suboptimal code by reading assembly-language listings of compiled code (see, for example Briggs [6]). Performance tuning of this sort is unavoidable for two reasons. Compilers are large and complex programs that will contain bugs that affect the performance, but not correctness, of generated code. More fundamentally, compiler writing is an art in which the insight necessary for effective heuristics arise from identifying past mistakes.

This paper describes a technique for partially automating the tedious process of detecting suboptimal code. In this approach, specialized programs, called *compiler auditors*, examine a complete instruction and data trace of a compiled test program and dynamically detect sequences of executed instructions that could have been compiled better. The auditor reports these sequences to the compiler writer, who still must examine the designated portions of the compiled program to determine if the code can be improved and how to modify the compiler. The advantage of compiler auditing is that auditors direct a compiler writer's attention to performance problems faster than a manual search and more reliably than intuition. Information and feedback from tools of this sort can reduce the time and effort required to produce a mature,

*This work was supported in part by the National Science Foundation under grant CCR-9101035 and by the Wisconsin Alumni Research Foundation.

117

high quality compiler that generates efficient, as well as correct code. The approach is simple and requires a small investment of time and effort.

A compiled code sequence is *suboptimal* if a shorter or faster sequence of instructions produces the same effect on the program's state. Suboptimal code arises from three sources:

- Defects in a compiler, which can either be errors of omission in which a compiler writer did not consider a case or errors of commission in which the compiler is actually incorrect (*i.e.*, has a bug). For example, a code generator may not produce a decrement-and-branch instruction because it is not included in the code-generator tables or because an earlier stage improperly normalized a loop's bounds.

- Conservative program analysis, which sacrifices precision in particular cases for correctness in general. In this case, a compiler does not perform an optimization because it cannot guarantee a necessary prerequisite. Frequently, program analysis is improved by identifying special cases that can be precisely analyzed.

- Heuristic algorithms, which do not always produce good results. A good example is graph-coloring register allocation, which addresses an NP-complete problem whose precise, efficient solution is not possible. The primary way to understand and improve a heuristic's performance is to examine its behavior on real programs.

These compiler problems are manifest as *performance*, not *correctness*, deficiencies and will not be detected by a typical user or inattentive compiler writer. The compiled code computes the correct output, albeit more slowly than necessary. Isolating and correcting these problems is costly. The decision whether to fix a problem depends on the performance cost of suboptimal code and the frequency with which it is produced. Compiler writers typically guess at the latter parameter. Another application for a compiler auditor is to examine a large body of code to determine how often a suboptimal code sequence arises.

Suboptimal code can be detected either by statically examining generated instructions or by dynamically examining a program trace. The first alternative is the basis of many compilers, which operate by generating simple code and then improve it by identifying and replacing suboptimal code (e.g., [8, 18]). Compiler writers typically tune their compilers by applying the same process in a more intuitive and ad-hoc way on the compiled code. Manually performing this process is difficult and error-prone, particularly for large procedures with complex control flow, because a human must mentally compute large program traces and slices. In addition, a compiler writer may easily miss an optimization twice: first when writing a compiler and then when tuning it. Finally, when applied to a compiler's assembly output, this approach misses inefficiencies introduced at lower levels such as the assembler and linker. As an example, a group at CSRD examined the Cedar parallelizer to determine why it could not restructure the Perfect Club benchmarks [4, 10]. This process was expensive and required several person-months per benchmark [16].

Another alternative is to examine the stream of instructions when a program executes. An auditor can either use this stream to directly identify inefficient sequences or it can construct a dynamic program slice to find redundant computations [1]. This process has the advantages that it is cheaply and easily mechanizable, does not require sophisticated static analysis, and concentrates on frequently-executed traces in a program. On the other hand, this approach changes the definition of suboptimal. Code executed in a sequence or slice is suboptimal for a particular execution. On other paths through the program, the code sequence may be optimal. Consequently, a code sequences does not demonstrate a compiler bug, rather it *illustrates* a

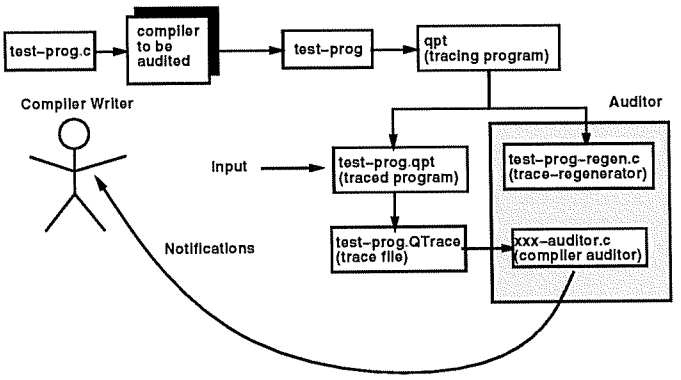

Figure 1: A compiler auditor uses the instruction and data references trace from a test program to detect shortcomings in the compiler that compiled the program. The trace is produced by the qpt tracing system [13, 3].

potential problem. However, the hypothesis of this work—justified by the evidence in Section 3—is that if a code sequence is suboptimal for one execution, it is likely to always be suboptimal.

The remainder of this paper describes how to construct compiler auditors for different facets of a compiler and demonstrates that they can illustrate inefficiencies in a compiler. Section 1 describes in detail the basic framework of the auditors, and how particular auditors can be created. Section 2 discusses how the output produced by auditors can be refined for a compiler writer. Section 3 provides empirical evidence for the efficacy of our approach. Finally, Section 4 discusses related work.

1 Auditing

Although different auditors operate differently, they function in the common framework shown in Figure 1. The compiler being audited compiles a test program (`test-prog.c`) into an executable file (`test-prog`). The `qpt` tracing system [3, 12, 13] instruments the executable file so it writes a highly-condensed trace file (`test-prog.QTrace`). `qpt` also produces a trace regeneration program (`test-prog-regen.c`) that reads the condensed trace and emits a stream of events: execution of an instruction (with its address and line number); basic block entry (with its address); read or write of a memory location (with its address); initiation, iteration, and termination of a loop (with a unique loop identifier); entry and exit to a function (with its address); and memory allocation and deallocation (with the chunk's size and address). The regeneration code is linked to a specific compiler auditor module (`xxx-audit.c`) to form a compiler auditor that can audit many executions of a test program with different input data. Also, the trace file from one program execution can drive many audits.

In the following, we discuss the design and implementation of three different auditors: register-usage auditor, common-subexpression elimination auditor, and loop-invariant auditor. Auditors for other uses can be constructed similarly. Some of these are considered in Section 1.3.

1.1 Register Usage Auditing

Register usage auditing nicely illustrates the strengths and weaknesses of the compiler auditing paradigm. It detects three kinds of redundant register-memory instructions:

- A redundant load, which loads a value that is already in a register.

```
type addr;
type reg_desc = record valid : boolean end;

// Map mem address to set of register descriptors
var holds : addr → set of reg_desc;

// Map each register (number) to unique descriptor
var reg : array[0..max_reg_no] of reg_desc;

On read of memory location x into register d:
foreach r ∈ holds(x) do
    if r.valid then
        print "Redundant load of", x, "which is in reg", r;
remove reg[d] from holds;
add reg[d] to holds(x);
reg[d].valid ← true;

On modification of register d (not including load):
reg[d].valid ← false;

On write of register s to memory location x:
if reg[s] ∈ holds(x) and reg[s].valid then
    print "Redundant store of reg", s, "to location", x;
if ¬reg[s].valid then
    // Delayed cleanup: value of register s was previously modified
    // and no memory location holds current value
    remove reg[d] from holds;
holds(x) ← {reg[s]};
reg[s].valid ← true;

On function entry and exit:
for i ← 0 to max_reg_no do reg[i].valid ← false;
```

Figure 2: Algorithm for detecting redundant loads and stores.

- A redundant store, which saves a value to a memory location that already holds the identical value.
- A redundant spill, which saves a register to the stack frame and then restores it while another register is either dead or holds a live value that is unused in spill range.

Keep in mind that this auditor detects redundancies along a single execution path. An apparently redundant operation may be necessary along other, unexecuted paths or the redundancy may be impractical to detect statically. Nevertheless, redundancies can elucidate shortcomings in various parts of a compiler. For example, common subexpression elimination (CSE) should eliminate redundant loads. Dead-store elimination should remove unnecessary stores [9] and a register allocator need not store an unmodified, spilled value. A register spill while another register holds a dead value indicates that the register allocator unnecessarily spilled a value along at least one path.

Register auditing operates by tracking the relationship between the contents of registers and the contents of memory locations. Figure 2 details how this relationship is computed. After a load operation, the destination register and the source memory location hold equivalent val-

```
type addr;
type reg_desc = record access_time, spill_time : time;
                       candidate_to_spill : set of int
                end;
var reg : array[0..max_reg_no] of reg_desc;

On read of memory location x into register d:
reg[d].access_time ← current_time;
if load is part of register spill then
    for i ← 0 to max_reg_no do
        if (reg[i].access_time < reg[d].spill_time) then
            reg[i].candidate_to_spill ← reg[i].candidate_to_spill ∪ {d};

On modification of register d (including load):
reg[d].access_time ← current_time;
foreach r ∈ reg[d].candidate_to_spill do
        print "Unnecessary spill of register", r, "while", d, "is dead";
reg[d].candidate_to_spill ← ∅;

On write of register s to memory location x:
if store is part of register spill then
    reg[s].spill_time ← current_time;

On use of register u (including store):
reg[u].access_time ← current_time;
foreach r ∈ reg[u].candidate_to_spill do
        print "Spill of register", r, "while", u, "is unused";
reg[u].candidate_to_spill ← ∅;

On function entry and exit:
for i ← 0 to max_reg_no do
    if reg[i] is callee-saved then
        save/restore reg[i].access_time and reg[i].spill_time;
    else
        reg[i].access_time ← current_time;  reg[i].spill_time ← 0;
```

Figure 3: Algorithm for detecting redundant spills.

ues. Similarly, after a store to memory, the source register and destination memory location are equivalent. However, unlike a load, a store destroys the equivalence between other registers and the target memory location. Machine operations that modify a register also destroy the equivalences between the register and memory locations. At function boundaries, all relationships are broken, under the assumption that a compiler does not perform interprocedural memory-access analysis.

The register-memory equivalences help detect redundant loads and stores. A load is redundant if the memory location's value is already in a register. Similarly, a store is redundant if the source register and target location are equivalent. Since the equivalences are based the source of the values, not the literal values themselves, this technique does not report spurious redundancies when two instructions manipulate the same bit-pattern.

Detecting redundant spills requires the additional mechanism in Figure 3. The field *access_time* records when a register was last read or modified and the field *spill_time* records when it was last spilled to the stack frame. When a spilled value is reloaded into register s, any register

r that has a *access_time* before s's *spill_time* is unused since its spill and might have held the value instead of forcing a spill. Register r may be live, in which case the allocator's choice of register s is defensible. However, if the value in r is subsequently redefined before being used (i.e., is dead), the allocator spilled the wrong variable. The auditor also notes which registers are live, but unused while another register is spilled.

A simple heuristic identifies loads and stores that are part of a spill because they write to the current stack frame at a constant offset from the frame pointer (or stack pointer, if the machine has no frame pointer).[1] Access and spill times for callee-saved registers are retained across calls because these registers are unaffected by the call.

As an example of redundant load detection and report, consider the following excerpt from the report on auditing the Unix utility `compress` (compiled using MIPS *cc* with optimizations off:

```
820 loads @0x4019d8 (output+300), in reg 15 from load 0x4019a8 (output+252)
Basic Block 5:
  0x004019a4: lw        r14,-32176(gp)
* 0x004019a8: lw        r15,-32128(gp)
  0x004019ac: nop
  0x004019b0: addu      r25,r14,r15
  0x004019b4: sw        r25,-32176(gp)
  0x004019b8: lw        r24,-32176(gp)
  0x004019bc: lw        r9,-32128(gp)
  0x004019c0: nop
  0x004019c4: sll       r10,r9,3
  0x004019c8: bne       r24,r10,0x401a78
  0x004019cc: nop
Basic Block 6:
  0x004019d0: lui       r18,0x1000
  0x004019d4: addiu     r18,r18,6064
* 0x004019d8: lw        r17,-32128(gp)
  0x004019dc: lw        r11,-32112(gp)
  0x004019e0: nop
  0x004019e4: addu      r8,r11,r17
  0x004019e8: sw        r8,-32112(gp)
```

The first line tells the number of times that a defect occurred (820) and the program counters of the two loads, which are also reported as offsets from the routine (`output`). Basic block 5 dominates block 6 and no other block jumps to block 6, so a register-to-register transfer of `r15` to `r17` would be faster than reloading the value at `0x4019d8`.

1.2 Common Subexpression and Loop-Invariant Code Auditing

Auditors for common subexpression elimination and loop-invariant code motion compute and compare the dynamic slices of the computation that produce the values in each register. These slices are directed acyclic graphs (DAGs) that contain the operands and sequence of operations that produced the value in a register. If the DAGs for two registers are isomorphic, the later computation is redundant since its result could have been obtained from the first calculation

[1] Load and store at a constant offset occasionally arise from other constructs, such as array references with a constant index, but they are infrequent.

```
type addr, operator;
type expr = record op : operator; arg1, arg2 : expr; end
var reg : array[0..max_reg_no] of expr;

On read of memory location x into register d:
set reg[d] to mk_expr (ld, x);

On write of register s to memory location x:
invalidate existing mk_expr (ld, x) node;

On computation rd ← rs1 ∘ rs2:
set reg[d] to mk_expr (∘, reg[rs1], reg[rs2]);

To set register d to expression e:
for i ← 0 to max_reg_no do
    if reg[i] = e then
      print "Expression", e, "is already in register", i;
reg[d] ← e;

On function entry:
for i ← 0 to max_reg_no do
    if reg[i] is callee-saved then save reg[i];
    else reg[i] ← null;

On function exit:
for i ← 0 to max_reg_no do
    if reg[i] is callee-saved then restore reg[i];
    else reg[i] ← null;
```

Figure 4: Algorithm for detecting redundant common subexpressions.

[11]. To detect loop-invariant code, each DAG node is labeled with the shallowest dynamic loop nesting depth at which it could be calculated. Loop invariant code has a nesting depth less than the current loop's nesting depth.

Figure 4 outlines the process of detecting common subexpressions. Each operation builds a new expression DAG from the expression DAGs of its operands. Before saving the new DAG, the auditor compares it against the expression DAGs already in the registers to detect identical expressions. All machine operations are represented as DAG nodes including loads, which appear as an operation on the memory address. When the contents of a memory location changes, extant references to the location are invalidated so they differ from subsequent references to the location's new contents. Only expressions for callee-saved registers are preserved across call boundaries.

This process can be efficiently implemented with the techniques of hash consing and reference counting. Hash consing uses a hash table to map an operator-operand-operand triple to a unique DAG node [17].[2] The benefit of uniquely representing nodes is that equality testing reduces to

[2] Hash consing can canonicalize arguments to commutative and associative operators. The auditor used for Section 3 does not canonicalize operations.

pointer equality. In addition, combining reference counting with hash consing eliminates the need to iterate over registers (at line *) in the expected case in which expression e is newly created (i.e., a reference count of 1) and so no register holds it.

Detecting loop-invariant expressions requires additional fields in each DAG node to record the dynamic loop nesting level and iteration at which the expression is computed and a flag signifying whether it is loop-independent. Arguments of a loop-invariant operations are either computed at a lower loop level or are themselves loop-invariant. The implementation is simple because `qpt` generates events for loop entry, loop re-entry and termination. The dynamic current level and iteration are adjusted at these events appropriately.

Below is an example illustrating a common subexpression detected in a unoptimized version of `compress`:

```
6562 times: CSE R15 @0x401944 (output+152) == CSE R9 @0x401954 (output+168)
(- 0x8 (& @0x10001760 0x7)
Basic Block 0:
  ...
  0x004018d0: lw        r16,-32176(gp)
  ...

Basic Block 1:
  ...
  0x004018f8: andi      r16,r16,0x7
  ...
  0x00401940: li        r14,8
* 0x00401944: subu      r15,r14,r16
  0x00401948: subu      r17,r17,r15
  0x0040194c: lw        r24,72(sp)
  0x00401950: li        r25,8
* 0x00401954: subu      r9,r25,r16
  ...
```

The first line reports how often (6562) the common subexpression on the second line occurs at the two instructions. A register-to-register transfer of `r15` to `r9` at `0x401950` (in place of the two instruction currently at `0x401950` and `0x401954`) would be faster.

We should point out an important limitation in our method. Because of register reuse, this process does not detect all common subexpressions that a compiler could eliminate before register allocation. Once again, an example from `compress`:

```
Basic Block 0:
  ...
  0x004018d0: lw        r16,-32176(gp)
  ...

Basic Block 1:
  ...
  0x004018f8: andi      r16,r16,0x7
  ...

  0x00401900: lui       r25,0x1000
* 0x00401904: addu      r25,r25,r16
```

```
  0x00401908: lbu       r25,1608(r25)
  0x0040190c: nop
  0x00401910: and       r8,r24,r25
  0x00401914: lw        r9,72(sp)
  0x00401918: nop
  0x0040191c: sllv      r10,r9,r16
  0x00401920: lui       r11,0x1000
* 0x00401924: addu      r11,r11,r16
```

If register `r25` had not been reused at `0x401908`, the recomputation of `(+ 0x1000 r16)` into `r11` would be detected.

1.3 Auditing Other Optimizations

Other compiler optimizations can be audited. Induction variable elimination auditing can track successive values assigned to a register by instructions within a loop. Tracking the values in a register without considering the instruction will miss induction variables if the register is subsequently reused for another purpose.

Dead code, with respect to a dynamic execution, is the instructions not run in a particular execution and is easy to find. However, a large fraction of a program may not run in any execution of a program, so this audit may not be helpful.

2 Validating Audit Reports

Compiler auditors produce a large number of reports. After an auditor examines a test program, the compiler writer needs to *validate* the reports by detecting places at which the compiler could have generated better code. Manually sifting through an auditor's complete output is extremely tedious. This section describes several techniques to partially automate the validation of audit reports. These techniques complement an auditor by using static analysis to classify the reports, so the compiler writer can concentrate on valid reports. The static validity criteria for audit reports are not always able to automatically distinguish valid and invalid reports. Even so, the criteria help reduce the number of invalid reports that must be manually considered.

An audit report can be classified into one of three categories:

- **Valid reports.** These report inefficiencies that are independent of the execution path through the program. As discussed above, these reports do not necessarily mean that the compiler could generate better code. For example, consider loop-invariant code in which the calculated expression is a single instruction. A compiler might (justifiably) recalculate the expression to avoid keeping a register occupied through the loop.[3]

- **Invalid reports.** These report inefficiencies that hold only for some executions of the audited program. Depending on the input, some paths in the program may never be taken and thus, in all realizable executions, an inefficiency might show up. However, as with any optimization, a compiler must be conservative and conclude that inefficiencies that do not occur along all paths are invalid.

[3] Of course, a compiler writer can adjust the loop-invariant auditor to report only expressions above a threshold size.

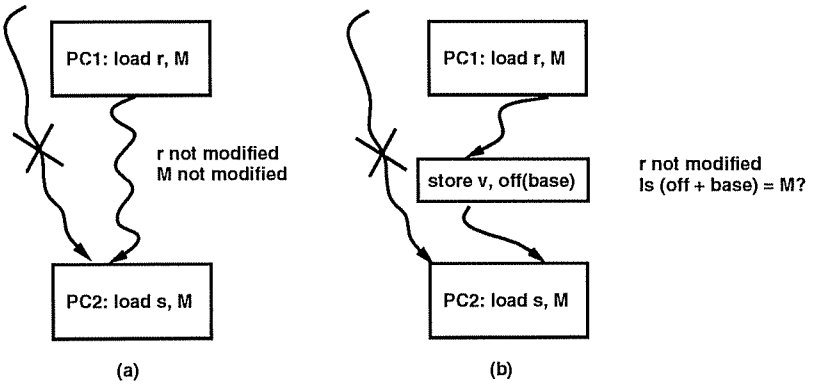

Figure 5: (a) Criteria for a valid report of redundant load. (b) Problem in automatically telling whether a report is valid.

- **Unnecessary reports.** Often, an auditor generates many reports on a single point in the program. For example, a common subexpression within a loop arises when the same operation is applied at two places (both within the loop) to the same operands. Each time around the loop, the operand registers hold different expressions. Many common subexpressions are detected and reported, but only one place in the assembly code need be changed. All but one of these reports is unnecessary, because the latter ones do not provide additional information.

Keep in mind that several factors mitigate the burden of invalid reports. First, short program executions suffice since a suboptimal code sequence only need execute once. Second, the compiler writer only must find the first valid report in the auditor's output. At this point, the compiler can be changed and the process reapplied.

In the discussion, program counters (e.g., PC1 and PC2) denote points in the control flow graph of the subroutine under consideration.

2.1 Redundant Loads

Consider an auditor report that the contents of memory location M is in register r at PC1 and r still holds M at PC2, at which point M is loaded into another register s (see Figure 5). The auditor will report that (for a particular execution), the load at PC2 is redundant, because M is already in r. This report is valid, if the following conditions hold:

1. PC1 dominates PC2 in the control flow graph,
2. r is not modified along any path from PC1 to PC2, and
3. M is not modified along any path from PC1 to PC2.

The first and second conditions can be checked automatically. The third condition is the problem. In general, it is impossible to determine if a store instruction on a path from PC1 to PC2 modifies location M. However, in a number of cases, we can obtain precise information about memory modifications. If only one path exists between PC1 and PC2, then the memory trace shows that M is not modified (or else, no report would be produced). Moreover, if no store instruction executes on *any* path from PC1 to PC2, the third condition trivially holds. Finally, we can make observations from the base-offset pairs in store instructions, but no general technique will

determine if a store instruction can modify M. Finding a dependence between the store and load instruction requires higher-level semantic information; for example, array indices from their respective statements [19].

Checking for partial redundancy [15] generalizes the validity test. PC1 may not dominate PC2, but r may hold the contents of M at PC2 along all paths from the entry point. This criteria also runs into a similar problem since any store instruction along a path to PC2 may modify M.

An audit report that violates either condition 1 or condition 2 is invalid. The compiler writer must examine the remaining reports to determine if condition 3 is violated. Redundant stores are treated similarly.

2.2 Redundant Spills

Consider a report that register r is spilled at PC1 and, later in the execution, at PC2 another register s is defined and not used between PC1 and PC2 (implying that s is dead at PC1). To validate this report, we must decide if s is actually dead along all paths after PC1. If so, the spill of r at PC1 is unnecessary and the report is valid. This property can be automatically checked.

Another problem is that many registers may be dead during the spill. The additional reports are not very useful, except to indicate the number of opportunities that the compiler missed. Simple postprocessing can combine these reports into a single report.

2.3 Redundant Common Subexpressions

Common subexpressions are similar to redundant loads. However, instead of checking that a memory location is not modified along all paths, we must check that inputs to an expression are not modified along all paths. Consider a report that register r contained expression E at PC1 and r still holds E at PC2, where E is recomputed. The report is valid if the following conditions hold:

1. PC1 dominates PC2,
2. r is not modified along any path from PC1 to PC2, and
3. Inputs to E are not modified along any path from PC1 to PC2

In general, E is an expression that includes memory locations, registers, and constants. E changes if any of its memory locations are modified. As discussed above, without a general method to determine that a store instruction does not affect a memory location along a path, condition 3 cannot be mechanically validated.

Again, the validity test can be generalized by checking for partial redundancy. But this again run into the same problem as in redundant loads. Also, as mentioned earlier, a common subexpression in which both PC1 and PC2 are in the same loop can generate many unnecessary audit reports that can be combined by simple postprocessing.

2.4 Loop Invariant code

Consider a report that expression E, computed at PC1, is loop invariant relative to the loop surrounding PC1 (see Figure 6). The report is useful if E is actually loop invariant and the calculation of E can be safely moved into a loop pre-header. The report is valid if all possible

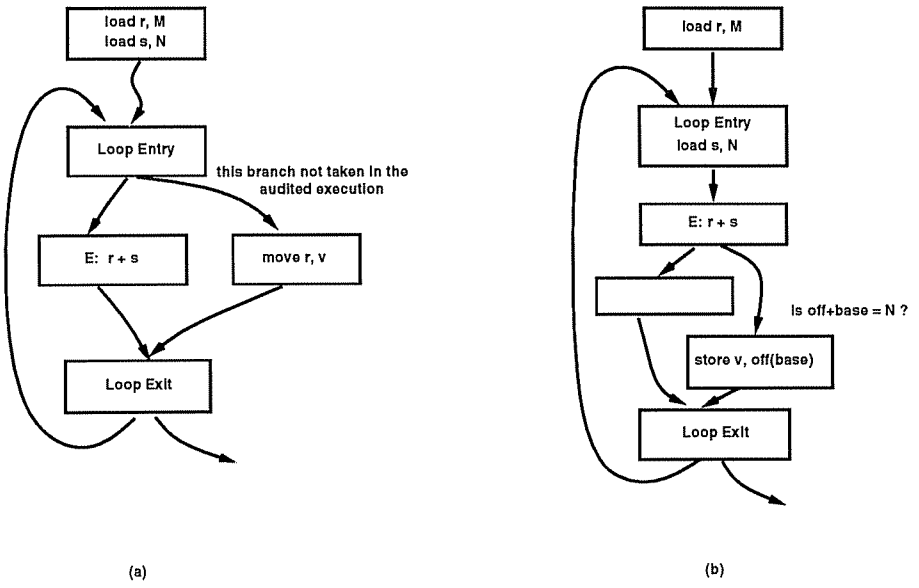

Figure 6: (a) A clearly invalid report of loop invariant expression. (b) An example in which it is difficult to automatically tell whether the report is valid.

executions of the loop result in a loop-invariant calculation at PC1. In general, it is impossible to determine this property automatically because of memory references. An operand of expression E may be a memory location. No general technique can determine if the location is modified in some execution of the loop. However, we can easily check if the code motion is profitable by testing whether PC1 dominates all loop exits [2].

2.5 Tools for Validation

The previous discussion shows that classification of auditor reports can only be partially automated. In the end, the compiler writer must examine the compiler's output at the points identified by an auditor. We developed a small set of utilities that are of immense help when examining the output. To quickly see which source instructions corresponding to a (dynamically) generated DAG, we provide a tool that computes a static slice (in assembly instructions) relative to a particular instruction. Other tools also produce a block-by-block listing of each subroutine's disassembled machine code, along with a graphic display of the control flow graph. These tools enable a compiler writer to quickly examine an audit report and determine whether it illustrates a flaw.

3 Empirical Evaluation

To evaluate the utility of compiler auditing, we implemented the register, common subexpression, and loop invariant auditors and tested them on low- and high-quality compilers. The hypothesis tested by these experiments is: if an auditor identifies a code sequence as suboptimal for one trace, it is suboptimal in general and illustrates a problem in the compiler. This section present the experimental results confirming this hypothesis.

Program	Lines	Instructions	Function
`compress` (C)	1,510	6,144	Unix compress utility
`matrix10` (F77)	439	15,360	10×10 matrix multiply
`tomcatv` (F77)	195	15,360	mesh generation program

Table 1: Test programs. `matrix10` is a adapted from the SPEC Benchmark `matrix300`. `tomcatv` is also a SPEC Benchmark, adapted to process a 17×17 grid. (C) indicates a C program. (F77) indicates a Fortran 77 program. Lines is the number of sources lines (excluding libraries). Instructions is the static number of machine instructions (including libraries) in the optimized version of the program.

3.1 Experimental Results

The tests were run on a DECstation 3100, which uses a MIPS R2000 processor and the MIPS C compiler and Fortran compiler (both version 4.1). The MIPS compilers are generally regarded to be high-quality compilers that perform most standard optimizations including common subexpression elimination, loop invariant code motion, and graph-coloring register allocation [7]. To test a high quality compiler (**high**), test programs were compiled at -O2, which performs all intraprocedural optimizations. To simulate a low quality compiler (**low**), the test programs were compiled at -O0, which performs no optimization. In all experiments, we did not audit library code to save time.

Table 1 lists the characteristics of the test programs. Each test program was run on a small input (to minimize the cost of tracing and auditing). The disassembled code was examined, by hand, to determine which reported deficiencies were due to missed compiler optimization and which were artifacts of executing a single path. Table 2 lists the number of potential defects uncovered (excluding the unnecessary reports; see Section 2) by the auditors and the fraction that a careful examination revealed to be suboptimal (valid reports). In general, since auditors produce many reports, we scrutinized reports arising from only the principle functions.

3.2 Analysis

The figures in Table 2 show that register and common subexpression auditing effectively uncover a manageable quantity of suboptimal code sequences and that, with two exceptions, at least 50% of these code sequences point to places at which the high-quality compiler could be improved. Loop invariant auditing in `compress` did not yield any valid reports; probably because `compress` is a carefully-written utility program, and its loop invariant expressions were removed at source level. As expected, more defects and a higher frequency of valid defects were found in the low-quality compiler.

A large fraction of the register and subexpression defects uncovered by the auditors arise from the interaction of the MIPS code generator and assembler. MIPS extends the underlying machine's instruction set with pseudo instructions that the assembler simulates by generating several actual instructions. Unfortunately, the assembler does a poor job of optimizing the resulting code (even within a basic block), so code sequences like:

```
lui r18,0x1000
lui r23,0x1000
lui r30,0x1000
```

Register Auditing					
		high		low	
Program	Portions examined	Defects	Valid Defects	Defects	Valid Defects
compress	*compress*(), *output*() (246 lines)	14	7 (50%)	40	31 (78%)
matrix10	*SAXPY* (9 lines)	4	2 (50%)	14	13 (93%)
tomcatv	*MAIN* (first 30 lines)	58	58 (58%)	58	58 (58%)
Common Subexpression Auditing					
		high		low	
Program	Portions examined	Defects	Valid Defects	Defects	Valid Defects
compress	*compress*(), *output*() (246 lines)	3	3 (100%)	4	4 (100%)
matrix10	whole program	25	17 (68%)	6	4 (67%)
tomcatv	whole program	13	5 (39%)	35	25 (71%)
Loop-Invariant Code Auditing					
		high		low	
Program	Portions examined	Defects	Valid Defects	Defects	Valid Defects
compress	*compress*(), *output*() (246 lines)	11	0 (0%)	5	0 (0%)
matrix10	whole program	30	25 (83%)	38	38 (100%)
tomcatv	whole program	34	19 (56%)	> 200	> 100

Table 2: Results of auditing the MIPS compiler at -O2 (**high**) and -O0 (**low**). Defects include both valid and invalid reports, but exclude unnecessary reports. Valid defects were verified, by hand, to be places at which the compiler could generate better (faster) code. To keep the task manageable, we examined only certain subroutines in the test programs. Also, each loop was limited to iterate at most five times (except in `compress`). The high accuracy for `tomcatv` is a result of examining a straight-line code sequence. Also, the identical number of high and low defects for this program is a coincidence. Finally, register auditing numbers are a bit inflated since the MIPS architecture does not have a `load_double` instruction, and the two loads that simulate it generate two reports.

```
addiu r30,r30,1320
addiu r23,r23,1288
addiu r18,r18,1904
```

occur, even in optimized code. This shortcoming is particularly noticeable in `compress`, which uses many global variables and data structures whose addresses are manipulated by unoptimized instructions.

The optimized version of `matrix10` contained more CSEs than the unoptimized version because the code for two unrolled loops (including one in *saxpy*!) contains many identical operation on induction variables that were introduced by unrolling. The number of defects detected in `tomcatv` is very large. The reason is that it consists of a single function, which has many nested loops and extremely large basic blocks. In the unoptimized code, many redundancies occur in the address calculation for two dimensional arrays. The high-quality compiler, on the other hand, missed many optimization opportunities due to loop unrolling.

In performing the audits described above, we also uncovered other shortcomings in the MIPS compiler. For example, large stack-allocated arrays sometimes forced variables to a point in the stack frame where they could not be referenced with a load or store instruction's 16-bit immediate. Placing scalar variables first in a stack frame would correct the problem.

4 Related Work

McNerney used an abstract interpretation to validate optimizations in a compiler for the Thinking Machines CM-2 [14]. He statically analyzed the compiler's intermediate form for a program to show equivalence between unoptimized and optimized code. Like this work, he did not prove properties about the compiler, but merely demonstrated the absence of errors for a particular input. Unlike this work, his tool demonstrated compiler correctness, not performance, and relied on static analysis, which restricted the programs that could be checked. The most serious restriction—which had little effect since the compiler had a similar restriction—was that loops had to be data independent. Compiler auditing, on the other hand, works for any program.

Boyd and Whalley [5] also developed a tool, *vpoiso*, for detecting errors in a compiler's optimizer. *vpoiso* automates a binary search for the optimization phase that performed a non-semantic-preserving transformation by selectively disabling optimizations and comparing the resulting program's output against the unoptimized program's output. Unlike compiler auditing, this tool detects correctness, but not performance errors, in a compiler. Obviously, tools that detect both types of errors are useful in developing a compiler.

5 Conclusions

Compiler auditing illustrates performance weaknesses in a compiler. By examining a trace of compiled code, auditors quickly exhibit suboptimal code sequences without the need for complex or expensive static analysis. Although these sequences are suboptimal for only one path, experience has shown that this property strongly indicates that the code is suboptimal in general and it should be examined by the compiler writer. Additional tools help a compiler writer sift through audit reports to detect valid ones.

Auditors will never replace compiler writers. However, information and feedback from tools of this sort can reduce the time and effort required to produce a mature, high quality compiler that generates efficient, as well as correct code. The approach is simple and requires a small investment of time and effort, which can have large benefits.

Acknowledgements

Harish Patil and Steve Kurlander provided helpful comments on this paper.

References

[1] Hiralal Agrawal and Joseph R. Horgan. Dynamic Program Slicing. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 246–256, June 1990.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1985.

[3] Thomas Ball and James R. Larus. Optimally Profiling and Tracing Programs. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 59–70, January 1992.

[4] William Blume and Rudolf Eigemann. Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks Programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):643–656, November 1992.

[5] Mickey R. Boyd and David B. Whalley. Isolation and Analysis of Optimization Errors. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 26–35, June 1993.

[6] Preston Briggs, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Coloring Heuristics for Register Allocation. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 275–284, June 89.

[7] Fred C. Chow and John L. Hennessy. The Priority-Based Coloring Approach to Register Allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, October 1990.

[8] Jack W. Davidson and Christopher W. Fraser. Code Selection through Object Code Optimization. *ACM Transactions on Programming Languages and Systems*, 6(4):505–526, October 1984.

[9] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A Practical Data Flow Framework for Array Reference Analysis and its Use in Optimization. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 68–77, June 1993.

[10] Rudolf Eigenmann, Jay Hoeflinger, Zhiyuan Li, and David Padua. Experience in the Automatic Parallelization of Four Perfect-Benchmark Programs. In Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*. MIT Press, August 1991.

[11] Susan Horwitz, Jan Prins, and Thomas Reps. On the Adequacy of Program Dependence Graphs for Representing Programs. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 146–157, January 1988.

[12] James R. Larus. Abstract Execution: A Technique for Efficiently Tracing Programs. *Software Practice & Experience*, 20(12):1241–1258, December 1990.

[13] James R. Larus. Efficient Program Tracing. *IEEE Computer*, 26(5):52–61, May 1993.

[14] Timothy S. McNerney. Verifying the Correctness of Compiler Transformations on Basic Blocks using Abstract Interpretation. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'91)*, pages 106–115, June 1991.

[15] E. Morel and C. Renvoise. Global Optimization by Suppression of Partial Redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.

[16] David Padua. email message. Personal Communication, November 1991.

[17] Jay M. Spitzen, Karl N.Levitt, and Lawrence Robinson. An Example of Hierarchical Design and Proof. *Communications of the ACM*, 21(12):1064–1075, December 1978.

[18] Richard M. Stallman. *Using and Porting GNU CC*. Free Software Foundation, September 1989.

[19] Steven W.K. Tjiang, Michael E. Wolf, Monica S. Lam, Karen L. Pieper, and John L. Hennessy. Integrating Scalar Optimization and Parallelization. In *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, pages C1–C16, Santa Clara, California, August 1991.