

Catching and Identifying Bugs in Register Allocation

Yuqiang Huang[†], Bruce R. Childers[†], and Mary Lou Soffa[‡]

[†]Department of Computer Science
University of Pittsburgh
Pittsburgh, Pennsylvania 15260
{yuqiangh, childers}@cs.pitt.edu

[‡]Department of Computer Science
University of Virginia
Charlottesville, Virginia 22904
soffa@cs.virginia.edu

Abstract. Although there are many register allocation algorithms that work well, it can be difficult to correctly implement these algorithms. As a result, it is common for bugs to remain in the register allocator, even after the compiler is released. The register allocator may run, but bugs can cause it to produce incorrect output code. The output program may even execute properly on some test data, but errors can remain. In this paper, we propose novel data flow analyses to statically check that the output code from the register allocator is correct in terms of its data dependences. The approach is accurate, fast, and can identify and report error locations and types. No false alarms are produced. The paper describes our approach, called SARAC, and a tool, called *ra-analyzer*, that statically checks a register allocation and reports the errors it finds. The tool has an average compile-time overhead of only 8% and a modest average memory overhead of 85KB.

1 Introduction

One of the most critical compiler transformations is register allocation, as a good allocator can make a dramatic difference in obtaining good performance [4, 11]. One study even reported that careful register allocation makes one order of magnitude difference in performance [26]! Thus, considerable effort has been given to developing new allocation algorithms or variants of existing ones [2-7, 11, 12, 24, 26, 28, 30]. Given the many algorithm variants and the complexity of modern architectures, implementing register allocation is often a complex and error prone task. Particularly, it is difficult to detect and locate bugs in an erroneous output of the allocator if the code runs to completion. Some efforts [13, 18, 21] have proposed techniques to ensure the allocator’s implementation is correct. In this paper, we describe a novel technique to check the correctness of register allocation and also to report the bugs. This technique is useful throughout the lifetime of a compiler, particularly during the development period.

Although a compiler undergoes much testing, bugs in the register allocator often slip past regression tests and are reported after release. What is worse is that many of these bugs cause the compiler to fail on some input programs, but not on others. The generated code may have bugs, although the compiler did not crash. Such latent bugs will not be discovered until a particular test input causes the program to fail. Assuming that a test input catches the bug, the developer is likely to believe that the bug is in the program itself, rather than the compiler. She will spend much time and effort tracking

down the bug to only discover that it is in the compiler and cannot be readily fixed. All of this leaves the developer in the unfortunate situation of having little confidence in the correctness of the generated code because bugs may remain even after testing.

The research community has recognized the difficulty of implementing compiler optimizations including register allocation and has proposed techniques to address the situation. Necula et al. [21] proposed a symbolic evaluation approach to check the allocator's output against the input. However, this approach reports false alarms and has four times compile-time overhead. Jaramillo et al. [13] proposed a dynamic checking approach that runs the allocator's input and output code. Then it compares the corresponding values to check that they are the same. However, it does not guarantee the correctness of the allocator's output unless *all* paths are exercised by test inputs.

In this paper, we propose a new approach, called SARAC, that uses *static analysis* to check the correctness of the allocator's output. SARAC reports the location and type of an error in the output due to an incorrect allocation. The analysis checks that the data flow semantics of the output match the semantics of the input. It traverses all program paths, using data flow analysis to gather information about the output. It then checks correctness using the gathered information. A checking step verifies that the data dependences of the input code are preserved in the output code, once the allocator has assigned registers and possibly spilled registers. The information collected during the analysis is used to determine error types and locations. Identifying errors in the dependences is a first step towards a complete tool for checking and reporting bugs.

Our approach does not produce false alarms and gives hints to the compiler engineer to help her diagnose and fix bugs in the allocator. Our analysis does not rely on knowledge about the allocator implementation; it can be used with different register allocation algorithms, including those that perform coalescing and rematerialization. It uses data flow techniques and can be easily implemented. Such independence from the register allocator suggests that a single error analysis tool can be built and employed for different allocators (in different compilers and target machines). Finally, the approach has minimal performance and memory overhead, making it efficient and practical. A prototype tool, called *ra-analyzer*, that implements SARAC has an average compile-time overhead of 8% and an average memory requirement of 85 KB.

This paper makes several contributions:

- A new way (SARAC) to statically check the correctness of a register allocator implementation and to identify and report the location and type of bugs, independently of the register allocator; no false alarms are generated.
- Techniques to support register allocators that perform coalescing, rematerialization and sub-register class allocation.
- The treatment of the register allocator as a black box. SARAC supports many allocator extensions, including live range splitting, interference region spilling, web splitting, spill coalescing, spill propagation and spill coloring.
- A tool (*ra-analyzer*) that implements SARAC in SUIF's back-end optimizer (MachSUIF [29]) for the Intel IA-32.
- An evaluation of *ra-analyzer*'s performance and memory overhead.

The next section describes how allocation preserves the semantics of the input code. The third section presents algorithms for gathering and using data flow information to

check for correctness. The fourth section evaluates *ra-analyzer*. The fifth section discusses related work and the final section concludes and describes future work.

2 Register Allocation

This section describes the motivation and background for our static analysis to catch and identify register allocation errors. To provide focus, we make several reasonable assumptions about the allocator. We assume that the allocator is not integrated with other optimizations (e.g. instruction scheduling) [3, 24], and it does not change the control flow graph, as is typical for register allocators. Initially, we assume a register allocator that does only allocation — e.g., it does not do coalescing or rematerialization. We also do not show address calculations. In a later section, we discuss how coalescing, rematerialization, sub-register class allocation and addresses can be incorporated. Lastly, we assume that the input code to the allocator is correct since we address register allocation errors.

When assigning *locations* (registers or memory) to hold *values* (variables or temporaries), a register allocator (e.g., on a RISC-style machine) can make only certain edits to the input code. One edit can change an input statement’s operand to a hardware register. Another edit is to insert store/load statements. A copy through a register might also be introduced. The edits take into account the data type and the target machine. For example, a floating point (FP) register should be used to hold a FP value and the appropriate register assignment made to a FP statement. Some target machines may require that specific hardware registers be used for certain operations. In this case, the register allocator has to ensure that its edits (and assignment) conform to the architectural constraints.

Figure 1 provides a running example, which counts the number of integer divisors for some number, n . The allocator’s input and output are shown in RTL notation [9]. RTL is a standard low level intermediate code representation used in various compilers (e.g., GNU gcc [10] and VPO [1]). In RTL, $r[n]$ is used to represent register n and $M[loc]$ is used to represent memory location loc . For example, $r[1]$ is register 1 and $M[c]$ is the memory location for variable c . A load is shown as $r[n]=M[loc]$ and a store as $M[loc]=r[n]$. A register-to-register copy is shown as $r[n]=r[m]$. Although our technique is not tied to a particular intermediate representation.

In the example, we assume that $r[1]$ is assigned by the allocator to hold variable n , and $r[2]$ is used to hold the other variables as necessary. However, two wrong allocation edits are made as shown in the incorrect output. The first wrong edit occurs at code point 8, where the wrong register has been assigned to the second source operand of the statement. The other incorrect edit is located at code point 12, where the wrong destination operand is used for the spill. The example also shows the locations where the errors are manifested. The location where an error is manifested is not necessarily the location where the wrong edit is made. For example, the erroneous edit at 12 is manifested as error 2 and 3 at code point 11 and 14, respectively.

2.1 Data Flow Semantics and Register Allocation

A semantically correct allocation of registers must preserve the input code’s semantics, particularly the data dependences. Thus, variable and temporary definition and use pairs

Source Code	Input to Allocator
<pre> /*count number of divisors to variable n that is passed as an argument*/ c=0; for (d=1; d<=n; d++) { if (n%d == 0) c++; } </pre>	<pre> 1:c=0; 2:d=1; 3:PC=((n<=0)?L3:PC+4); L1: 4:t=n%d; 5:PC=((t!=0)?L2:PC+4); 6:c=c+1; L2: 7:d=d+1; 8:t=d<=n; 9:PC=((t==1)?L1:PC+4); L3: </pre>
Correct Output from Allocator	Incorrect Output from Allocator
<pre> 1:r[1]=M[n]; 2:r[2]=0; 3:M[c]=r[2]; 4:r[2]=1; 5:M[d]=r[2]; 6:PC=((r[1]<=0) ? L3:PC+4); L1: 7:r[2]=M[d]; 8:r[2]=r[1]%r[2]; 9:PC=((r[2]!=0)?L2:PC+4); 10:r[2]=M[c]; 11:r[2]=r[2]+1; 12:M[c]=r[2]; L2: 13:r[2]=M[d]; 14:r[2]=r[2]+1; 15:M[d]=r[2]; 16:r[2]=r[2]<=r[1]; 17:PC=((r[2]==1)?L1:PC+4); L3: </pre>	<pre> 1:r[1]=M[n]; 2:r[2]=0; 3:M[c]=r[2]; 4:r[2]=1; 5:M[d]=r[2]; 6:PC=((r[1]<=0) ? L3:PC+4); L1: 7:r[2]=M[d]; 8:r[2]=r[1]%r[1]; err1: wrong reg 9:PC=((r[2]!=0) ? L2:PC+4); 10:r[2]=M[c]; 11:r[2]=r[2]+1; err2: stale (c) 12: M[d]=r[2]; wrong store(causes err2,3) L2: 13:r[2]=M[d]; 14:r[2]=r[2]+1; err3: eviction (d) 15:M[d]=r[2]; 16:r[2]=r[2]<=r[1]; 17:PC=((r[2]==1) ? L1:PC+4); L3: </pre>

Figure 1: Example source, input to register allocator, correct and incorrect output code

(“du-pairs”) in the input should be maintained in the output. We define a “*du-pair*” notationally as $(p.x = , q. = x)$, where the definition of the variable or temporary x at code point p reaches the use of x at q . A code point is a label on a statement in the input or output. For example, in the allocator’s input of Figure 1, the variable c is defined at code point 1 and used at code point 6, giving the du-pair $(1.c = , 6. = c)$.

After register allocation, there is not necessarily a one-to-one correspondence between the input du-pairs (involving variables and temporaries) and the output du-pairs (involving registers and memory locations). The allocator can insert loads, stores or copies to move values between the registers and memory. The output correspondence of an input du-pair is termed a “*du-sequence*”:

A du-sequence $(s.d = , \dots, t. = u)$ is a chain of du-pairs such that d holds the value v at s , u holds the same value v at t , and there is a connected chain of du-pairs starting at s and ending at t that can register copy, load, or store the value v .

A du-sequence can perform a number of moves; a typical du-sequence has no moves or one store and reload. For example, there is du-sequence $(2.r[2] = , 3.M[c] = r[2] , 10.r[2] = M[c] , 11. = r[2])$ in the correct output of Figure 1. The

notation $3.M[c]=r[2]$ shows a store at code point 3. Similarly, $10.r[2]=M[c]$ shows a load at 10.

When the allocator correctly maintains the data flow of the input, each input du-pair has a corresponding output du-sequence, where the start of the du-sequence maps to the definition in the du-pair and the end of the du-sequence to the use of the du-pair. Thus, a combination of propagation and substitution is used to recover the du-pair from the du-sequence. For example, in Figure 1 the correct output code points 2 and 11 map to input code points 1 and 6, and $2.r[2]=$ corresponds to $1.c=$ and $11.=r[2]$ to $6.=c$. Hence, the input du-pair $(1.c=, 6.=c)$ corresponds to the du-sequence $(2.r[2]=, 3.M[c]=r[2], 10.r[2]=M[c], 11.=r[2])$. The input du-pair can be recovered by propagation and substitution as shown in the steps:

1. $(2.r[2]=, 3.M[c]=r[2], 10.r[2]=M[c], 11.=r[2])$ // Initial du-sequence
2. $(2.r[2]=, 10.r[2]=r[2], 11.=r[2])$ // After propagation of $r[2]$
3. $(2.r[2]=, 11.=r[2])$ // After propagation of $r[2]$ again
4. $(1.c=, 6.=c)$ // Final du-pair after c was substituted for $r[2]$

When a use has multiple reaching definitions, all defined values need to be in the same register (or memory location) before the use. For example, the use $6.=c$ has the reaching definitions $6.c=$ and $2.c=$ in the allocator's input of Figure 1. These are maintained in the correct output as $(11.r[2]=, 12.M[c]=r[2], 10.r[2]=M[c], 11.=r[2])$ and $(2.r[2]=, 3.M[c]=r[2], 10.r[2]=M[c], 11.=r[2])$.

Thus, the input and output have the equivalent data flow semantics if and only if the input's du-pairs can be recovered from the output's du-sequences. Hence, we use the "recovery" process to check the correctness of an allocation. Because of the "recovery" process, there are no false positive for our techniques.

2.2 Sources of Errors

A bug in the allocator that causes the output program to crash or produce a wrong result (but not the compiler) is manifested through incorrect code edits that can be made by the allocator. For a register allocator, the incorrect edits are:

1. *incorrect register assignment*: the wrong register is used for an operand;
2. *wrong store or load*: a value is stored or loaded incorrectly (the store or load may be redundant or it may use the wrong memory address for a variable or temporary);
3. *missing store or load*: a value is not spilled or reloaded when needed;
4. *wrong register type*: the wrong type is used (e.g., a load-byte statement is used when a load-word statement is needed);
5. *constraint violation*: specific architectural constraints are violated.

These edits can violate the semantics of the input code and affect data dependences. The first three edits can cause the du-sequences in the output code to have no correspondence with the input du-pairs. These incorrect edits can challenge the compiler engineer to detect. We focus on these edits as an important and necessary step to catch and report bugs in an allocation. Both the wrong register type and constraint violation edits usually preserve the correct data dependence. Our algorithms can be extended to automatically check these using a linear inspection of the input and output.

An incorrect edit can lead to errors in the program. An error happens when a du-pair in the input cannot be recovered from the allocator’s output. *We define an error as a violation of the input code’s data flow.* Note the distinction between an “incorrect edit” and an “error”: An incorrect edit is the cause of an error. The incorrect edit defines where something was done wrong to the code, but it is not necessarily the code point where the error is exposed. An incorrect edit may not manifest itself as an error until a value affected by the edit is used. For instance, in Figure 1 the wrong edit at code point 12 is not exposed until code points 11 and 14. In fact, an incorrect edit can be made that does *not* cause an error in the program. For example, when a duplicate load is inserted, it may do no harm in terms of the program’s data flow. Our concern is *incorrect edits that cause the program to fail—crashing or computing a wrong value—by disobeying the input code’s data flow.*

The incorrect edits can lead to three error types: *stale value error*, *wrong operand error*, or *eviction error*. Although these errors all involve data flow, we distinguish between them to report causal information about what went wrong. A *stale value error* happens when referring to a register or memory location that holds an old version of the needed value. A wrong or missing store is a common cause. For example, the incorrect output of Figure 1 shows that the wrong store is generated and that `r[2]` is spilled to `M[d]`, rather than to `M[c]`. Thus, there is no du-sequence for `c` along the loop back edge that reaches the use at code point 11. Consequently, a stale value for `c` is used. Equivalently, the input du-pair $(6.c = , 6. = c)$ cannot be recovered. A *wrong operand error* occurs when referring to a register or memory location that does not hold the needed value at all. The value is actually held in some other location(s). This error is usually caused by an incorrect register assignment. An *eviction error* occurs when referring to a value that is not held in any location at all. This error is usually caused by an wrong store. Figure 1 shows examples for both wrong operand and eviction errors.

3 Error Analysis for Register Allocation

To find register allocation errors, we develop a technique, called SARAC (Static and Automatic Register Allocation Checking) that includes mapping generation and data flow analysis. The technique *implicitly* and *efficiently* gathers information about the du-pairs and du-sequences to ensure that the du-pairs in the allocator’s input code match the du-pairs recovered from the du-sequences in the allocator’s output code. As most register allocators operate at the procedural level, SARAC uses the code generated for a procedure. The technique is also applicable to local register allocation and can be extended to interprocedural register allocation [28].

```
SARAC(input,output) {
  Map map = mapGen(input,output); // Step 1: mapping generation
  Dataflow sets = defAnalysis(map,output); //Step 2: dataflow analysis
  errAnalysis(output,map,sets); //Step 2: check the allocation
}
```

Figure 2: SARAC steps

The three steps of SARAC are shown in Figure 2. First, mapping information is generated using the allocator’s input and output. Then, iterative forward data flow analysis, called `defAnalysis`, is performed on the output using mapping information.

This analysis collects three types of data flow sets needed to check the correctness of the output and report error locations and types. Finally, a linear scan, called `errAnalysis`, exposes def-use violations.

3.1 Step 1: Mapping Generation (`mapGen`)

SARAC needs to know which value (of the original operand) in the input is actually defined/used by the output. Therefore, a mapping or association is determined that relates an operand in the output to its corresponding operand in the input. Intuitively, a location (register or memory) in the output is mapped to the corresponding value (variable or temporary) in the input. A mapping can also relate constants in the output and input. Mappings are generated for all necessary statements, including statements in the function prologue and epilogue. For load, store or register copy statements injected by the allocator, there is no corresponding statement in the input. Thus, no mapping is generated for these statements. For each of the other statements in the output code, there is a corresponding statement in the input.

```

mapGen(input,output) {
  Map map := ∅;
  // get blocks in same order for traversal
  Blocks Bin[] := canonicalOrder(input);
  Blocks Bout[] := canonicalOrder(output);
  Block Bi := Bin.getNextElement();
  Block Bo := Bout.getNextElement();
  while (Bi≠null) {
    // create maps for stmts in input and output
    foreach Statement Si∈Bi {
      Statement So := find(Si, Bo);
      if (So≠null)
        // map all (*) opers in So to opers in Si
        map := map∪{So.*→Si.*};
    }
    Bi := Bin.getNextElement();
    Bo := Bout.getNextElement();
  }
  return map;
}

```

Figure 3: Pseudocode for mapping generation

As shown in Figure 3, `mapGen` generates mappings based on the allocator’s input and output, where the allocator is viewed as a black box. First, the basic blocks in the input and output code are put in a canonical order. Next, the input blocks are traversed. For each input statement, the corresponding output statement (if present) is found in a basic block by `find`. Finally, the operands in the output statement are mapped to operands in the input statement. In the figure, the notation “*” means “any” (e.g., all operands). Although a mapping includes information about statement and operand number, an abbreviation (e.g., *location*→*value*) is used in the paper. For example, the output code in Figure 1 has statement `r[2]=0` corresponding to the input statement `c=0`. Thus, the mappings are `r[2]→c` and `0→0`.

3.2 Step 2: Data Flow Analysis (`defAnalysis`)

To check if the register allocation is correct and to determine error locations and types,

`defAnalysis` needs to gather information about the behavior of the register allocator using the output code and the mappings. `defAnalysis` gathers three types of information at all points in the program: (1) the values that are currently held in locations (registers and memory), (2) the stale values and (3) the evicted values. Note if we only wanted to know if a register allocation is correct, we would not need the eviction information. We develop a data flow algorithm to gather the information by using the mappings to get the values in the input code associated with locations in the output code. For example, when $r[2]=1$ at output code point 4 in Figure 1 is processed, the original destination operand d is retrieved from the mappings. This gives three pieces of information. First, the current value of d is defined in $r[2]$. Second, the value c in $r[2]$ is evicted. Finally, any previous values of d in other locations become stale.

These three types of information are collected in three data flow sets — the Location set (L), Stale set (ST) and Eviction set (E). Each set consists of triples $\langle l, v, c \rangle$, where l is a location (register or memory) from the output code, v is a value (name) from the input code or another location from which the value can be found, and c is a vector consisting of a series of code points where the relationship between l and v occurred. Thus, the semantics of $\langle l, v, c \rangle$ for L, ST and E are defined as follows.

- L records the fact that location l holds v . The vector c records the du-sequence for v (as a series of code points involved in the sequence).
- ST records that location l holds a stale v due to a series of code points in c , where a value has been killed because of a new definition at the start of that series.
- E records that v has been evicted from location l at a statement in c . For E, c is always a vector with a single element.

3.2.1 Data Flow Equations

A statement S in the output code can either be a statement passed from the input with registers assigned or a copy statement introduced by the register allocator. We use O to represent original statements and l_d to represent the destination of the statement in the output code. We use C to represent copy statements. A copy statement is either a load, store or register copy inserted by the allocator. Thus, S has the formats:

$$O: l_d = \text{exp} \{ \text{original statement} \} \quad \text{or} \quad C: l_d = l_s \{ \text{copy statement} \}$$

We now describe each set's Gen, Kill, IN and OUT. In a basic block, each set's IN for a statement is its OUT from the immediately preceding statement. The merge points are described separately for each set. The three sets are computed in the same phase.

Our data flow equations extend the traditional data set operations mostly because of the third element of the triple, c , which is an *ordered* set. The elements of c are a set of code points that are used to compute the du-sequence as data flow proceeds. We redefine \cap and $-$ to handle the set c . We also define other operators to propagate the value along du-sequences and to produce a new triple.

Definition of \cap :

$$\langle l, v, c \rangle \cap \langle l', v', c' \rangle = \begin{cases} \{ \langle l, v, c \rangle, \langle l', v', c' \rangle \} & \text{if } l=l' \wedge v=v' \wedge c \neq c' \\ \langle l, v, c \rangle & \text{if } l=l' \wedge v=v' \wedge c=c' \\ \emptyset & \text{otherwise} \end{cases}$$

Definition of $-$:

$$\langle l, v, c \rangle - \langle l', v', c' \rangle = \begin{cases} \emptyset & \text{if } l == l' \wedge v == v' \\ \langle l, v, c \rangle & \text{otherwise} \end{cases}$$

These two operators are similar to the normal set operators on the first two elements in the triple. The third element c is handled in a special way.

Computing the Location Set (L)

$$L_gen[S] = \begin{cases} \langle l_d, v, \langle S \rangle \rangle & \text{if } S \in O \wedge l_d \rightarrow v \\ \langle l_d, l_s, \langle S \rangle \rangle & \text{if } S \in C \end{cases}$$

There are two cases for $L_gen[S]$. The first case occurs when a statement S in O defines a new value in l_d . The location l_d must be mapped to a value v . Therefore, a triple “ $\langle l_d, v, \langle S \rangle \rangle$ ” is generated. For example, when $r[2]=1$ at code point 4 in Figure 1 is processed, a triple “ $\langle r[2], d, \langle 4 \rangle \rangle$ ” is generated. The second case happens to a statement S in C , which does not define a new value but copies a value. The value to copy is in l_s . “ $\langle l_d, l_s, \langle S \rangle \rangle$ ” is generated to indicate that the value will be found at l_s when applying the value propagation. For example, when $M[d]=r[2]$ at code point 5 in Figure 1 is processed, a triple “ $\langle M[d], r[2], \langle 5 \rangle \rangle$ ” is generated to show that the value in $M[d]$ can be found from $r[2]$.

$L_kill[S]$ considers that the execution of S destroys the value in l_d :

$$L_kill[S] = \langle l_d, *, * \rangle$$

This Kill computes the triple indicating any value held in the destination of S .

For the value propagation (i.e., collapsing C statements in a du-sequence), the operator \oplus is defined.

Definition of \oplus :

$$\langle l', v', \langle S_1, \dots, S_i \rangle \rangle \oplus \langle l, v, \langle S \rangle \rangle = \begin{cases} \langle l, v, \langle S \rangle \rangle & \text{if } S \in O \\ \langle l, v', \langle S_1, \dots, S_i, S \rangle \rangle & \text{if } S \in C \wedge l' == v \\ \emptyset & \text{otherwise} \end{cases}$$

This operator just returns the right hand side triple if S is in O . If S is in C , then there are two cases. First, the value propagation along a du-sequence is performed if l' is v and vector $\langle S_1, \dots, S_i \rangle$ appended with S is the third element of the result triple. Second, the value of null is returned if l' is not v .

Given the Gen, Kill and IN sets, $L_out[S]$ is computed as:

$$L_out[S] = (L_in[S] \oplus L_gen[S]) \cup (L_in[S] - L_kill[S])$$

$L_out[S]$ has all the locations (registers and memory) that hold a value, regardless of whether it is current or stale. When $M[d]=r[2]$ at code point 5 in Figure 1 is processed, $L_in[5]$ has “ $\langle r[2], d, \langle 4 \rangle \rangle$ ” and $L_gen[5]$ consists of “ $\langle M[d], r[2], \langle 5 \rangle \rangle$ ”. The triple “ $\langle M[d], d, \langle 4, 5 \rangle \rangle$ ” is computed from “ $L_in[5] \oplus L_gen[5]$ ”. This triple shows that $M[d]$ holds value d after code point 5, which was computed at code point 4 and propagated at code point 5.

At the merge point to block B , L_in is:

$$L_in[B] = \cap L_out[Predecessors(B)]$$

L_in is computed by \cap on L_outs of all predecessors to B . A correct register allocation puts the same value in the same location along any preceding path for a later use of that value from that location. Therefore, \cap removes the “inconsistent triples” which have different values in the same location.

Computing the Stale Set (ST)

$$ST_gen[S] = L_gen[S]$$

$ST_gen[S]$ is the same as $L_gen[S]$ though its two cases have different semantics. First, when S in O defines a new v in l_d (where $l_d \rightarrow v$), every previous v held in some other locations (not l_d) becomes stale. Which locations holding v will be discovered from $L_in[S]$ later on. Second, S in C is considered. $ST_gen[S]$ is computed using a place holder l_s (i.e., the source of S) to represent the actual value. If l_s holds a stale value, l_d also holds a stale value after the value propagation.

$ST_kill[S]$ is computed similar to $L_kill[S]$:

$$ST_kill[S] = \langle l_d, *, * \rangle$$

When a stale value in l_d is destroyed by S , this fact must be reflected in $ST_kill[S]$.

The operator \bullet is defined for finding stale values.

Definition of \bullet :

$$\langle l', v', * \rangle \bullet \langle l, v, \langle S \rangle \rangle = \begin{cases} \langle l', v, \langle S \rangle \rangle & \text{if } S \in O \wedge l' \neq l \wedge v' = v \\ \langle l, v, \langle S \rangle \rangle & \text{if } S \in C \\ \emptyset & \text{otherwise} \end{cases}$$

The first case applies to S in O . Any other location l' (i.e., $l' \neq l$) that holds v' (i.e., $v' = v$) is discovered and a new triple “ $\langle l', v, \langle S \rangle \rangle$ ” is produced. The second case applies to S in C and the right hand side triple is simply returned. The last case yields null.

$ST_out[S]$ is computed as:

$$ST_out[S] = (ST_in[S] \oplus (L_in[S] \bullet ST_gen[S])) \cup (ST_in[S] - ST_kill[S])$$

For S in O , “ $L_in[S] \bullet ST_gen[S]$ ” computes the triples where v in any location other than l_d becomes stale because S defines new v in l_d . For example, when $r[2] = r[2] + 1$ at code point 11 in Figure 1 is processed, $ST_gen[11]$ consists of “ $\langle r[2], c, \langle 11 \rangle \rangle$ ”. The triple “ $\langle M[c], c, \langle 2, 3 \rangle \rangle$ ” is retrieved from $L_in[11]$. “ $L_in[11] \bullet ST_gen[11]$ ” produces “ $\langle M[c], c, \langle 11 \rangle \rangle$ ”. For S in C , “ $\langle l_d, l_s, \langle S \rangle \rangle$ ” is computed from \bullet operation and “ $ST_in[S] \oplus (L_in[S] \bullet ST_gen[S])$ ” does the stale value propagation. For example, “ $ST_in[10] \oplus (L_in[10] \bullet ST_gen[10])$ ” produces “ $\langle r[2], c, \langle 11, 10 \rangle \rangle$ ”, which shows that the previous c became stale at code point 11 and propagated to $r[2]$ at code point 10 along the loop back edge.

At the merge point to block B , ST_in is:

$$ST_in[B] = \cup ST_out[Predecessors(B)]$$

ST_in is computed by the union on ST_outs of all predecessors to B . The union is done because if the value is stale along any path to the block, it is possible that the stale value might be used in the current (or later) block. Hence, the union operation preserves the fact that the value is stale along some path.

Computing the Eviction Set (E)

The equations for E are closely related to the ones for L .

$$E_gen[S] = \langle l_d, *, \langle S \rangle \rangle, \quad E_kill[S] = L_gen[S]$$

$E_gen[S]$ records that any value in l_d will be evicted because of S . But which value is actually evicted must be discovered from $L_in[S]$. $E_kill[S]$ is the same as $L_gen[S]$.

To obtain the value currently held in a location (e.g., l_d) and then indicate that it is evicted from there, the operator \diamond is defined and its semantics is self-explanatory.

Definition of \diamond :

$$\langle l', v', * \rangle \diamond \langle l, *, \langle S \rangle \rangle = \begin{cases} \langle l', v', \langle S \rangle \rangle & \text{if } l' = l \\ \emptyset & \text{otherwise} \end{cases}$$

$E_out[S]$ is computed as:

$$E_out[S] = (E_in[S] \cup (L_in[S] \diamond E_gen[S])) - (L_in[S] \oplus E_kill[S])$$

The operator \diamond discovers the value evicted by S from l_d with the computation “ $L_in[S] \diamond E_gen[S]$ ”. “ $L_in[S] \oplus E_kill[S]$ ” gives the triples that a value is put into l_d by S .

At the merge point to block B , E_in is computed as:

$$E_in[B] = \cup E_out[Predecessors(B)]$$

$E_in[B]$ holds any value’s history of being most recently evicted from any location along all preceding paths.

3.3 Step 3: Checking and Reporting (errAnalysis)

```

errAnalysis(output, map, sets) {
  L:=sets.L; ST:=sets.ST; E:=sets.E;
  foreach Block B eoutput {
    if (B≠Binitial)
      setFinalization(B, map, L, ST, E);
    foreach Statement S eB {
      typeCheck(S, map);
      constraintCheck(S, map);
      if (S e O)
        useCheck(S, map, L, ST, E);
    }
  }
  setFinalization(B, map, L, ST, E) {
    L_union := ∪ L_out[Predecessors(B)];
    L_inconsistent := {<l, v, <B> |
      ∀<l', v', *> e (L_union L_in[B])};
    ST_in[B] := ST_in[B] L_inconsistent;
    E_in[B] := E_in[B] ∪ L_inconsistent;
    computeLocalFlow(B, map, L, ST, E);
  }
}

useCheck(S, map, L, ST, E) {
  foreach leuses(S) {
    v := getMap(S, l, map);
    if (<l, v, *> e L_in[S]) {
      if (<l, v, c> e ST_in[S]) {
        ε := "S uses stale value,
          c made v in l stale";
      } else ε := null;
    } elseif (<l', v, c> e L_in[S]) {
      ε := "S uses wrong operand,
        but c defined v in l'";
    } else {
      ∀<l'', v'', c> e E_in[S];
      ε := "S uses evicted value,
        c evicted v from l''";
    }
  }
}

```

Figure 4: Pseudocode for checking algorithm

Once L , ST and E are collected, they are used to check the output code. The error analysis step ensures that the du-pairs from the input are preserved in the output. The algorithm for identifying and reporting errors is shown in `errAnalysis` in Figure 4.

For non-initial blocks, a finalization step is performed on the data flow sets by `setFinalization`. The finalization is actually done in `defAnalysis`, but we show it here for clarity. It computes $L_{inconsistent}$ – the “inconsistent triples” where the values in the same location are different for different paths. These triples are not computed into L_{in} because a correct register allocation should put the same value into the same location for any path. To report causes rather than just check errors, we assume that the inconsistent values (in the same location) are “evicted” at the merge. Therefore, $L_{inconsistent}$ is added to E_{in} and removed from ST_{in} . Finally, local data flow sets are updated by the equations discussed in Section 3.2.1.

The next step in `errAnalysis` iterates over all the statements. First, the operands of the output are verified that they have the correct types as specified by the input. Second, it verifies that architectural constraints are satisfied with `constraintCheck`, which depends on the target architecture (not shown for brevity). Finally, `useCheck` applies to O statements (C statements are implicitly checked because of the value propagation performed in `defAnalysis`).

`useCheck` checks that all uses in every O statement are correct in terms of the input’s data flow. It reports the error location and type for any data flow violation. For each use l (i.e., location), it first consults the mappings to determine which value it should use. When l actually holds v , which is shown as a triple “ $\langle l, v, * \rangle$ ” in L_{in} , it further checks if v in l is stale. Next, it checks if v is in other locations. If this is true, it implies that the wrong operand might be used. Otherwise, an eviction error must have occurred. The history of v being most recently evicted from any location l' is reported.

3.4 Extensions

Two important extensions to a register allocator are coalescing and rematerialization [4, 5, 6, 11]. This section describes how SARAC can support these extensions. It shows how sub-register class allocation and address expressions are incorporated.

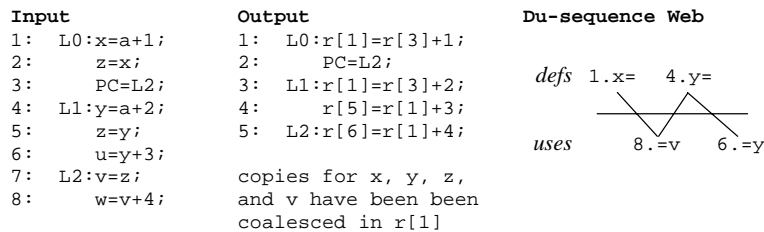


Figure 5: Register coalescing example and its du-sequence web

Register Coalescing. Register coalescing removes unnecessary copies from the input code. As shown in Figure 5, the copies at input code points 2, 5, and 7 for z are removed in the output. Thus, $r[1]$ can hold x , y , z or v ; a location can correspond to multiple values. The analyses described earlier rely on a one-to-one mapping between locations and values and consequently cannot directly handle coalescing.

To support coalescing, SARAC needs to handle the effect of removing copies. SARAC infers coalescing by examining the du-sequences in the *input code* and updating the mappings to capture all possibly coalesced values. The idea is to use a “*du-sequence web*” to capture the relationship between a definition that begins a

du-sequence and a use that ends the sequence. We define a *du-sequence web* as a set of du-sequences sharing a start or end, where the copy statements in each du-sequence are collapsed. There may be many independent webs for the input code, each corresponding to a set of related du-sequences. The most right column of Figure 5 shows a web for the input code. In this web, the du-sequence (1.x=, 2.z=x, 7.v=z, 8.=v) is represented by the edge between 1.x= and 8.=v. The web also captures the relationships among the du-sequences (4.y=, 5.z=y, 7.v=z, 8.=v) and (4.y=, 6.=y).

The webs are used to update the mappings. Once the webs are constructed, each web is assigned a unique name, say n . Then, the name in the mappings for the web's definitions and uses are changed to n . In the example, $r[1] \rightarrow x$ (where, $r[1]$ is the destination of $r[1]=r[3]+1$) is changed to $r[1] \rightarrow n$. Any input code copy that is actually not coalesced is also considered as C statement besides the copies injected by the register allocator. Thus, the mappings for any copy statement passed from the input to output are removed. With the updated mappings, `defAnalysis` and `errAnalysis` are performed normally. In `defAnalysis`, the value n is propagated along the output du-sequence. In `errAnalysis`, only the uses in a du-sequence web are analyzed.

Rematerialization. Rematerialization improves spill code by recomputing values rather than reloading them from memory. It usually considers constant expressions in the code, such as integer constants in load-immediate statements and address offsets.

To handle rematerialization, the mappings are extended to bind constants to values and locations. The idea is to bind constants in the input and output code to values and locations in the mappings. The bindings are created by scanning the output code to find uses of constant expressions (i.e., the use is reachable by a constant definition, like a load-immediate). A similar step is performed to bind constants to values in the input code. `errAnalysis` compares a location that is bound to a constant to the corresponding value's binding. If the constants match, then the output code is correct.

Sub-register Class Allocation. Some architectures allow different registers to overlap. For instance, the IA-32 has the AH and AL registers, which overlap a part of the AX register. Such overlapping registers are a "register alias set" [30] and an allocator has to take into account the overlap when assigning registers. A write to a register will destroy the value in any member of its alias set.

To handle sub-register class allocation, only modest modifications are needed to SARAC's data flow equations at several points. The equations have to be changed to take into account the effect on the full register alias set. For example, when $L_kill[S]$ is computed, the register alias set of l_d is considered, rather than just l_d .

Address Generation. Some allocators determine an effective address (rather than a variable or temporary name) for spilling a value. In this case, this address is typically computed as an offset from the stack pointer. In SARAC, a "memory location" is the effective address used in a store/load. Assuming that the allocator makes only the edits described earlier, there can be no intervening manipulation of the stack pointer between a store and an associated load. That is, the allowable edits do not permit the insertion of statements that change the stack pointer (except in the function prologue and epilogue). Thus, the effective addresses can be easily determined. When the allocator directly

manipulates the stack pointer, SARAC determines an address by evaluating the operations done to the stack pointer and offset.

4 Experiments

We implemented SARAC as a tool (*ra-analyzer*) for SUIF’s backend code optimizer (MachSUIF, version 2.02.07.15), on the Intel IA-32 [29]. A global graph coloring register allocator [11] was implemented as a separate pass in MachSUIF. *ra-analyzer* is run after register allocation. Two experiments were conducted. First, faults were injected into the allocator’s output to explore how the tool might be used to find bugs. Second, the performance and memory overhead of the tool were measured.

For the experiments, we used benchmarks in SPECint2K [8], MediaBench [15] and MiBench [19] that are compilable by base SUIF. The procedures in the benchmarks span a wide range of code sizes and complexities. All experiments were run on a RedHat Linux computer with a 2.4 GHz Pentium 4 and 1 GB RAM.

4.1 Fault Injection

We checked if MachSUIF’s allocator causes errors in the benchmarks and found no errors for two possible reasons. First, MachSUIF’s allocator is correct. Second, a very limited number of test suites (many benchmarks cannot be compiled by SUIF) may not expose all latent bugs. Thus, we believe that *ra-analyzer* is particularly useful in a regression testing environment or during the development of a compiler.

To illustrate how *ra-analyzer* might be used by compiler engineers, we injected bugs into the output of MachSUIF’s allocator. We then used *ra-analyzer* to find the bugs. The bugs were automatically injected by a “fault injector”. The fault injector made incorrect edits to the output code, including incorrect register assignment, wrong store/load, missing store/load. For each edit type, the fault injector randomly selected a basic block to change. An appropriate statement was found to modify, based on the edit type. If an appropriate statement could not be located, the edit was abandoned and a new one was tried. The injector attempted to make 5 changes for each edit type, but it sometimes made fewer edits when it could not find a candidate. Each function in every benchmark had 0 to 25 incorrect edits.

As an example, the fault injector changed one register operand to a different register in the FFT benchmark. In this case, the statement `movl $1, %ecx` was changed to `movl $1, %ebx`. The register `%ecx` holds the virtual register `$vr12`. When *ra-analyzer* checked the code, it reported the error message:

```
addl %ecx,%eax
//Wrong operand - %ecx,"movl $1,%ebx" defined $vr12 in %ebx
```

From the error message, compiler engineers can identify what went wrong. For example, consistently using the wrong register might suggest that liveness analysis or the interference graph construction has a problem. With the information from *ra-analyzer*, compiler engineers can use a debugger to step through the allocator and find bugs.

In the fault injection experiments, 65 to 10,749 total incorrect edits were made to the benchmarks. The simpler programs (e.g., *FFT*) had the fewest edits, while the more complex ones (e.g., *255.vortex*) had the most. Of the total edits, there were 22– 3,198

incorrect register assignment edits, 29–5,104 wrong store/load edits, and 7– 2,447 missing store/load edits. The edits made covered the possible changes to the code described in Section 2.2. The edits lead to a total of 108–18,103 errors. There were 18–2,648 stale errors, 49–7,552 wrong operand errors and 35–7,903 eviction errors. When *ra-analyzer* was applied on the code, it correctly caught the errors without generating any false positives or negatives, and reported their locations and types.

4.2 Performance and Memory Overhead

Benchmarks	# Statements			Memory Overhead			Performance Overhead				
	Tot	Procs	Avg	Avg	Max	Min	Analyzer	RA	RA%	MachSuif	Tot%
<i>164.gzip</i>	17,396	106	164	44,338	553,736	200	4.06	3.29	123%	53.30	8%
<i>175.vpr</i>	56,693	300	189	44,481	1,971,892	100	13.02	10.95	119%	169.55	8%
<i>181.mcf</i>	4,844	26	186	40,473	230,884	1,044	1.13	0.95	120%	28.14	4%
<i>197.parser</i>	40,677	324	126	43,675	2,147,404	100	11.64	7.15	163%	112.89	10%
<i>255.vortex</i>	203,810	923	221	80,572	10,027,076	100	53.29	41.78	128%	599.66	9%
<i>256.bzip2</i>	10,680	74	144	48,238	988,144	200	3.21	2.30	139%	32.09	10%
<i>300.twolf</i>	99,780	191	522	454,336	9,881,344	196	87.95	25.29	348%	307.81	29%
<i>FFT</i>	953	7	136	22,057	77,244	1,932	0.23	0.19	122%	6.65	3%
<i>bitcount</i>	816	15	54	7,177	21,000	1,328	0.10	0.13	81%	12.19	1%
<i>dijkstra</i>	434	6	72	10,934	32,792	200	0.07	0.06	122%	1.95	3%
<i>sha</i>	824	8	103	14,381	56,184	5,044	0.15	0.21	71%	4.19	4%
<i>stringsearch</i>	974	10	97	17,967	31,176	552	0.17	0.17	99%	10.25	2%
<i>jpeg</i>	82,923	506	164	38,805	925,564	100	20.90	17.12	122%	279.54	7%
<i>adpcm</i>	710	5	142	27,743	57,408	9,900	0.12	0.13	92%	5.70	2%
<i>epic</i>	11,452	49	234	88,801	1,935,300	956	6.22	4.46	139%	41.49	15%
<i>g721</i>	3,942	28	141	32,769	425,360	3,552	0.79	0.80	98%	13.48	6%
<i>mpeg2</i>	45,995	206	223	67,238	1,919,996	200	13.76	10.26	134%	131.44	10%

Table 1: Memory and performance overhead

Table 1 shows the performance and memory overhead of *ra-analyzer* for the benchmarks. The major column “# Statements” describes benchmark size. The secondary column “Tot” is the total number of intermediate code statements in a benchmark, “Procs” is the number of procedures, and “Avg” is the average number of statements.

In Table 1, the major column “Memory Overhead” gives statistics about the memory overhead. The average (Avg), maximum (Max) and minimum (Min) data in bytes are presented for procedures in each benchmark. As expected, MiBench has the lowest memory requirements. These programs have small procedures (e.g., *bitcount* has an average of 54 statements in a procedure), and as a result, the size of the data flow sets tends to be small. Other programs, namely *255.vortex* and *300.twolf*, have larger memory requirements. In *255.vortex*, `Draw701()` needs 10 MB because of its large number of intermediate code statements (5,228). However, *255.vortex*’s average memory requirement is consistent with the other benchmarks because it has only a few large procedures and many smaller ones. On the other hand, *300.twolf* has a relatively small number of procedures that are quite large and complex (varying from 3 to 4,462 intermediate statements). As a result, its average memory consumption is the largest among all programs. In this benchmark, `uclosepns()` has the maximum memory overhead (9.8 MB) because it has a large number of statements (4,001) and basic blocks (417). Although it doesn’t have the most statements in *300.twolf*, `uclosepns()` has the

most basic blocks and as a result, it incurs the most memory overhead. The average memory overhead is 85 KB for all benchmarks. This overhead is minimal.

We also investigated how the data flow sets (L, ST, and E) and the mappings contribute to total memory overhead. Because ST is a subset of L (see the data flow equations in Section 3.2), *ra-analyzer* records stale values only in ST for efficiency (i.e., L does not record stale values, which are already in ST). Across all benchmarks, L has the least memory consumption and ST has the most. L tends to be small (e.g., for `uclosepns()`, it is 375KB) because of the relatively small number of locations (operands) that it records. ST, on the other hand, tracks stale values. Thus, it is generally quite large (e.g., in `uclosepns()`, it is 6.26 MB). E is typically moderate in size; in `uclosepns()`, it is 3.2 MB. The mappings also consume memory, which is proportional to the number of intermediate statements and the number of operands. For the benchmarks, the mappings take 88 bytes to 450 KB (average 19 KB).

In Table 1, the major column “Performance Overhead” gives *ra-analyzer*’s run-time performance. The column “Analyzer” is the total run-time in seconds for *ra-analyzer* and the column “RA” is for MachSUIF’s allocator. The run-times are totals and account for compilation of all procedures in a benchmark. The column “RA%” is the percentage overhead of *ra-analyzer* over the allocator, which varies from 71% to 348% (average 96%). We expect that the run-time of *ra-analyzer* should be about the same as the run-time for the register allocator since both do somewhat similar analysis steps. In all benchmarks, except *300.twolf* and *197.parser*, the overhead follows this expectation, ranging from 71% to 139%. In *300.twolf* the overhead is 348% and in *197.parser* the overhead is 163%. This higher overhead is due to the use of iterative data flow analysis in *ra-analyzer*. In these two benchmarks, there is at least one complicated procedure where the data flow sets take a while to converge because of multiple, deep loop nests. For example, in *300.twolf*, the procedure `uclosepns()` takes the most time (10.96 Sec). It has 15 loop nests (with a maximum nest depth of 3), and takes up to 5 iterations for the data flow sets to converge.

The last two columns compare *ra-analyzer*’s performance to overall compile-time. The column labeled “MachSUIF” is the run-time of the MachSUIF compiler without *ra-analyzer*. The column “Tot%” is the total percentage increase in compile-time when *ra-analyzer* is run. On most benchmarks, *ra-analyzer*’s overhead is less than 10%. In *300.twolf*, the overhead is 29%. Despite this one benchmark, the tool works well: The average overhead relative to total compile-time is 8%. This small cost is worth the benefit of ensuring that the register allocation is correct.

5 Related Work

Several researchers have focused on proving the correctness of compiler optimization algorithms. Lacey et al. [14] used temporal logic to express data flow analysis and prove optimization correctness via reasoning. They did not consider register allocation. Naik and Palsberg [20] presented a proof for the correctness of an ILP register allocation algorithm. Ohori et al. [23] proposed a framework to construct and prove register allocation algorithms. Our work differs in that it addresses the implementation difficulties of register allocation, rather than algorithm correctness. Indeed, our work is complementary to the correctness proof of allocation algorithms.

Lerner et al. [16, 17] proved the soundness of several optimization implementations. Their approach requires the compiler engineer to use a domain-specific language to implement optimizations to automate reasoning about correctness. The verification of the register allocator's implementation is not presented.

Similar to our work, some research efforts suggest automatically checking semantic equivalence between the input and output code [18, 21, 22, 25, 27]. However, the range of optimizations that can be handled in these approaches is typically limited. Among these efforts, only McNerney et al. [18] and Nacula et al. [21] have examined how to check the output of the register allocator. The abstract interpretation approach in [18] applies only to a restricted domain of programs and did not present evaluation data. Nacula et al. [21] utilize symbolic evaluation in their translation validation infrastructure. However, this approach reports false alarms and has significant compile-time overhead. By focusing on allocation, SARAC can exploit properties of the allocation process (e.g., the property that def-use pairs are preserved in the output). As a result, our technique is accurate and fast. It also reports error casual information.

6 Conclusion and Future Work

This paper describes SARAC, a new approach to catch and identify bugs in register allocation. The approach statically checks that the input def-use pairs are maintained in the output code, given that the register allocator conducts limited edits. It is accurate and fast. The approach can be extended to handle register coalescing, rematerialization and sub-register class allocation. A prototype tool (*ra-analyzer*) shows that our approach has minimal compile-time and memory overhead.

A goal for our future work is to make *ra-analyzer* standalone so that it can be used with other compilers and machine architectures. To achieve this goal, SARAC will need to support more register allocators and register file structures, particularly ones that allow predication or have irregular register types. We also plan to more fully support type and architectural constraint checking. This support is important because the types and architectural constraints can be a common error source in a register allocator. Another issue is how to interface the tool to different compilers and intermediate representations. A final issue in making SARAC standalone is to develop a way to describe machine dependent information about registers to the tool.

7 References

- [1] M. E. Benitez and J. W. Davidson. A portable global optimizer and linker. *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, June 1988.
- [2] D. Bernstein, D. Q. Goldin et al. Spill code minimization techniques for optimizing compilers. *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, June 1989.
- [3] D. G. Bradlee, S. J. Eggers, and R. R. Henry. Integrating register allocation and instruction scheduling for RISCs. *4th Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems*, April 1991.
- [4] P. Briggs, K. D. Cooper and L. Torczon. Improvements to graph coloring register allocation. *ACM Trans. on Programming Languages and Systems*, 3(16): 428-455, May 1994.

- [5] P. Briggs, K. D. Cooper and L. Torczon. Rematerialization. *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, June 1992.
- [6] G. J. Chaitin. Register allocation & spilling via graph coloring. *Symp. on Compiler Construction*, June 1982.
- [7] F. C. Chow and J. L. Hennessy. The priority-based register allocation coloring approach. *ACM Trans. on Programming Languages and Systems*, 4(12):501-536, October 1990.
- [8] CPU2000 benchmark. Standard Performance Evaluation Corporation (SPEC), URL: <http://www.spec.org>.
- [9] J. W. Davidson and C. W. Fraser. Register allocation and exhaustive peephole optimization. *Software --- Practice and Experience*, 14 (9): 857-865, September 1984.
- [10] GCC. URL: <http://gcc.gnu.org/>.
- [11] L. George and A. W. Appel. Iterated register coalescing. *ACM Trans. on Programming Languages and Systems*, 3(18): 300-324, May 1996.
- [12] R. Gupta, M. L. Soffa and T. Steele. Register allocation via clique separators. *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, July 1989.
- [13] C. S. Jaramillo, R. Gupta and M. L. Soffa. Verifying optimizers through comparison checking. *Int'l. Workshop on Compiler Optimization Meets Compiler Verification*, April 2002.
- [14] D. Lacey, N. D. Jones, E. V. Wyk and C. C. Frederiksen. Proving correctness of compiler optimizations by temporal logic. *Symp. on Principles of Programming Languages*, January 2002.
- [15] C. Lee, M. Potkonjak and W. H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. *ACM/IEEE Int'l. Symp. on Microarchitecture*, 1997.
- [16] S. Lerner, T. Millstein and C. Chambers. Automatically proving the correctness of compiler optimizations. *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, June 2003.
- [17] S. Lerner, T. Millstein, E. Rice and C. Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. *Symp. on Principles of Programming Languages*, 2005.
- [18] T. M. McNeerney. Verifying the correctness of compiler transformations on basic blocks using abstract interpretation. *ACM/SIGPLAN Workshop Partial Evaluation and Semantics-Based Program Manipulation*, 1991.
- [19] MiBench. University of Michigan, URL: <http://www.eecs.umich.edu/mibench/>.
- [20] M. Naik and J. Palsberg. Correctness of ILP-based register allocation. Unpublished manuscript. URL: <http://theory.stanford.edu/~mhn/pubs/regalloc.pdf>.
- [21] G. C. Necula. Translation validation for an optimizing compiler. *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, June 2000.
- [22] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, June 1998.
- [23] A. Ohori. Register allocation by proof transformation. *12th European Symp. on Programming*, April 2003.
- [24] S. S. Pinter. Register allocation with instruction scheduling: a new approach. *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, June 1993.
- [25] A. Pnueli, M. Siegel and F. Singerman. Translation validation. *4th Tools and Algorithms for Construction and Analysis of Systems*, April 1998.
- [26] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Trans. on Programming Languages and Systems*, 5(21): 895-913, September 1999.

- [27] M. C. Rinard. Credible compilation. Technical Report MIT-LCS-TR-776, MIT, March 1999.
- [28] V. Santhanam and D. Odnert. Register allocation across procedure and module boundaries. *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, June 1990.
- [29] M. D. Smith and G. Holloway. Machine SUIF. URL: <http://www.eecs.harvard.edu/hube/research/machsuiif.html>.
- [30] M. D. Smith, N. Ramsey and G. Holloway. A generalized algorithm for graph-coloring register allocation. *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, June 2004.

8 Appendices

Mapping Grammar. To define the notation for a mapping, we give a short grammar:

```

<mapping> := operandposn: <out> → <in>
<out> := codept . location | codept . #constant
<in> := codept . value | codept . #constant
where,
operandposn – operand number in a statement
codept – a statement number in the input or output code
location – a register or memory location
value – a temporary or variable

```

For example, consider the code from Figure 1. The statement $r[2]=0$ at output code point 2 corresponds to $c=0$ at input code point 1; therefore, the mapping for the first operand $r[2]$ at code point 2 is: $1:2.r[2] \rightarrow 1.c$, where $r[2]$ is a *location* (memory or register) and c is a *value* (temporary or variable). Similarly, there is a mapping $2:6.\#0 \rightarrow 3.\#0$ to give the correspondence between the constants at output code point 6 and input code point 3. The mappings generated for the incorrect output code by mapGen in Figure 3 are:

```

1:2.r[2]→1.c    2:2.#0→1.#0
1:4.r[2]→2.d    2:4.#1→2.#1
1:6.r[1]→3.n    2:6.#0→3.#0    3:6.L3→3.L3
1:8.r[2]→4.t    2:8.r[1]→4.n    3:8.r[1]→4.d
1:9.r[2]→5.t    2:9.#0→5.#0    3:9.L2→5.L2
1:11.r[2]→6.c   2:11.r[2]→6.c   3:11.#1→6.#1
1:14.r[2]→7.d   2:14.r[2]→7.d   3:14.#1→7.#1
1:16.r[2]→8.t   2:16.r[2]→8.d   3:16.r[1]→8.n
1:17.r[2]→9.t   2:17.#1→9.#1   3:17.L1→9.L1

```

The mapping in bold is for error 1 in the incorrect output (code point 8).