Detecting Bugs in Register Allocation

YUQIANG HUANG and BRUCE R. CHILDERS
University of Pittsburgh
and
MARY LOU SOFFA
University of Virginia

Although register allocation is critical for performance, the implementation of register allocation algorithms is difficult, due to the complexity of the algorithms and target machine architectures. It is particularly difficult to detect register allocation errors if the output code runs to completion, as bugs in the register allocator can cause the compiler to produce incorrect output code. The output code may even execute properly on some test data, but errors can remain. In this article, we propose novel data flow analyses to statically check that the value flow of the output code from the register allocator is the same as the value flow of its input code. The approach is accurate, fast, and can identify and report error locations and types. It is independent of the register allocator and uses only the input and output code of the register allocator. It can be used with different register allocators, including those that perform coalescing and rematerialization. The article describes our approach, called SARAC, and a tool that statically checks a register allocation and reports the errors and their types that it finds. The tool has an average compile-time overhead of only 8% and a modest average memory overhead of 85KB. Our techniques can be used by compiler developers during regression testing and as a command-line-enabled debugging pass for mysterious compiler behavior.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Code generation, compilers, optimization*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Validation*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids, diagnostics*

General Terms: Languages, Verification

Additional Key Words and Phrases: Register allocation

ACM Reference Format:
Huang, Y., Childers, B. R., and Soffa, M. L. 2010. Detecting bugs in register allocation. ACM Trans. Program. Lang. Syst. 32, 4, Article 15 (April 2010), 36 pages.
DOI = 10.1145/1734206.1734212 http://doi.acm.org/10.1145/1734206.1734212

15

Authors' addresses: Y. Huang (corresponding author), B. R. Childers, Department of Computer Science, University of Pittsburgh, 210 S. Bouquet St. Rm. 6404, Pittsburgh, PA 15260; email: yuqiangh@cs.pitt.edu; M. L. Soffa, Department of Computer Science, University of Virginia, PO Box 400740, 151 Engineer's Way, Olsson Hall, Rm. 206, Charlottesville, VA 22904.

DOI 10.1145/1734206.1734212 http://doi.acm.org/10.1145/1734206.1734212

1. INTRODUCTION

One of the most critical compiler transformations is register allocation, as a good register allocator can make a dramatic improvement in program performance [Briggs et al. 1994; George and Appel 1996]. One study reported that careful register allocation can improve performance by one order of magnitude [Poletto and Sarkar 1999]. Thus, considerable effort has been given to developing new allocation algorithms or variants of existing ones [Bernstein et al. 1989; Bradlee et al. 1991; Briggs et al. 1992, 1994; Chaitin 1982; Chow and Hennessy 1990; George and Appel 1996; Gupta et al. 1989; Pinter 1993; Poletto and Sarkar 1999; Santhanam and Odnert 1990; Smith et al. 2004]. Given the many algorithms and the complexity of modern processor architectures, implementing register allocation is often a complex and error-prone task. It is particularly difficult to detect and locate errors in an erroneous output program of the register allocator if the program runs to completion. Various research efforts have proposed techniques to ensure the register allocation for a given program is correct [Jaramillo et al. 2002; McNerney 1991; Nandivada et al. 2007; Necula 2000; Pereira 2006]. In this article, we describe a novel technique to check the correctness of register allocation using only the input and output to the register allocator and to report the location and types of errors found. This technique is useful throughout the lifetime of a compiler; however, it is especially helpful during the development period.

Bugs can cause the compiler to fail on some input programs, but not on others. The generated code may have errors, although the compiler does not crash. Such latent bugs will not be discovered until a particular test input causes the program to fail, which could happen after the compiler is released. Assuming that a test input catches a bug, the developer may 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 [2000] proposed a symbolic evaluation approach to check the allocator's output code against its input code. However, this approach reports false alarms and increases the compile-time by up to four times. Jaramillo et al. [2002] proposed a dynamic checking approach that runs the allocator's input and output code. It compares values in the input and output 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. Nandivada et al. [2007] proposed a framework for designing and verifying register allocation algorithms; the major aim of this framework is to prove the soundness of the type system.

In this article, we propose a new approach, called SARAC, which uses *static analyses* to check the value flow (values of variables) of the allocator's output [Dor et al. 2004; Steffen et al. 1990] with the value flow of its input. SARAC

reports the location and type of errors in the output code due to an incorrect allocation. The analyses check that the value flow of the output code matches the value flow of the input code. SARAC traverses all program paths, using data flow analysis to gather information about the output code. It then checks the value flow of variables using the gathered information. A checking step verifies that the value flow of the input code is preserved in the output code, according to data dependencies, once the allocator has assigned registers and possibly spilled registers. The information collected during the analyses is used to determine error types and locations. Other errors involving register types and instruction set architecture constraints are still possible. Identifying errors in the value flow is an important step toward a tool that fully checks and reports any bugs in the output of the register allocator.

SARAC has several desirable characteristics. It gives hints to the compiler engineer to help her diagnose and fix bugs in the allocator. It is accurate and 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. 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). It uses data flow analyses and can be easily implemented. 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 85KB. Both the performance and memory usage of *ra-analyzer* scale well with function body size, the number of operands, and the complexity of the Control Flow Graph (CFG). Our techniques can be used by compiler developers during regression testing and as a command-line-enabled debugging pass for mysterious compiler behavior.

This article makes several contributions, including:

—an efficient way (SARAC) to statically check that the value flow after register allocation is the same as before register allocation and to identify and report the location and type of bugs, independently of the register allocator;

—data flow equations that analyze value locations, stale values, and evicted values using only the input and output code of a register allocator to provide information about the type of register allocation errors;

—techniques to support register allocators that perform coalescing, rematerialization, and register aliasing;

—a tool (*ra-analyzer)* that implements SARAC in SUIF's back-end optimizer (MachSUIF [Smith and Holloway]) for the Intel IA-32 instruction set architecture;

—a thorough evaluation of our techniques in *ra-analyzer* and MachSUIF, including validation, performance, and memory overhead, and scalability with program complexity.

The article is organized in the following way. The next section describes how register allocation preserves the value flow of the input code. The third section presents algorithms for gathering and using data flow information to check for

correctness. The fourth section describes and evaluates *ra-analyzer*. The fifth section discusses related work, and the final section concludes the article.

2. REGISTER ALLOCATION

This section describes our register allocator model, including the input and output code to the allocator. It also gives the motivation and background for our static analyses that catch and identify bugs in the register allocation.

2.1 Register Allocation Model and Assumptions

We assume that a set of intermediate code instructions is input to the register allocator and the output is a set of intermediate code instructions with registers assigned to the operands. We assume the input code is semantically correct relative to the source-code. The register allocator is a global allocator that allocates registers at the function level. It has a limited number of registers and can inject spill code through copy instructions (e.g., loads, stores, and register copies). It can also perform register coalescing to remove copy instructions. Since we focus on value flow, we assume the register allocator uses correct types and obeys the constraints imposed by the machine instruction set. The calling convention that we use for the register allocator is caller-save (callee-save would work as well). We assume the register allocator does not incorporate other optimizations (e.g., instruction scheduling) [Bradlee et al. 1991; Pinter 1993], and it does not change the structure of the Control Flow Graph (CFG) or the order of instructions within basic blocks. The core function of the register allocator is to assign *locations* (registers or memory slots) to hold *values* of variables.

As typical of most back-end optimizers, we further assume that register promotion is done in a pass prior to register allocation [Cooper and Lu 1997]. Thus, aliasing, assignment through pointers, array accesses, and related phenomena have been handled and expanded into low-level code by the register promotion pass and are not the concern for our technique. For example, in the most conservative situation, a variable whose address is taken will be loaded before any use of the variable and stored after any definition of the variable. The input code to the register allocator has these load and store operations (to move values between memory and virtual registers) [Cooper and Lu 1997]. Global and array values will have similar explicit loads and stores. These loads and stores will not be removed by the register allocator. Our technique treats these load and store operations like other computational operations in the input intermediate code. We also assume that values are not mutated during store operations; however, memory addressing modes are allowed for destination operands in CISC-style instructions.

Our technique treats the register allocator as a black box; that is, we only consider the input code to the register allocator and the output. We assume that two variables cannot be assigned to the same register if one is live at a program point where the other is defined, where liveness is defined by a variable being used after the current point. It should be noted that if another definition of liveness is used, such as a variable is live if it is both defined before and used

after the current point, then our technique may report a register allocation error when in fact there is none. This situation would occur, for example, when one branch of a conditional defines a variable X and the other branch defines a variable Y. After the merge point, one path uses X and another path uses Y. Using our definition of liveness, X and Y would have to be in different registers while the alternative definition of liveness could place both X and Y in the same register.

To simplify the following discussion, we often use an instruction identification number (ID) to represent an instruction. We use RTL to give actual instructions in examples [Benitez and Davidson 1988; Davidson and Fraser 1984; GCC]. Although RTL is used for notational convenience, our techniques can work with other intermediate representations that have the characteristics described next.

The input and output to the register allocator is intermediate code (IR) instructions. The intermediate code is a set of instructions $\{i_0, i_1, \ldots\}$, where an instruction $i = (id, op, defs, uses)$ is a 4-tuple and id is the instruction identification number. In the definition of instruction i, *defs* is the ordered set of operands defined by i and *uses* is the ordered set of operands used by i. The order of operands in *defs* and *uses* is the order in which they appear in instruction i. The operands in the intermediate code are the set $Operands = Registers \cup Memory\_Slots \cup Variables \cup Immediates$. The *Registers* set contains all the usable hardware registers in a target machine. In RTL, `r[x]` refers to register `x`. The *Memory_Slots* set represents memory locations in the activation record, with constant stack pointer manipulations. In RTL, `M[loc]` refers to memory slot `loc`. The *Variables* set represents source variables, compiler temporary variables, and virtual registers. The *Immediates* set represents constant values.

In the definition of instruction i, *op* is the opcode, which is defined as:

—*copy* for copies, including load, store, and copy between registers or variables. A *copy* instruction "propagates" a value from its use to its definition. The definition and use operand of a copy instruction belong to *Registers*, *Memory_Slots*, or *Variables*. In RTL, a load is shown as "`r[x]=M[loc]`", a `store` is "`M[loc]=r[x]`", and a register-to-register copy is "`r[x]=r[y]`".

—*call* for function calls; if a call has return values, they are in the *defs* set of the call instruction.

—*comp* for all other types of instructions, including arithmetic/logic, jump, branch, return, and parameter passing instructions. Load and store operations in the input intermediate code are also handled as comp instructions. A parameter passing instruction initializes a parameter to the value defined in the caller. Using RTL notation, a parameter passing instruction is defined as "`x=parameter`", indicating that x is a parameter and defined by the caller.

An instruction of *call* or *comp* type is considered by the allocator as defining a "new" unique value for each of its definition operands. The "new" value can be represented by a unique symbolic name, say v. Collectively, we call these two

type instructions as *noncopy* type instructions. A *copy* type instruction, on the other hand, is considered as only propagating a value without defining a "new" value.

A set of functions is also defined to manipulate the IR code. These are:

—isCopy(i) → *Boolean*: If $i = (id, copy, defs, uses)$, return *true*; otherwise, return *false*.

—isCall(i) → *Boolean*: If $i = (id, call, defs, uses)$, return *true*; otherwise, return *false*.

—id(i) → *Integer*: Given $i = (id, op, defs, uses)$, return id.

—opcode(i) → {*copy*, *call*, *comp*}: Given $i = (id, op, defs, uses)$, return op.

—defs(i) → ordered set of *Operands*: Given $i = (id, op, defs, uses)$, return *defs*.

—uses(i) → ordered set of *Operands*: Given $i = (id, op, defs, uses)$, return *uses*.

—isAlloc(*Operand*) → *Boolean*: If $Operand \in Registers \cup Memory\_Slots \cup Variables$, return *true*; otherwise, return *false*.

—isCallerSave(*Operand*) → *Boolean*: This function returns *true* if the operand given as a parameter belongs to *Registers* and is also caller-save; otherwise, it returns *false*. A caller-save register may be overwritten during the execution of a function call and, thus, must be saved by the caller.

The functions defs() and uses() return the ordered definition/use set of an instruction. The other functions are self-explanatory.

Since the purpose of a register allocator is to allocate registers to an instruction's operands, a *one-to-one* associating relation (i.e., a *mapping*) exists for the *noncopy* type instructions between the input and output of the register allocator. Given an input instruction i of *noncopy* type and its associated output instruction i', a register allocator maintains the following property.

$$\text{id}(i) = \text{id}(i'), \text{opcode}(i) = \text{opcode}(i'), |\text{defs}(i)| = |\text{defs}(i')|, and\ |\text{uses}(i)| = |\text{uses}(i')|.$$

This property states that, for each *noncopy* instruction, its associated output instruction has the same identification number, the same opcode, and the same defs/uses set size. Therefore, the operands of an input *noncopy* instruction i and its transformed output i' also have a *one-to-one* operand associating relation. The details of how SARAC generates these mappings are presented in Section 3.1.

2.2 Example

To discuss register allocation, we give an example in Figure 1, which counts the number of integer divisors for some number, `m`. We use this example throughout the article. In Figure 1, the number after each instruction is the instruction identification number (ID). The *copy* instructions are indicated, and all the instructions not indicated are of *noncopy* type.

In the example, we assume that `r[1]` is assigned by the allocator to hold variable `m` and `r[2]` is used to hold other variables as necessary. Figure 1(a) shows the source program, while Figure 1(b) shows the input intermediate code to the register allocator. Figure 1(c) gives the output code that should be generated by

(a) Source Code

```
/*Program function – count number
of divisors to variable m that is
passed as an argument*/

c=0;
for (d=1; d<=m; d++) {
  if (m%d == 0)
    c++;
}
```

(b) Input to Allocator

```
   m=parameter;                    0
   c=0;                            1
   d=1;                            2
   PC=((m<=0)?L3:PC+4);            3
L1:t=m%d;                          4
   PC=((t!=0)?L2:PC+4);            5
   c=c+1;                          6
L2:p=d; /*coalesced in output*/    7 cp
   d=p+1;                          8
   t=d<=m;                         9
   PC=((t==1)?L1:PC+4);           10
L3:
```

(c) Correct Output from Allocator

```
   M[m]=parameter;                 0
   r[1]=M[m];                     20 cp
   r[2]=0;                         1
   M[c]=r[2];                     21 cp
   r[2]=1;                         2
   M[d]=r[2];                     22 cp
   PC=((r[1]<=0) ? L3:PC+4);       3
L1:r[2]=M[d];                     23 cp
   r[2]=r[1]%r[2];                 4
   PC=((r[2]!=0)?L2:PC+4);         5
   r[2]=M[c];                     24 cp
   r[2]=r[2]+1;                    6
   M[c]=r[2];                     25 cp
L2:r[2]=M[d];                     26 cp
   r[2]=r[2]+1;                    8
   M[d]=r[2];                     27 cp
   r[2]=r[2]<=r[1];                9
   PC=((r[2]==1)?L1:PC+4);        10
L3:
```

(d) Incorrect Output from Allocator

```
   M[m]=parameter;                 0
   r[1]=M[m];                     20 cp
   r[2]=0;                         1
   M[c]=r[2];                     21 cp
   r[2]=1;                         2
   M[d]=r[2];                     22 cp
   PC=((r[1]<=0) ? L3:PC+4);       3
L1:r[2]=M[d];                     23 cp
   r[2]=r[1]%r[1]; /*err1: wrg reg (d)*/  4
   PC=((r[2]!=0)?L2:PC+4);         5
   r[2]=M[c];                     24 cp
   r[2]=r[2]+1; /*err2: stale (c)*/  6
   M[d]=r[2]; /*wrg spill(cause err2,3)*/  25 cp
L2:r[2]=M[d];                     26 cp
   r[2]=r[2]+1; /*err3: eviction (d)*/  8
   M[d]=r[2];                     27 cp
   r[2]=r[2]<=r[1];                9
   PC=((r[2]==1)?L1:PC+4);        10
L3:
```

Fig. 1. Example: source, input to register allocator, correct and incorrect output where the number after each instruction is the instruction ID, and *cp* indicates that an instruction is a *copy*.

the register allocator, assuming it is correctly implemented. Finally, Figure 1(d) gives output code that may be generated by a register allocator with a bug(s).

In this code, there are a few important items to notice. In Figure 1(b), instruction "p=d" is removed from the correct/incorrect output code due to coalescing. A number of *copy* instructions are also injected into the correct/incorrect output. Each *noncopy* instruction in the input stays in the correct/incorrect output code. Figure 1(d) shows the effect of a bug in the register allocator. In this code, two incorrect code edits were made by the register allocator. The first incorrect edit occurs to instruction 4, where the wrong register has been assigned to the second source operand. The other incorrect edit happens at instruction 25, where the wrong memory location is used for the spill. The example also shows the instructions where these errors are manifested. The instruction where an error is manifested is not necessarily the instruction where the incorrect edit is made. For example, the incorrect edit at instruction 25 is manifested as errors 2 and 3 at instructions 6 and 8.

2.3 Value Flow Preservation by Register Allocation

A correct allocation of registers must preserve the input code's semantics, particularly the data dependencies and the values of these dependencies, or the *value flow* [Dor et al. 2004; Steffen et al. 1990]. The *copy* type instructions only "propagate" a value, and the *noncopy* type instructions define a "new" unique value for each definition. Every input *noncopy* instruction will be mapped to an instruction in the output code. Instructions of *copy* type can be injected or removed by the register allocator. Intuitively, an allocation has correct value flow if the "values" defined/used by every *noncopy* instruction in the output are the same as the "values" defined/used by the mapped input instruction.

To discuss how the allocator may maintain or violate value flow, we use two data-dependency-related definitions. In the definitions, we use the notation "$i.x=$" to indicate the definition of operand x in instruction i and "$i. = x$" to indicate the use of operand x in i. We also use the notation "$i.x_n = x_m$" to indicate a *copy* type instruction i that uses x_m and defines x_n.

Definition 1. A *du-pair* $(i_0.x =, i_1. = x)$ in a program function is a 2-tuple, such that:

(i) isAlloc(x) is true,
(ii) there is a definition of x at instruction i_0,
(iii) there is a use of x at instruction i_1, and
(iv) x defined at i_0 reaches i_1; that is, there is a path from i_0 to i_1 such that there is no other definition of x along the path.

The preceding definition describes a data dependency (e.g., reaching definitions) with our definition-use terms. Figure 2 displays the complete list of du-pairs for the example function in Figure 1. For instance, in Figure 2(a), the du-pair `(2.d=,4.=d)` indicates that d is defined at instruction 2 and then used at instruction 4.

Definition 2. Given the du-pairs (Du-pairs) for a function, a *du-sequence* $(i_0.x_0 =, i_1.x_1 = x_0, \ldots, i_n.x_n = x_{n-1}, i_{n+1}. = x_n)$ is a (n + 2)-tuple, where n $\geq$ 0, such that:

(i) ¬isCopy(i_0) is true,
(ii) ¬isCopy(i_{n+1}) is true,
(iii) isCopy(i_k) is true, where n $\geq$ k $\geq$ 1, and
(iv) $(i_k.x_k =, i_{k+1}. = x_k) \in$ Du-pairs, where n $\geq$ k $\geq$ 0.

Definition 2 specifies a du-sequence as a sequence of instructions where the start and end instructions are of *noncopy* type and the instructions in between are of *copy* type. A unique symbolic value, say v, is defined by an instruction of *noncopy* type. It is then propagated by a series of *copy* type instructions and finally used by a *noncopy* instruction. A "new" unique symbolic value may also be defined and used without being propagated by copy instructions. We use **start**(du-sequence) to denote the beginning of a du-sequence ("$i_0.x_0=$ ") and **end**(du-sequence) to denote the end ("$i_{n+1}. = x_n$"). Figure 2 displays the

(a) Du-pairs of Input

```
{(0.m=,3.=m),
 (0.m=,4.=m),
 (2.d=,4.=d),
 (8.d=,4.=d),
 (4.t=,5.=t),
 (1.c=,6.=c),
 (6.c=,6.=c),
 (2.d=,7.=d),
 (8.d=,7.=d),
 (7.p=,8.=p),
 (8.d=,9.=d),
 (0.m=,9.=m),
 (9.t=,10.=t)}
```

(b) Du-sequences of Input

```
{(0.m=,3.=m),
 (0.m=,4.=m),
 (2.d=,4.=d),
 (8.d=,4.=d),
 (4.t=,5.=t),
 (1.c=,6.=c),
 (6.c=,6.=c),
 (2.d=,7.p=d,8.=p),
 (8.d=,7.p=d,8.=p),
 (8.d=,9.=d),
 (0.m=,9.=m),
 (9.t=,10.=t)}
```

(c) Du-pairs of Correct Ouput

```
{(0.M[m]=,20.=M[m]),
 (1.r[2]=,21.=r[2]),
 (2.r[2]=,22.=r[2]),
 (20.r[1]=,3.=r[1]),
 (22.M[d]=,23.=M[d]),
 (27.M[d]=,23.=M[d]),
 (20.r[1]=,4.=r[1]),
 (23.r[2]=,4.=r[2]),
 (4.r[2]=,5.=r[2]),
 (21.M[c]=,24.=M[c]),
 (25.M[c]=,24.=M[c]),
 (24.r[2]=,6.=r[2]),
 (6.r[2]=,25.=r[2]),
 (22.M[d]=,26.=M[d]),
 (27.M[d]=,26.=M[d]),
 (26.r[2]=,8.=r[2]),
 (8.r[2]=,27.=r[2]),
 (8.r[2]=,9.=r[2]),
 (20.r[1]=,9.=r[1]),
 (9.r[2]=,10.=r[2])}
```

(d) Du-sequences of Correct Output

```
{(0.M[m]=,20.r[1]=M[m],3.=r[1]),
 (0.M[m]=,20.r[1]=M[m],4.=r[1]),
 (2.r[2]=,22.M[d]=r[2],23.r[2]=M[d],4.=r[2]),
 (8.r[2]=,27.M[d]=r[2],23.r[2]=M[d],4.=r[2]),
 (4.r[2]=,5.=r[2]),
 (1.r[2]=,21.M[c]=r[2],24.r[2]=M[c],6.=r[2]),
 (6.r[2]=,25.M[c]=r[2],24.r[2]=M[c],6.=r[2]),
 (2.r[2]=,22.M[d]=r[2],26.r[2]=M[d],8.=r[2]),
 (8.r[2]=,27.M[d]=r[2],26.r[2]=M[d],8.=r[2]),
 (8.r[2]=,9.=r[2]),
 (0.M[m]=,20.r[1]=M[m],9.=r[1]),
 (9.r[2]=,10.=r[2])}
```

(e) Du-pairs of Incorrect Ouput

```
{(0.M[m]=,20.=M[m]),
 (1.r[2]=,21.=r[2]),
 (2.r[2]=,22.=r[2]),
 (20.r[1]=,3.=r[1]),
 (22.M[d]=,23.=M[d]),
 (27.M[d]=,23.=M[d]),
 (20.r[1]=,4.=r[1]),
 (4.r[2]=,5.=r[2]),
 (21.M[c]=,24.=M[c]),
 (24.r[2]=,6.=r[2]),
 (6.r[2]=,25.=r[2]),
 (22.M[d]=,26.=M[d]),
 (25.M[d]=,26.=M[d]),
 (27.M[d]=,26.=M[d]),
 (26.r[2]=,8.=r[2]),
 (8.r[2]=,27.=r[2]),
 (8.r[2]=,9.=r[2]),
 (20.r[1]=,9.=r[1]),
 (9.r[2]=,10.=r[2])}
```

(f) Du-sequences of Incorrect Output

```
{(0.M[m]=,20.r[1]=M[m],3.=r[1]),
 (0.M[m]=,20.r[1]=M[m],4.=r[1]),
 (4.r[2]=,5.=r[2]),
 (1.r[2]=,21.M[c]=r[2],24.r[2]=M[c],6.=r[2]),
 (2.r[2]=,22.M[d]=r[2],26.r[2]=M[d],8.=r[2]),
 (6.r[2]=,25.M[d]=r[2],26.r[2]=M[d],8.=r[2]),
 (8.r[2]=,27.M[d]=r[2],26.r[2]=M[d],8.=r[2]),
 (8.r[2]=,9.=r[2]),
 (0.M[m]=,20.r[1]=M[m],9.=r[1]),
 (9.r[2]=,10.=r[2])}
```

Fig. 2. Du-pairs and Du-sequences for input to register allocator, correct and incorrect output of the example.

complete list of du-sequences for the example function in Figure 1. For example, in Figure 2(b), the du-sequence `(2.d=,7.p=d,8.=p)` states that a "new" unique value is defined by *noncopy* instruction 2 (i.e., "`d=1`"), propagated by *copy* instruction 7 (i.e., "`p=d`"), and finally used by *noncopy* instruction 8 (i.e., "`d=p+1`").

When the allocator correctly maintains the data dependency of the input code, each input du-sequence has one or more associated output du-sequences. The start of the input du-sequence associates to the start of the output du-sequence; the end of the input du-sequence associates to the end of the output du-sequence. For instance, consider Figures 2(b) and 2(d). The input code to the register allocator has the du-sequence `(2.d=,7.p=d,8.=p)` and the correct output code has the du-sequence `(2.r[2]=, 22.M[d]=r[2],26.r[2]=M[d],8.=r[2])`. These du-sequences are associated in that "`2.d=`" associates to "`2.r[2]=`" and "`8.p=`" associates to "`8.=r[2]`". In other words, the input definition operand `d` (of instruction 2) is mapped to the output definition `r[2]` and the input use `p` (of instruction 8) is mapped to the output use `r[2]`.

Multiple du-sequences may connect to each other by sharing the same start or ending instruction. That is, a start definition may propagate along multiple du-sequences to reach different ending uses; likewise, an ending use may be reachable along multiple du-sequences from different start definitions. For instance, Figure 2(b) has two input du-sequences `(1.c=,6.=c)` and `(6.c=,6.=c)` that have the same ending instruction. During register allocation, the allocator must make the two defined values of c available in (at least) one common register (memory slot) before the merge point. This action ensures that both definitions reach the common use. Figure 2(d) shows the two correct output du-sequences `(1.r[2]=,21.M[c]=r[2],24.r[2]=M[c],6.=r[2])` and `(6.r[2]=, 25.M[c]=r[2],24.r[2]=M[c],6.=r[2])`, where both defined values for c are available in `M[c]` before the merge point. From the perspective of the register allocator, multiple definitions that reach the same use may be considered as different versions of a value. We next define a *du-sequence-web* to consist of connected sequences and a helper definition called *connected*.

Definition 3. Given two du-sequences s and s', **singly-connected**(s, s') is true if $s \neq s'$ and $\text{start}(s) = \text{start}(s') \vee \text{end}(s) = \text{end}(s')$. The relationship "*connected*" is the transitive closure of singly-connected sequences; that is, if **singly-connected**(s_0, s_1) is true and **singly-connected**(s_1, s_2) is true, then **connected**(s_0, s_2) is true.

Definition 4. A *du-sequence-web* is a set W of du-sequences, $\{s_1, \ldots, s_n\}$, such that:

(i) $\forall s_i \in W, \forall s_j \in W, s_i \neq s_j$, then connected$(s_i, s_j)$ is true, and
(ii) $\forall s_i \in W, \forall s \notin W$, then connected$(s_i, s)$ is false.

Figure 3 shows the various ways that du-sequences can be connected, using shared definitions or uses for the input function in Figure 1. The graphs imposed by the connections of du-sequences are also shown. For instance, in Figure 3(a), `{(1.c=,6.=c),(6.c=,6.=c)}` represents the du-sequence-web that involves

(a) Du-sequence-webs of Input

```
{{(0.m=,3.=m),(0.m=,4.=m),(0.m=,9.=m)},
 {(2.d=,4.=d),(8.d=,4.=d),(2.d=,7.p=d,8.=p),
  (8.d=,7.p=d,8.=p),(8.d=,9.=d)},
 {(4.t=,5.=t)}
 {(1.c=,6.=c),(6.c=,6.=c)},
 {(9.t=,10.=t)}}
```

(b) Visual Effect for Du-sequence-webs of Input

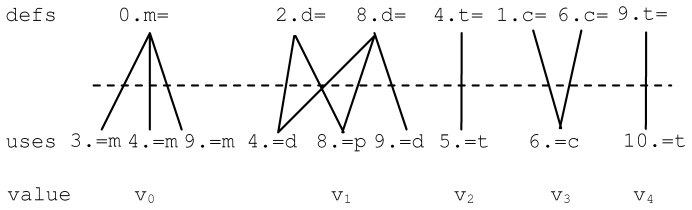

Fig. 3. Du-sequence-webs for input to register allocator of the example.

variable c (with a given value v_3). Also, {(4.t=,5.=t)} and {(9.t=,10.=t)} have only one du-sequence.

2.4 Sources of Errors

A bug in the register allocator that causes the output program (but not the compiler) to crash or produce a wrong result is manifested through incorrect code edits that can be made by the allocator. For the register allocator model defined in Section 2.1, the possible incorrect edits are:

(1) *incorrect register (or memory slot) assignment*: the wrong register (or memory slot) is used for an operand in a *noncopy* instruction;
(2) *incorrect instruction of copy type*: a value is spilled or copied incorrectly by using the wrong register or memory slot in a *copy* instruction;
(3) *missing instruction of copy type*: a value is not spilled or copied when needed.

These edits can violate the value flow of the input code by causing the du-sequences from the input function to not associate to the output du-sequences from the transformed function or vice versa. The incorrect edits can allow the program to run to completion but produce a wrong result. If there are no expected correct program results for comparison, the impact of the incorrect edits on the program's value flow will not be detected. Even if it is noticed that the program being compiled produces a wrong result, there is no obvious starting point to isolate where the incorrect edits occurred. Thus, these incorrect edits can pose a challenge to the compiler engineer.

Note the distinction between an "incorrect edit" and an "error": An incorrect edit is the cause of an error. The incorrect edit defines where the error was introduced, but it is not necessarily the place where the error is manifested. An incorrect edit may not manifest itself as an error until a value affected by the edit is used, which is the instruction at the end of a du-sequence. For

instance, in Figure 1(d), the incorrect edit at instruction 25 is not manifested until instructions 6 and 8.

The incorrect edits can lead to three error types: *wrong operand error*, *stale value error*, or *eviction error*. Although these errors all involve value flow violations, we distinguish between them to report causal information about an error. The three errors are defined as follows:

—A *wrong operand error* occurs when a register or memory slot is referenced that does not hold the needed value. The value is actually held in some other location(s). This error is usually caused by an incorrect register assignment.

—A *stale value error* happens when a register or memory location is referenced that holds an old version of the needed value. A wrong or missing spill is a common cause.

—An *eviction error* occurs when a value is referenced that is not held in any location. This error is usually caused by a wrong spill.

For each type of error, there is an instance in the incorrect output of Figure 1(d). First, there is a wrong operand error at instruction 4. This error is caused by an incorrect register assignment at instruction 4, where d is assigned to `r[1]` instead of `r[2]`. Thus, there is no definition of d that is used at instruction 4 and a wrong value (of `m`) is used instead. As shown in Figure 2, the input du-sequences `(2.d=,4.=d)` and `(8.d=,4.=d)` do not have the associated output du-sequences. The "new" defined value of d is available in `r[2]` at instruction 4, but it is not correctly used. Second, there is a stale value error at instruction 6. This error is introduced by the wrong spill at instruction 25, where `r[2]` is spilled to `M[d]`, rather than to `M[c]`. Thus, there is no du-sequence for c along the loop's back edge that reaches the use at instruction 6. A stale value for `c` is used instead. That is, the input du-sequence `(6.c=,6.=c)` does not have an associated output du-sequence. At instruction 6, only the stale value of `c` (previously defined at instruction 1) is available in `r[2]`. Finally, there is an eviction error at instruction 8. This error is also caused by the wrong spill at instruction 25. During the execution of the incorrect code, right before instruction 25, `M[d]` is the only location that holds the value of `d`. At instruction 25, the value of `d` is evicted from `M[d]` due to the wrong spill. Therefore, no location holds d after instruction 25 and the value of d cannot be loaded into `r[2]` at instruction 26. Finally, an eviction error is caught at instruction 8, where the value of d is referenced. As shown in Figure 2(f), the output du-sequence `(6.r[2]=,25.M[d]=r[2],26.r[2]=M[d],8.=r[2])` does not have an associated input du-sequence.

3. ERROR ANALYSIS FOR REGISTER ALLOCATION

To find register allocation errors, we developed a Static and Automatic Register Allocation Checking technique, called SARAC. Intuitively, the technique checks if the du-sequences in the register allocator's input function match the du-sequences in the allocator's output. However, to report error type rather than just detect errors, the approach first constructs du-sequences for the allocator's input function. A unique value, named v, is given to each input

```
1: SARAC(input,output) {
2:   //Step 1: mapping generation
3:   Map mappings = mapGen(input,output);
4:   //Step 2: data flow analysis
5:   Dataflow sets = defAnalysis(output,mappings);
6:   //Step 3: check the allocation
7:   errAnalysis(output,mappings,sets);
8: }
```

Fig. 4. Pseudocode for SARAC steps.

du-sequence-web. The unique value v is then mapped to the output definition and use. At the output definition point, v is considered as the new defined value; at the use point, v is the value that is expected to be used. Then, a data flow analysis on the output function propagates v from the definition point along *copy* instructions to the use of v by a *noncopy* instruction. Finally, a check is done to verify that the value flowing into the use is actually the expected one. The data flow analysis on the output function implicitly constructs the output du-sequences, and it also collects information about error types.

SARAC has three steps as shown in Figure 4. First, mapping information is generated by a step called **mapGen** using input and output functions to the allocator. Then, iterative forward data flow analysis, called **defAnalysis**, is performed on the output function using the mapping information. This analysis collects three types of data flow sets that are used to check the correctness of the output and to report error locations and types. Finally, a linear scan, called **errAnalysis**, is done to find and report value flow violations in the output function with the collected data flow sets.

3.1 Mapping Generation (mapGen)

SARAC needs to collect what values (represented by a name, v) are defined/used in the input code to the register allocator and what values are actually defined/used in the output code. First, the input du-sequences are constructed and each du-sequence-web is assigned a unique value, named v. Associations (i.e., mappings) between input and output *noncopy* instruction definitions/uses are generated. An associated definition in the output code for a *noncopy* instruction is considered as defining v. An associated use in the output code for a *noncopy* instruction is expected to use v.

To express the mapping information, we use the following grammar.

<mapping> : = id:posn:<dou>: <out> $\mapsto$ *<in>*
<dou> : = DEF|USE
<out> : = location | # constant
<in> : = value | # constant
where
id – the identification number for *noncopy* instructions
posn – operand number in an instruction's definition or use set
location – a register or memory slot
value – a given unique name representing value(s)

For example, in Figure 1, the instruction "`r[2]=0`" in the output code associates to "`c=0` in the input code. Therefore, the mapping for the definition operand

r[2] at instruction 1 is 1:1:DEF:r[2]→v_3, where r[2] is a *location* and v_3 is a *value*, which is the unique name given to the du-sequence-web involving "1.c=" (see Figure 3). A mapping can also associate immediate constants in the output and input. For instance, there is a mapping 1:1:USE:# 0→# 0 that gives the association between the constants at output instruction "r[2]=0" and input instruction "c=0". A mapping is generated for all *noncopy* instructions in the output. No mapping is generated for a *copy* instruction.

As shown in Figure 5, mapGen() generates mappings between the allocator's input and output code, where the allocator is viewed as a black box. First, the input function is processed to traverse each du-sequence-web. This traversal replaces each definition/use in the start/end instructions with a given unique name, v. Then, for each output *noncopy* instruction, its mapped input instruction is retrieved. The retrieval is based on the input instruction's identification number, which is the same as the mapped output instruction. We note that the input and output code is not necessarily in a specific order to retrieve the mapped instructions. Finally, each definition/use in every output *noncopy* instruction is mapped to its associated definition/use in the input, which has been replaced with a unique name v.

The mappings are used by both defAnalysis and errAnalysis. defAnalysis uses the mappings to track values and their name associations. errAnalysis uses the mappings to detect errors at uses and to report error causes. Although a complete mapping has an instruction ID, operand number, and an identifier to distinguish between definitions and uses, we use an abbreviation (i.e., *location*↦*value*) to simplify the discussion of the data flow equations for SARAC. For example, the output code in Figure 1 has an instruction "r[2]=0" that is associated to the input instruction "c=0". We abbreviate the complete mappings as simply r[2] ↦ c and 0 ↦ 0 in this case.

3.2 Data Flow Analysis (defAnalysis)

To check if the register allocation is correct and to determine error locations and types, defAnalysis gathers 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 all locations (registers and memory), (2) the stale values, and (3) the evicted values. Note if we only wanted to know whether a register allocation is correct, we would not need the eviction information. Because we also want to report the error types to help isolate a bug, all three types of information are gathered. 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 output instruction 2, "r[2]=1", in Figure 1 is processed, the mapped destination operand v_1 (that replaces d) is retrieved from the mappings. This mapping is used to derive three pieces of information. First, the current value of v_1 is defined in r[2]. Second, the value v_3 in r[2] is evicted. Finally, any previous value of v_1 in other locations is stale.

This information is collected in three data flow sets: the Location set (L), the Stale set (ST) and the Eviction set (E). Each set consists of triples (l, v, c), where

```
1:  mapGen(input,output) {
2:    replaceValue(input);
3:    Map mappings:=∅;
4:    foreach (Instruction i'∈output) {
5:      if (¬isCopy(i')=true) {
6:        Integer id := id(i');
7:        Instruction i := getInstr(input,id);
8:        for (k=1; k≤|defs(i)|; k++) {
9:           Opnd l := getSetElemt(defs(i'),k);
10:          Opnd v := getSetElemt(defs(i),k);
11:          mappings:=mappings∪"id:k:DEF: l→v";
12:       }
13:       for (k=1; k≤|uses(i)|; k++) {
14:          Opnd l := getSetElemt(uses(i'),k);
15:          Opnd v := getSetElemt(uses(i),k);
16:          mappings:=mappings∪"id:k:USE: l→v";
17:       }
18:     }
19:   }
20:   return mappings;
21: }

22: replaceValue(input) {
23:   DuSeqS duseqs := Du-Sequences(input);
24:   foreach (DuSeq duseq∈duseqs) {
25:     Def def := start(duseq);
26:     if (def is visted)
27:       continue;
28:     //provide a unique name for a web
29:     Variable v := genName();
30:     traverseWeb(input,def,v,duseqs);
31:   }
32: }

    //replace the start/end opnds in a du-sequence web by v
33: traverseWeb(input,def,v,duseqs) {
34:   if (def is visited) then return;
35:   Mark def as visited;
36:   Integer id := getId(def);  //def="id.x="
37:   Opnd x := getOpnd(def);
38:   Instruction i := getInstr(input,id);
39:   replaceDefValue(i,x,v);  //replace x with v at i
40:   //follow ending uses for du-sequences starting with def
41:   foreach (DuSeq duseq∈duseqs and start(duseq)=def) {
42:     Use use := end(duseq);
43:     if (use is visited) then continue;
44:     Mark use as visited;
45:     Integer id := getId(use);  //use="id.=x"
46:     Opnd x := getOpnd(use);
47:     Instruction i := getInstr(input,id);
48:     replaceUseValue(i,x,v);  //replace x with v at i
49:     //follow starting defs for du-sequences ending with use
50:     foreach (DuSeq duseq∈duseqs and end(duseq)=use) {
51:       Def def := start(duseq);
52:       traverseWeb(input,def,v,duseqs);
53: }}}
```

Fig. 5. Pseudocode for mapping generation.

l is a location (register or memory) from the output code, v is a value (name) from the input code, and c is a vector consisting of instructions. The instructions in c indicate how the relationship between l and v is built and have different semantics in three different sets; this information will help debugging when an error is exposed. The semantics of (l, v, c) for L, ST and E are defined as follows.

—The Location set L records the fact that location l holds v. The vector c records a du-sequence for v; that is, it incrementally records the intervening copy operations in a du-sequence such that when the data flow equations reach a fixed point, there will be sequential descriptions for every du-sequence in the program.
—The Stale set ST records that location l holds a stale v. When the data flow equations reach a fixed point, the vector c incrementally records the non-copy instruction that makes v become stale and the copy instructions that propagate stale v to location l.
—The Eviction set E records that v has been evicted from location l. For E, c is always a single-instruction vector, and that instruction kills v from l.

In a triple (l, v, c), we use the notation * to indicate a *don't care* value; that is, any value in a field in the triple matches with a *.

3.2.1 *Data Flow Equations.* A forward data flow analysis is used to compute L, ST, and E. We assume a control flow graph representation for the output code of the register allocator.

Our data flow equations extend the traditional dataset operations due to the third field of the triple, which is a vector. We redefine "$\cap$" and "$-$" to handle the vector field c.

$$P \cap Q = \{(l,v,c) \mid ((l,v,c) \in P \wedge (l,v,*) \in Q) \vee ((l,v,*) \in P \wedge (l,v,c) \in Q)\} \quad (1)$$

$$P - Q = \{(l,v,c) \mid (l,v,c) \in P \wedge (l,v,*) \notin Q\} \quad (2)$$

These two operators are similar to the normal set operators, but the operations are based only on the first two fields in the triple. The third field c is handled in a special way. We also use the traditional "$\cup$" definition.

$$P \cup Q = \{(l,v,c) \mid (l,v,c) \in P \vee (l,v,c) \in Q\} \quad (3)$$

Computing the Location Set (L).

$$L\_gen[i] = \{(l, v, \langle i \rangle) \mid \neg \text{isCopy}(i) \wedge l \in defs(i) \wedge l \mapsto v\} \quad (4)$$

L_gen[i] is computed when the processed instruction i is of *noncopy* type. For each such instruction, each of its definitions is considered as defining a new value (i.e., the mapped v). Therefore, a triple $(l, v, \langle i \rangle)$ is generated for each definition operand l. For instance, when instruction 6, "`r[2]=r[2]+1`, in Figure 1(d) is processed, a triple (`r[2]`, v_3, <6>) is generated in *L_gen[i]* due to `r[2]`$\mapsto v_3$.

L_kill[i] considers that the execution of i destroys the value in each of its definition operands and that crossing a function might destroy the value in

each caller-save register.

$$L\_kill[i] = \{(l, {}^*, {}^*) \mid l \in defs(i) \vee (isCall(i) \wedge isCallerSave(l))\} \quad (5)$$

Without knowledge about the usage of caller-save registers from the callee function, the global register allocator assumes that every caller-save register would be overwritten within the callee function. Therefore, the same assumption applies when checking the register allocation.

For value propagation of *copy* instructions, an operator $\oplus$ is defined as follows.

$$P \oplus i = \left\{ (l_d, v, \langle i_1, \ldots, i_k, i\rangle) \;\middle|\; \begin{array}{c} \exists (l_s, v, \langle i_1, \ldots, i_k\rangle) \in P \wedge \\ isCopy(i) \wedge \{l_d\} = defs(i) \wedge \{l_s\} = uses(i) \end{array} \right\} \quad (6)$$

The operator states that, if the source of a *copy* instruction holds a value that is indicated by $(l_s, v, \langle i_1, \ldots, i_k\rangle) \in P$, the same value will be propagated to the definition operand of that *copy* instruction. The vector $\langle i_1, \ldots, i_k, i\rangle$ is also modified to record the chain of value definition and propagation, which will result in a du-sequence.

Given the Gen, Kill and IN sets, *L_out[i]* is computed as follows.

$$L\_out[i] = L\_gen[i] \cup \big(L\_in[i] \oplus i\big) \cup \big(L\_in[i] - L\_kill[i]\big) \quad (7)$$

The value propagation of *copy* type instructions is conducted when computing *L_out[i]*. For example, when instruction 21, "`M[c]=r[2]`", in Figure 1(d) is processed, it is known that `(r[2],`v_3`,<1>)` is in *L_in[21]*. Therefore, a triple `(M[c],`v_3`,<1,21>)` is generated to indicate the fact that v_3 is defined in `r[2]` at instruction 1 and is copied to `M[c]` at instruction 21. Finally, *L_out[i]* records all the locations (registers and memory) that hold a value up to instruction i, regardless of whether the value is current or stale.

Computing the Stale Set (*ST*).

$$ST\_gen[i] = L\_gen[i] \quad (8)$$

ST_gen[i] is the same as *L_gen[i]*. For instructions of *noncopy* type, every definition operand l is considered as having a new value (i.e., v) defined. Therefore, the new definition of v makes the previous v held in any other location stale. All the locations (other than l) that hold a previous v will be discovered from *L_in[i]*.

ST_kill[i] is computed identically as *L_kill[i]*.

$$ST\_kill[i] = L\_kill[i] \quad (9)$$

The operator $\bullet$ for finding stale values is defined as follows.

$$P \bullet Q = \{\!\{(l', v, \langle i\rangle) \mid (l, v, \langle i\rangle) \in P \wedge (l', v, {}^*) \in Q \wedge l \neq l'.\}\!\} \quad (10)$$

For instance, when instruction 6, "`r[2]=r[2]+1`", in Figure 1(d) is processed, the definition operand `r[2]` is mapped to v_3 and (`M[c]`, v_3, $< 1, 21 >$) is retrieved from *L_in[6]*. Due to the new definition of v_3 into `r[2]`, the previous v_3 in `M[c]` is now stale. Therefore, a triple(`M[c]`, v_3, $< 6 >$) is generated.

ST_out[i] is computed as follows.

$$ST\_out[i] = (ST\_gen[i] \bullet L\_in[i]) \cup (ST\_in[i] \oplus i) \cup (ST\_in[i] - ST\_kill[i]) \quad (11)$$

In the equation, the operator $\oplus$ is applied to propagate the stale value. For a *copy* type instruction, if its source operand holds a stale value, then the value propagated into its definition is also stale. Whether the value in the source of a *copy* is stale or not is discovered from *ST_in[i]*. For example, when instruction 24, "r[2] = M[c]", in Figure 1(d) is processed, it is known that (M[c], v_3, $<6>$) is in *ST_in[24]* along the loop's back edge. Therefore, a triple (r[2], v_3, $<6, 24>$) is generated to record the fact that v_3 in M[c] is stale due to the new definition of v_3 at instruction 6 and is copied to r[2] at instruction 24 along the loop's back edge.

Computing the Eviction Set (*E*).

The equations for computing E are closely related to those for L.

$$E\_gen[i] = \{\!\{(l, ^*, \langle i \rangle) | (l, ^*, ^*) \in L\_kill[i]\}\!\} \tag{12}$$

$$E\_kill[i] = L\_gen[i] \tag{13}$$

E_gen[i] records that any value saved in l will be evicted because of i. What value actually evicted will be discovered from *L_in[i]*. *E_kill[i]* indicates that a new value is defined into the definition(s) of a *noncopy* instruction i.

To discover the actually evicted value, the operator $\diamondsuit$ is defined as follows.

$$P \diamondsuit Q = \{(l, v, \langle i \rangle) | (l, ^*, \langle i \rangle) \in P \wedge (l, v, ^*) \in Q\} \tag{14}$$

E_out[i] is computed as follows.

$$E\_out[i] = ((E\_gen[i] \diamondsuit L\_in[i]) \cup E\_in[i]) - (E\_kill[i] \cup (L\_in[i] \oplus i)) \tag{15}$$

The operator $\oplus$ is used here to discover the value propagated into the destination of a *copy* instruction i, which essentially kills a triple in E. Please also note that a triple in *E_gen[i]* can also be in *E_kill[i]*. For instance, *E_gen[i]* and *E_kill[i]* may both have $(l, v, \langle i \rangle)$. This situation happens when a previous instruction defines v in l and the following instruction i redefines v in l. In this case, $(l, v, \langle i \rangle)$ should not be in *E_out[i]* because v is still in l after executing instruction i. The association order in the equation enforces this semantics.

3.2.2 *The defAnalysis Algorithm.* Given the mapping information, defAnalysis analyzes the output code of the register allocator and returns the data flow sets L, ST, and E. The algorithm is shown in Figure 6. It implements a forward iterative data flow analysis.

In the algorithm, the function **setInitialization()** is called to initialize the IN sets of L, ST, and E for each basic block. As in standard iterative data flow analysis, the initialization value for IN (i.e., empty, ø , or universal, U) depends on which set operation (e.g., intersection or union) is applied at merge points in the control flow graph and whether a basic block is a CFG entry node. Given the initial IN sets, setInitialization(), calls computeLocalFlow() to compute the OUT sets for each basic block. The function **computeLocalFlow()** implements the data flow equations from Section 3.2.1. The data flow equations are executed in the order that they were discussed in Section 3.2.1.

In the iterative steps, **mergeFlow()** is called to execute set operations on L, ST, and E at a merge point. The code on lines 29−31 performs the set operations

```
 1: defAnalysis(output,mappings) {
 2:   Dataflow sets;
 3:   foreach (Block B∈output)
 4:     setInitialization(B,mappings,sets.L,sets.ST,sets.E);
 5:   change := true;
 6:   while (change) {
 7:     change := false;
 8:     foreach (Block B∈output and B≠Binitial) {
 9:       oldsets := sets_in[B];
10:       mergeFlow(output,B,sets.L,sets.ST,sets.E);
11:       if (oldsets ≠ sets_in[B]) {
12:         change := true;
13:         computeLocalFlow(B,mappings,sets.L,sets.ST,sets.E);
14:        }
15:     }
16:   }
17:   return sets;
18: }

19: setInitialization(B,mappings,L,ST,E) {
20:   if (B = B_initial) {
21:     L_in[B]:= ∅; ST_in[B]:= ∅; E_in[B]:= ∅;
22:   } else {
23:     L_in[B]:= U; ST_in[B]:= ∅; E_in[B]:= ∅;
24:   }
25:   computeLocalFlow(B,mappings,L,ST,E);
26: }

27: mergeFlow(output,B,L,ST,E) {
28:   //P ∈ Predecessors(B)
29:   L_in[B] := ∩L_out[P];
30:   ST_in[B] := ∪ST_out[P];
31:   E_in[B] := ∪E_out[P];
32:   //the following computes and uses L_inconsistent
33:   L_union := ∪ L_out[P];
34:   L_inconsistent := {(l,v,<B>)|∀(l,v,*)∈(L_union-L_in[B])};
35:   E_in[B] := E_in[B}∪ L_inconsistent;
36:   ST_in[B] := ST_in[B]-L_inconsistent;
37: }
```

Fig. 6. Pseudocode for data flow analysis algorithm.

on *L*, *ST*, and E. First, at the merge point to block *B*, *L_in* is computed by ∩ on *L_outs* of all predecessors to B. A correct register allocation has the same value in a common location along any preceding path for a later common use. Therefore, ∩ removes any "inconsistent triples" where the same location holds different values in different paths. For example, a triple from one path might show that `r[1]` holds `x`, but the other path indicates that `r[1]` does not hold `x`. Second, *ST_in* is computed as the union on *ST_outs* of all predecessors to *B*. The union is performed because if the value is stale along any path to the merge point, it is possible that the stale value might be used later. Hence, the union

operation preserves the fact that the value is stale along some path. Finally, *E_in* is computed as the union of *E_outs* of all predecessors to *B*. *E_in[B]* holds a value's history of most recent evictions from any location along all preceding paths.

As shown on lines 33–35 of Figure 6, the function mergeFlow() also collects the inconsistent L datasets, where a common location at a merge point holds the different values from the preceding paths. With a correct register allocation, different values cannot flow past merge points in a common location. Thus, it is safe to assume that these different values are evicted from the common location at the merge. The merge point (the label of a block) is also recorded to indicate where the inconsistency is exposed, and the resulting datasets are collected into the E set to help error reporting.

The data flow equations in Section 3.2.1 indicate that *ST* records a subset of *L*. In other words, *ST* records the locations that hold a stale value and *L* records all the locations that hold any value. As *L_inconsistent* is not recorded in *L_in*, there is no reason to continuously record it in *ST_in*. For optimization, *L_inconsistent* is removed from *ST_in* at line 36 in the pseudocode.

3.3 Checking and Reporting (errAnalysis)

Once *L*, *ST*, and *E* are collected, they are used to check the output code. The error analysis step ensures that the value flow from the input is preserved in the output. The algorithm for identifying and reporting errors is shown in Figure 7.

The errAnalysis algorithm iterates over all instructions in the output code. It calls **useCheck()** when processing every *noncopy* instruction. A *copy* type instruction is implicitly checked due to the value propagation performed in defAnalysis. useCheck verifies whether all uses in each *noncopy* instruction are correct in terms of the input value flow. It reports the error location and type for any value flow violations. For each use *l*, it consults the mappings to determine which value *l* should use. When *l* actually holds *v*, which is recorded as a triple (*l*, *v*, *) in *L_in*, useCheck further verifies if *v* in *l* is stale. Next, useCheck checks if *v* is in other locations. If so, then a wrong operand is used. Otherwise, an eviction error must have occurred. The history of *v*'s most recent eviction from any location is reported.

3.4 SARAC Applied to the Example

Based on the example in Figure 1, this section illustrates how SARAC works. We describe how SARAC can detect the errors in the incorrect output code (Figure 1(d)) from the example.

3.4.1 *Mapping Information.* When mapGen() is applied to the input and the incorrect output code in Figure 1, the mappings generated are as in Figure 8. The mappings in bold are the ones that relate to the instructions where there are errors. For example, at output instruction 4, the mapping indicates that the second register source operand, `r[1]`, expectedly holds v_0.

```
 1: errAnalysis(output,mappings,sets) {
 2:   L:=sets.L; ST:=sets.ST; E:=sets.E;
 3:   foreach (Block B∈output) {
 4:     foreach (Instruction i∈B) {
 5:       if (¬isCopy(i)=true)
 6:         useCheck(i,mappings,L,ST,E);
 7:     }
 8:   }
 9: }

10: useCheck(i,mappings,L,ST,E) {
11:   foreach (Opnd l∈uses(i)) {
12:     Opnd v := getMapping(i,l,mappings);
13:     if ((l,v,*)∈L_in[i]) {
14:       if ((l,v,c)∈ST_in[i])
15:         ε := "i uses stale value, c made v in l become stale";
16:       else
17:         ε := null;
18:     } elsif ((l',v,c)∈L_in[i] and (l',v,*)∉ST_in[i])
19:       ε := "i uses wrong operand, but c defined v in l'";
20:     else {
21:       ∀(l',v,c)∈E_in[i];
22:       ε := "i uses evicted value, c evicted v from l'";
23:     }
24:   }
25: }
```

Fig. 7. Pseudocode for checking algorithm.

```
0:1:DEF:M[m] ↦ v0
1:1:DEF:r[2] ↦ v3     1:1:USE:#0 ↦ #0
2:1:DEF:r[2] ↦ v1     2:1:USE:#1 ↦ #1
3:1:USE:r[1] ↦ v0     3:2:USE:#0 ↦ #0
4:1:DEF:r[2] ↦ v2     4:1:USE:r[1] →v0   4:2:USE:r[1] ↦ v1 //err1
5:1:USE:r[2] ↦ v2     5:2:USE:#0 ↦ #0
6:1:DEF:r[2] ↦ v3     6:1:USE:r[2] ↦ v3  6:2:USE:#1 ↦ #1  //err2
8:1:DEF:r[2] ↦ v1     8:1:USE:r[2] ↦ v1  8:2:USE:#1 ↦ #1  //err3
9:1:DEF:r[2] ↦ v4     9:1:USE:r[2] ↦ v1  9:2:USE:r[1] ↦ v0
10:1:USE:r[2] ↦ v4   10:2:USE:#1 ↦ #1
```

Fig. 8. Mappings between the incorrect ouput and input code in Figure 1.

3.4.2 *Data Flow Sets.* When the defAnalysis algorithm converges, L, ST, and E have been computed at each instruction of the output code. Figure 9 shows the data flow sets for some selected points in the incorrect output. Figure 9 uses solid horizontal lines to mark different basic blocks of the output. Consider two merge points in the example. First, the merge point (L1) shows the joined data flow sets of two blocks, which end by instructions 3 and 10. Similarly, the merge (L2) shows the joined data flow sets from instructions 5 and 25. The data flow sets are not changed when there is no destination operand in an instruction. In the figure, the data flow sets after instruction 22 are the same as the ones

Places	Location Set (L)	Stale Set (ST)	Eviction Set (E)
After 22/3	(M[m],v_0,<0>),(r[1],v_0,<0,20>), (M[c],v_3,<1,21>),(r[2],v_1,<2>), (M[d],v_1,<2,22>)	---	(r[2],v_3,<2>)
L1	(M[m],v_0,<0>),(r[1],v_0,<0,20>), (M[c],v_3,<1,21>),(M[d],v_1,<2,22>), (M[d],v_1,<8,27>)	(M[c],v_3,<6>)	(r[2],v_3,<2>),(M[d],v_3,<L2>), (r[2],v_2,<24>),(r[2],v_2,<L2>), (r[2],v_3,<L2>),(r[2],v_1,<9>), (r[2],v_1,<L1>),(r[2],v_4,<L1>)
After 23	(M[m],v_0,<0>),(r[1],v_0,<0,20>), (M[c],v_3,<1,21>),(M[d],v_1,<2,22>), (M[d],v_1,<8,27>),(r[2],v_1,<2,22,23>), (r[2],v_1,<8,27,23>)	(M[c],v_3,<6>)	(r[2],v_3,<2>),(M[d],v_3,<L2>), (r[2],v_2,<24>),(r[2],v_2,<L2>), (r[2],v_3,<L2>),(r[2],v_4,<L1>)
After 4/5	(M[m],v_0,<0>),(r[1],v_0,<0,20>), (M[c],v_3,<1,21>),(M[d],v_1,<2,22>), (M[d],v_1,<8,27>),(r[2],v_2,<4>)	(M[c],v_3,<6>)	(r[2],v_3,<2>),(M[d],v_3,<L2>), (r[2],v_3,<L2>),(r[2],v_1,<4>), (r[2],v_4,<L1>)
After 24	(M[m],v_0,<0>),(r[1],v_0,<0,20>), (M[c],v_3,<1,21>),(M[d],v_1,<2,22>), (M[d],v_1,<8,27>),(r[2],v_3,<1,21,24>)	(M[c],v_3,<6>), (r[2],v_3,<6,24>)	(M[d],v_3,<L2>),(r[2],v_1,<4>), (r[2],v_2,<24>),(r[2],v_4,<L1>)
After 6	(M[m],v_0,<0>),(r[1],v_0,<0,20>), (M[c],v_3,<1,21>),(M[d],v_1,<2,22>), (M[d],v_1,<8,27>),(r[2],v_3,<6>)	(M[c],v_3,<6>)	(M[d],v_3,<L2>),(r[2],v_1,<4>), (r[2],v_2,<24>),(r[2],v_4,<L1>)
After 25	(M[m],v_0,<0>),(r[1],v_0,<0,20>), (M[c],v_3,<1,21>),(r[2],v_3,<6>), (M[d],v_3,<6,25>)	(M[c],v_3,<6>)	(r[2],v_1,<4>),(r[2],v_2,<24>), (r[2],v_4,<L1>),(M[d],v_1,<25>)
L2	(M[m],v_0,<0>),(r[1],v_0,<0,20>), (M[c],v_3,<1,21>)	(M[c],v_3,<6>)	(r[2],v_3,<2>),(r[2],v_1,<4>), (r[2],v_2,<24>),(r[2],v_4,<L1>), (M[d],v_1,<25>),(M[d],v_1,<L2>), (r[2],v_2,<L2>),(r[2],v_3,<L2>), (M[d],v_3,<L2>)
After 26	(M[m],v_0,<0>),(r[1],v_0,<0,20>), (M[c],v_3,<1,21>)	(M[c],v_3,<6>)	(r[2],v_3,<2>),(r[2],v_1,<4>), (r[2],v_2,<24>),(r[2],v_4,<L1>), (M[d],v_1,<25>),(M[d],v_1,<L2>), (r[2],v_2,<L2>),(r[2],v_3,<L2>), (M[d],v_3,<L2>)
After 9/10	(M[m],v_0,<0>),(r[1],v_0,<0,20>), (M[c],v_3,<1,21>),(M[d],v_1,<8,27>), (r[2],v_4,<9>)	(M[c],v_3,<6>)	(r[2],v_3,<2>),(r[2],v_2,<24>), (M[d],v_3,<L2>),(r[2],v_2,<L2>), (r[2],v_3,<L2>),(r[2],v_1,<9>)

Fig. 9. Data flow sets at selected instructions of the incorrect output in Figure 1.

after instruction 3. These data flow sets are also the same for instructions after 4 and 5 and for instructions after 9 and 10.

3.4.3 *Identify Errors.* Incorrect edits in the incorrect output of Figure 1 are identified by errAnalysis using the data flow sets (Figure 9) and the mappings (Figure 8). The incorrect edits are: an incorrect register assignment at instruction 4 and a wrong spill at 25.

At instruction 4, mappings indicate that the second source operand r[1] is expected to hold v_1. However, after instruction 23, L reports that v_1 is actually in r[2]. Therefore, a wrong operand error is reported. L also reports that v_1 flows into r[2] along two paths. First, v_1 is defined at instruction 2, spilled to M[d] at 22, and reloaded into r[2] at 23. Second, v_1 is defined at 8, spilled to M[d] at 27, and reloaded into r[2] at 23. This information will help a compiler engineer discover that v_1 is in r[2] at instruction 4 and r[2] should have been used as the second source operand. A register assignment error is identified.

At instruction 6, the mapping information states that source operand r[2] should hold v_3. After instruction 24, the L set indicates that r[2] does in fact hold v_3. However, ST reports that the value of v_3 in r[2] is stale. Triple

(r[2], v_3, <6, 24>) in *ST* indicates that the value of v_3 in memory is stale due to the new definition of v_3 (into r[2]) at instruction 6. The stale value of v_3 is loaded into r[2] at instruction 24. Another triple in *ST*, (M[c], v_3, <6>), indicates that the value of v_3 in M[c] remains stale because the newly defined v_3 is never spilled to M[c] along the loop's back edge. This error is attributed to the wrong spill code edit at 25, where the newly defined v_3 (in r[2]) is incorrectly spilled to M[d].

At instruction 8, the mappings show that the source operand r[2] should hold v_1. The *L* set, however, tells that v_1 is not in any location. An eviction error is reported. The *L* set after instruction 26 shows that no value is loaded into r[2] by the load instruction 26. There is no value in M[d] when the control flow merges at L2, which is indicated by triples (M[d], v_1, <L2>) and (M[d], v_3, <L2>) in *E*. That is, v_1 is evicted from M[d] due to inconsistencies among the *L* sets from the predecessors. The preceding path ending at instruction 5 shows that v_1 is in M[d], which is indicated by triples (M[d], v_1, <2, 22>) and (M[d], v_1, <8, 27>) in *L*. The other path ending at instruction 25 shows that v_3 is in M[d], which is indicated by triple (M[d], v_3, <6, 25>) in *L*. Therefore, the values in M[d] are inconsistent and there is no valid value in M[d] flowing past L2. Triple (M[d], v_3, <6, 25>) in *L* shows how v_3 flows into M[d]. v_3 is defined in r[2] at instruction 6 and spilled to the wrong memory location M[d] at 25. The eviction error is caused by the wrong spill edit at instruction 25.

3.5 Extensions

Two important extensions to a register allocator are rematerialization [Briggs et al. 1992, 1994; Chaitin 1982; George and Appel 1996] and register aliasing [Smith et al. 2004]. This section describes how SARAC supports these extensions.

3.5.1 *Rematerialization.* Rematerialization improves spill code by recomputing values rather than reloading them from memory. The recomputed values are not touched by the register allocator; only the rematerialized instructions are moved (usually the instruction ID is changed). Rematerialization is done for constant expressions in the code, such as integer constants in load-immediate instructions and address offsets.

To handle rematerialization, our approach is extended to minimally use information from the allocator. The assumption about strict independence from the register allocator is relaxed to let the allocator indicate to SARAC the mapping information used for rematerialized instructions. The mapping information given by the register allocator is not necessarily assumed to be correct. Any incorrect rematerialization will be exposed and caught by SARAC's data flow analyses.

3.5.2 *Register Aliasing.* Many processor architectures allow registers from different subregister classes to overlap. Such overlapping registers are defined as a "register alias set" [Smith et al. 2004]. For instance, the Intel IA-32 has registers AL and AH in the 8-bit register class. These two registers overlap with registers AX in the 16-bit register class and EAX in the 32-bit register class.

The "register alias set" of AL and AH is {AL, AX, EAX} and {AH, AX, EAX} respectively. The register allocator has to consider the overlap when assigning subregister classes because a write to a register will destroy the value in any register of its alias set.

To handle register aliasing, SARAC needs to consider the effects of the "register alias set" on the data flow equations. Only a modest modification is needed to the data flow equation for computing *L_kill[i]*. The computations of *ST_kill[i]* and *E_gen[i]* are based on the modified *L_kill[i].* The modification is as follows.

$$L\_kill[i] = \left\{ (l', *, *) \;\middle|\; \begin{array}{c} \forall l', l' \in reg\_alias(l) \wedge \\ (l \in def(i) \vee (isCall(i) \wedge isCallerSave(l))) \end{array} \right\} \tag{16}$$

The function **reg_alias**(l) returns the register alias set if l is a hardware register; otherwise, it returns a set with l itself as the single element.

4. EXPERIMENTS

SARAC was implemented as a tool called *ra-analyzer*. We used MachSUIF, version 2.02.07.15, for the Intel IA-32 [Smith and Holloway]. MachSUIF is SUIF's back-end optimizer. A global graph coloring register allocator [George and Appel 1996] is included as a pass in MachSUIF. *ra-analyzer* is run after register allocation. We conducted three experiments with *ra-analzyer*. First, faults were injected into the allocator's output to validate *ra-analyzer* and to explore how the tool might be used to find bugs. Second, the performance and memory overhead of the tool was measured. Finally, the scalability of our technique was investigated.

For the experiments, we used all the benchmarks from SPECint2K, MediaBench [Lee et al. 1997], and MiBench that can be compiled by base SUIF. The functions in the benchmarks span a range of code sizes and complexities. The experiments were run on a RedHat Linux computer with a 2.4 GHz Pentium 4 and 1GB RAM.

4.1 Fault Injection

We checked if MachSUIF's allocator causes errors in the benchmarks and found no errors. There are two possible reasons that there are no errors. First, MachSUIF's allocator is correct (the ideal situation). Second, it could be that not enough test programs were used to cover all possible situations for the allocator. In particular, because many benchmark programs cannot be compiled by SUIF, there may be latent bugs which are not exposed by the benchmarks that can be compiled. Indeed, this situation illustrates the dilemma faced by a compiler engineer and user. In particular, we believe that *ra-analyzer* can improve the confidence in the allocator. It is especially useful for regression tests during the development of the compiler.

To validate *ra-analyzer* based on many runs and to illustrate how *ra-analyzer* might be used by a compiler engineer, we injected bugs into the output of MachSUIF's allocator. We 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, and

Table I. Validation Experiment by Fault Injection

	Errors				Fault Edits						Errors
Benchmarks	wrg_opnd	stale	eviction	total	wrg_assn	wrg_ld	wrg_st	mis_ld	mis_st	total	Faults
164.gzip	974	482	708	2,164	436	331	355	163	102	1,387	1.56
175.vpr	2,650	1,406	2,644	6,700	1,198	864	980	567	185	3,794	1.77
181.mcf	212	90	318	620	108	83	93	21	4	309	2.01
197.parser	2,732	1,142	2,588	6,462	1,268	1,030	1,035	286	77	3,696	1.75
255.vortex	7,552	2,648	7,903	18,103	3,198	2,245	2,859	1,834	613	10,749	1.68
256.bzip2	693	280	450	1,423	321	210	234	113	48	926	1.54
300.twolf	1,883	1,037	2,145	5,065	813	642	680	382	137	2,654	1.91
FFT	49	19	64	132	24	14	15	9	3	65	2.03
bitcount	136	61	63	260	66	53	60	4	3	186	1.40
dijkstra	51	18	39	108	22	15	22	13	5	77	1.40
sha	79	55	56	190	36	28	32	11	5	112	1.70
stringsearch	84	35	76	195	36	32	33	13	3	117	1.67
jpeg	4,652	2,321	6,485	13,458	2,035	1,634	1,757	638	225	6,289	2.14
adpcm	49	38	35	122	22	21	18	10	5	76	1.61
epic	481	278	420	1,179	213	168	176	78	36	671	1.76
g721	268	115	199	582	115	103	88	18	8	332	1.75
mpeg2	1,836	803	1,652	4.291	856	592	703	343	144	2,638	1.63

missing store/load. For each edit type, the fault injector randomly selected a basic block to change. An appropriate instruction was selected for a modification, based on the edit type. If an appropriate instruction could not be located, the edit was abandoned and a new one was tried. The injector attempted to make five changes for each edit type, but it sometimes made fewer edits when it could not find a candidate. Each function in every benchmark had zero to twenty-five incorrect edits. For each incorrect edit, we know the location and type of errors.

Table I shows the fault injection experiment. Sixty-five 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 covered the possible changes to the code described in Section 2.4. 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 all of the errors without generating any false positives or negatives, and reported their locations and types. The ratio of errors to faulty edits was also investigated, with an average of 1.72 across all benchmarks. This supports the claim that an erroneous edit can be detected as multiple errors because it can break multiple data dependencies.

As an example, the fault injector changed one register operand to a different register in the FFT benchmark. In this case, the instruction `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, a compiler engineer can identify what went wrong. For example, consistently using the wrong register might suggest that liveness analysis or interference graph construction had a problem. Randomly using a

Table II. Memory Usage and Performance

	# Instructions			Memory Usage (KByte)			Performance (Sec.)		
Benchmarks	Total	# Funcs	Avg. Instrs	Avg	Max	Min	ra-analyzer	RA	Total
164.gzip	17,396	106	164	44.3	553.7	0.2	4.06	3.29	53.30
175.vpr	56,693	300	189	44.5	1,971.9	0.1	13.02	10.95	169.55
181.mcf	4,844	26	186	40.5	230.9	1.0	1.13	0.95	28.14
197.parser	40,677	324	126	43.7	2,147.4	0.1	11.64	7.15	112.89
255.vortex	203,810	923	221	80.6	10,027.1	0.1	53.29	41.78	599.66
256.bzip2	10,680	74	144	48.2	988.1	0.2	3.21	2.30	32.09
300.twolf	99,780	191	522	454.3	9,881.3	0.2	87.95	25.29	307.81
FFT	953	7	136	22.1	77.2	1.9	0.23	0.19	6.65
bitcount	816	15	54	7.2	21.0	1.3	0.10	0.13	12.19
dijkstra	434	6	72	10.9	32.8	0.2	0.07	0.06	1.95
sha	824	8	103	14.4	56.2	5.0	0.15	0.21	4.19
stringsearch	974	10	97	18.0	31.2	0.6	0.17	0.17	10.25
jpeg	82,923	506	164	38.8	925.6	0.1	20.90	17.12	279.54
adpcm	710	5	142	27.7	57.4	9.9	0.12	0.13	5.70
epic	11,452	49	234	88.8	1,935.3	1.0	6.22	4.46	41.49
g721	3,942	28	141	32.8	425.4	3.6	0.79	0.80	13.48
mpeg2	45,995	206	223	67.2	1,920.0	0.2	13.76	10.26	131.44

wrong register suggests that the coloring function might cause the error. With the information from *ra-analyzer*, a compiler engineer can use a debugger to step through the allocator and find bugs.

4.2 Performance and Memory Overhead

Table II shows the memory usage and performance of *ra-analyzer* for the benchmarks. The major column "# Instructions" gives the benchmark size. The secondary column "Total" is the total number of intermediate code instructions in a benchmark, "# Funcs" is the number of functions, and "Avg. Instrs" is the average number of instructions per function.

The memory overhead experiment was conducted to ensure that SARAC does not impose excessively high memory demands due to the collection of its data flow sets. In Table II, the major column "Memory Usage" gives statistics about the memory overhead (i.e., the sum of the sizes of L, ST, and E datasets and mappings for a function). The average (Avg), maximum (Max), and minimum (Min) data sizes in kilobytes are presented for the functions in each benchmark. As expected, MiBench has the lowest memory requirements. These programs have small functions (e.g., *bitcount* has an average of 54 instructions in a function), 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 10MB because of its large number of intermediate code instructions (5,228). However, *255.vortex*'s average memory requirement is consistent with the other benchmarks because it has only a few large functions and many smaller ones. On the other hand, *300.twolf* has a relatively small number of functions that are quite large and complex (varying from 3–4,462 intermediate instructions). As a result, its average memory consumption is the largest among all programs. In this benchmark, `uclosepns()` has the maximum memory overhead (9.8MB) because it has a large number of instructions

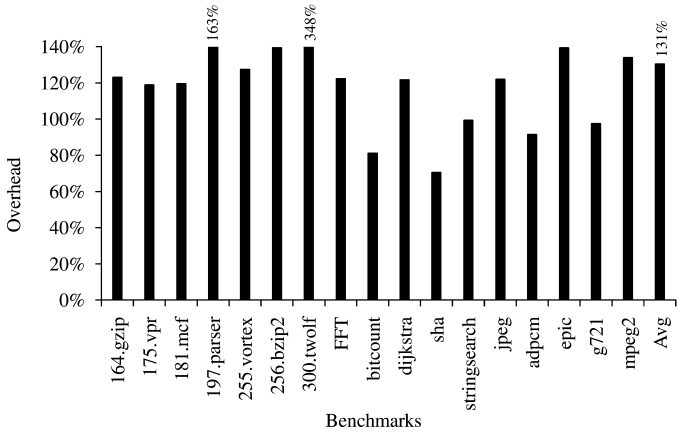

Fig. 10. Compile-time performance overhead versus register allocation.

(4,001) and basic blocks (417). Although it doesn't have the most instructions in *300.twolf*, `uclosepns()` has the most basic blocks and as a result, it incurs the most memory overhead. The average memory overhead is 85KB for all benchmarks. This overhead is minimal and has a small impact on the compiler, given the many gigabytes of memory available in today's machines.

Performance experiments were also conducted. These experiments measure the overhead imposed by checking the register allocation; we need to ensure that SARAC's overhead is reasonable enough that it will be useful in practice. In Table II, the major column "Performance" gives *ra-analyzer*'s runtime. The column "ra-analyzer" is the total runtime in seconds for *ra-analyzer*, the column "RA" is the runtime for MachSUIF's register allocator, and the column "Total" is the runtime for MachSUIF without *ra-analyzer*. The runtimes are totals and account for compilation of all functions in a benchmark. The system call "time()" was used to measure the runtimes. The I/O time for reading/writing intermediate code was included, which can be significant for MachSUIF.

We summarize the performance overhead from the table in Figure 10. This figure gives the percentage overhead of *ra-analyzer* versus MachSUIF's register allocator. The overhead varies from 71%–348%, with an average of 131%. We expect that the runtime of *ra-analyzer* should be about the same as the runtime 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%–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 function where the data flow sets take a while to converge due to multiple, deep loop nests.

Figure 11 summarizes the total percentage increase in compile-time when *ra-analyzer* is run. The bar labeled "ra-analyzer" summarizes the overhead relative to the whole compiler runtime from Table II. For most benchmarks, *ra-analyzer*'s overhead is less than 10%. In *300.twolf*, the total compile-time overhead is 29% for *ra-analyzer* due to this benchmark's complexity as noted

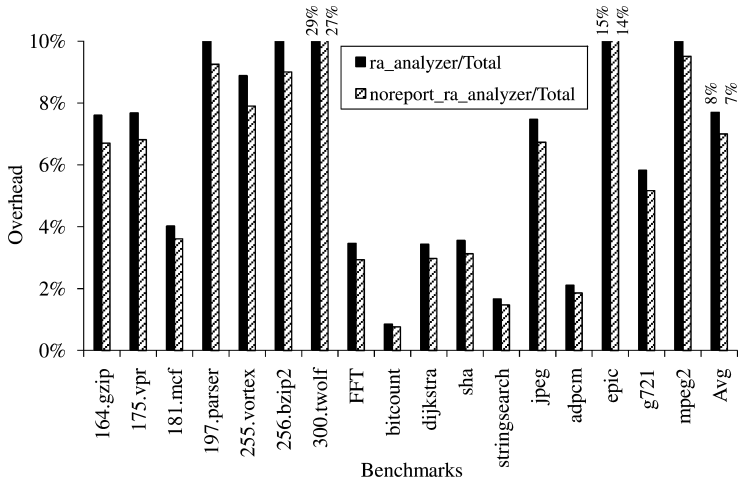

Fig. 11. Compile-time performance overhead versus full compilation.

before. Figure 11 also shows the total overhead when the eviction information is not collected. Recall that this information is needed to report errors, rather than to actually detect errors. As a result, it can be turned off. As the graph shows, disabling the collection of the eviction set does help ra-analyzer's performance somewhat. For example, *300.twolf*'s overhead is improved to 27%. However, the total overhead is not significantly impacted due to the iterative data flow analysis (it takes the same number of iterations to converge with and without collecting the E set). Finally, the average overhead relative to total compile-time is 8% with the E set and 7% without the eviction set. From this experiment, we conclude that the benefit of ensuring that the register allocation is correct is worth the small average overhead cost, especially when developing a compiler.

4.3 Scalability

To understand what aspects of SARAC have the largest impact on *ra-analyzer*'s performance and memory requirements, we studied its scalability as function size is increased. The scalability study investigates what are the important factors that most affect performance and memory usage.

Table III shows memory scalability. For the benchmarks, the functions are sorted into multiple ranges according to the number of instructions in a function. "Funcs" is the number of functions in a range. In each range, the average (Avg), minimum (Min), and maximum (Max) total memory usage (Total) data in bytes are presented. Correspondingly, the data flow sets (L, ST, and E) and the mappings contributing to the total memory usage are also presented.

The average values show that the memory usage (Total, Mapping, L, ST, and E) of *ra-analyzer* scales well with function body size (Instrs). That is, the larger function body size, the larger memory consumption, as expected. The total memory usage and the mappings demonstrate linear scalability. There are also deviations from this trend. In many ranges, some maximum values are larger than the corresponding minimum values in the following ranges. These

Table III. Memory Usage Scalibility of *ra-analyzer*

# Instructions	Funcs		Total (byte)	Mappings (byte)	L (byte)	ST (byte)	E (byte)	Instrs	Opnds	Preds	Blks
		Avg	*434*	*388*	*34*	*11*	*1*	*5*	*22*	*2*	*3*
1–9	121	Min	100	88	12	0	0	2	20	2	3
		Max	980	884	60	24	12	9	26	2	3
		Avg	*6,481*	*4,398*	*290*	*877*	*915*	*50*	*57*	*9*	*8*
10–99	1,293	Min	880	736	84	60	0	10	26	2	3
		Max	21,304	8,420	1,020	5,656	6,208	98	100	29	20
		Avg	*45,327*	*20,284*	*1,077*	*12,756*	*11,210*	*224*	*193*	*35*	*25*
100–499	1,109	Min	10,468	9,652	60	396	360	100	68	2	3
		Max	232,160	38,948	5,056	86,636	101,520	492	382	169	107
		Avg	*241,160*	*63,033*	*3,125*	*94,778*	*80,224*	*693*	*552*	*100*	*67*
500–999	186	Min	64,824	57,784	292	4,724	2,024	519	281	10	9
		Max	666,556	87,820	4,464	352,280	221,992	993	820	179	119
		Avg	*807,285*	*123,859*	*5,763*	*380,416*	*297,246*	*1,358*	*1,056*	*188*	*124*
1000–1999	50	Min	140,176	120,396	600	9,796	9,384	1,026	1,210	19	15
		Max	2,367,520	133,052	10,428	1,395,132	828,908	1,633	1,048	468	283
		Avg	*2,279,540*	*225,952*	*10,849*	*1,250,763*	*791,976*	*2,405*	*1,928*	*313*	*205*
2000–2999	18	Min	630,336	217,616	7,388	190,700	214,632	2,206	1,761	204	139
		Max	5,252,940	235,104	17,896	3,196,864	1,803,076	2,638	2,238	530	351
		Avg	*7,513,830*	*366,500*	*15,668*	*4,470,704*	*2,660,958*	*3,875*	*3,941*	*584*	*390*
3000–3999	2	Min	6,710,252	369,876	14,896	3,473,144	2,852,336	3,965	4,125	563	381
		Max	8,317,408	363,124	16,440	5,468,264	2,469,580	3,784	3,756	605	398
		Avg	*7,500,023*	*418,027*	*19,495*	*4,185,669*	*2,876,832*	*4,423*	*3,834*	*571*	*375*
4000–5999	5	Min	3,863,508	458,304	18,592	1,941,340	1,445,272	4,462	3,307	369	261
		Max	10,027,076	444,476	31,668	4,741,600	4,809,332	5,228	3,184	841	506

deviations are attributed to factors other than the number of instructions in the functions. These other factors include the number of operands (Opnds), the CFG complexity, and the characteristics of the live ranges. The CFG complexity is indicated by the number of basic blocks (Blks), the number of predecessors (Preds), and the loop nest level. The characteristics of live ranges can be represented by the interference graph of live ranges. For instance, the maximum values (i.e., for function `table_pointer()` in *197.parser)* of *L*, *ST*, *E*, and Total in the range 10–99 are larger than the corresponding minimum values (i.e., for function `usage()` in *jpeg*) in range 100–499. Although these two functions have similar body size (one has 98 instructions and the other has100), their CFG complexity are quite different. `table_pointer()` has 20 basic blocks and each basic block has an average of 1.45 (i.e., 29/20) predecessors, while `usage()` has only 3 basic blocks. This difference causes the CFG of table_pointer() to be more complex than usage(). Furthermore, `table_pointer()` has 100 different operands in the input intermediate code, but `usage()` has 68. The study also shows that the live range interference graph for `table_pointer()` is more complex than `usage()`.

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 the benchmarks, *L* has the least memory consumption and *ST* has the most. *L* tends to be small (e.g., for `uclosepns()`, it is 18KB) because of the relatively small number of locations (operands) that it records. *ST*, on the other

Table IV. Performance Scalibility of *ra-analyzer*

# Instructions	Funcs		Time (Sec)	Iters	Instrs	Opnds	Preds	Blks
		Avg	*0.0006*	*2*	*5*	*22*	*2*	*3*
1–9	121	Min	0	2	2	20	2	3
		Max	0.002	2	9	27	2	3
		Avg	*0.006*	*2.3*	*50*	*57*	*9*	*8*
10–99	1,293	Min	−0.048	2	32	41	7	6
		Max	0.022	4	92	102	18	15
		Avg	*0.042*	*2.7*	*224*	*193*	*35*	*25*
1,109	1,109	Min	0.001	3	102	91	19	15
		Max	0.21	4	449	616	10	9
		Avg	*0.21*	*3.1*	*693*	*552*	*100*	*67*
500–999	186	Min	0.08	2	514	265	34	27
		Max	0.67	5	993	820	179	119
		Avg	*0.64*	*3.4*	*1,358*	*1,056*	*188*	*124*
1000–1999	50	Min	0.24	2	1,026	1,210	19	15
		Max	1.94	5	1,739	1,595	271	175
		Avg	*2.08*	*3.8*	*2,405*	*1,928*	*313*	*205*
2000–2999	18	Min	0.72	2	2,206	1,761	204	139
		Max	4.47	5	2,826	2,921	164	116
		Avg	*7.85*	*5*	*3,875*	*3,941*	*584*	*390*
3000–3999	2	Min	7.10	5	3,965	4,125	563	381
		Max	8.61	5	3,784	3,756	605	398
		Avg	*7.57*	*4.4*	*4,423*	*3,834*	*571*	*375*
4000–5999	5	Min	4.06	3	4,462	3,307	369	261
		Max	10.96	5	4,001	3,909	629	417

hand, tracks stale values. Thus, it is generally quite large (e.g., in `uclosepns()`, it is 6.26MB). E is typically moderate in size; in `uclosepns()`, it is 3.2MB. The mappings also consume memory, which is proportional to the number of intermediate instructions and the number of operands. For the benchmarks, the mappings take 88 bytes to 450KB (average 19KB).

Table IV shows performance scalability. Across the benchmarks, the functions are sorted into multiple ranges according to the number of instructions. In each range, the average (Avg), minimum (Min), and maximum (Max) runtimes (Time) of *ra-analyzer* in seconds are reported based on the function level. The system call "gettimeofday()" was used to measure the time that *ra-analyzer* spent on checking each function. The I/O time for reading/writing intermediate code was excluded from the measurement. "Iters" is the number of iterations that the iterative data flow analysis step (`defAnalysis`) of *ra-analyzer* takes to converge. "Time" and "Iters" have a close (but not linear) relationship.

The average values show that the performance of *ra-analyzer* scales well with function body size (Instrs). It has a monotonically increasing trend. Similar to memory scalability, performance also has deviations. In many ranges, the maximum time is larger than the minimum time in the immediately following ranges. These deviations are attributed to other factors besides the function body size, including the number of operands (Opnds) and the CFG complexity. The linear relationship between "Time" and "Iters" indicates that the CFG complexity is perhaps the most critical factor that affects performance. The reason

is that the CFG complexity is important in determining how many iterations the iterative data flow analysis will take to converge. In Table IV, the Max and Min data in the bottom range happens to the function `uclosepns()` and `config1()` in *300.twolf*, respectively. The function `uclosepns()` takes the most time (10.96 sec), although it has only 4,001 instructions. 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. However, `config1()` takes much less time (4.056 sec) even though it has a larger body size (4,462 instructions). This is because `config1()` has a less complex CFG (i.e., with fewer deep loop nests).

5. RELATED WORK

The importance of compiler optimization and the error-prone nature of compiler implementation have led to a number of research efforts related to our work. These efforts can be categorized into verification of optimization implementations, static checking of optimization outputs, dynamic checking of optimization outputs, and data flow analysis in related areas.

5.1 Verification of Optimization Implementations

The research of Lerner et al. [2003] focused on optimization implementations. It automatically proved the soundness of optimization implementations, where optimizations are proved correct once and for all, for any input code. However, this approach requires compiler optimizations to be implemented in a specific way. A compiler engineer has to use a domain-specific language to implement optimizations to automate the reasoning about correctness. Optimizations must be expressed as first-order predicates. Furthermore, the input code to the optimizations needs to be written in a subset of the domain-specific language. For predicates to be proved automatically by a theorem prover, the domain-specific language is strict and limited compared with a general-purpose language such as C. With such a domain-specific language, it is difficult to express advanced data structures in complicated optimizations, including register allocation. In their later work [Lerner et al. 2005], the domain-specific language expresses optimizations using explicit data flow facts manipulated by local propagation and transformation rules. Lee [2003] presents a register allocation framework and formally verifies it with an inductive theorem prover. In this work, the verification of the register allocator's implementation is not introduced.

5.2 Static Checking of Optimization Outputs

Several research efforts suggested statically checking the output code of an optimization against the input code [Barthe et al. 2006; Blech and Gregoire 2008; Leroy 2006; Li et al. 2007; McNerney 1991; Nandivada et al. 2007; Necula 2000; Necula and Lee 1998; Pereira 2006; Pnueli et al. 1998; Rinard 1999; Rival 2004]. In these works, the semantic equivalence between the input and the output code is automatically checked. These techniques guarantee the correctness of the output code, and therefore, improve the reliability of the optimization implementation. However, the range of optimizations that can

be handled is typically limited. Pnueli et al. [1998] and Rinard [1999] do not consider register allocation in their work. Another similar project [Rival 2004] also does not consider aggressive optimizations, including register allocation. This work checks the semantic equivalence between the source and compiled programs in one pass while our work focuses on register allocation to identify and report detailed information about bugs. Leroy [2006], McNerney [1991], Necula [2000], Nandivada et al. [2007], and Pereira [2006] have examined how to check the register allocator's output code.

Leroy's work [2006] formally certifies a compiler backend including register allocation by using Coq proof assistant. When certifying register allocation correctness, it relies on the register allocator to give correct mapping information. This reliance may reduce certification confidence. Our approach does not rely on the register allocator to provide correct mapping information even when extended to support rematerialization.

The approach of McNerney [1991] checks several optimizations including register allocation. They implemented a specialized program equivalence prover for the domain of assembly language programs emitted by the Connection Machine Fortran compiler and targeted for the CM-2. Using abstract interpretation, the prover removes details such as register and stack usage, as well as evaluation order within functional blocks, thereby reducing the problem to a trivial tree comparison. The symbolic values of the register allocator's input and output code are compared at block boundaries to detect mismatches. However, it detects mismatching values without gathering information about the cause of mismatches. This approach applies only to a restricted domain of programs. Evaluation data is not presented.

The approach of Necula [2000] also can handle several optimizations including register allocation. However, this approach reports false alarms and can have high compile-time overhead (about four times). This approach is based on symbolic evaluation. Simulation relations are used to describe under what conditions two program fragments are equivalent. The mismatching symbolic values are reported without collecting the information for investigating the cause of mismatches. However, the error causes are valuable to isolating allocator bugs.

Nandivada et al. [2007] propose a framework for designing, verifying, and evaluating register allocation algorithms. A major component of their framework is a type checker that checks the output of a register allocator to help find bugs and to prove the soundness of the type system. Our work focuses on checking the value flow consistency of register allocation.

The work by Pereira [2006] presents a collection of data flow algorithms that statically validate the data flow semantic consistency of register allocation. This work formalizes the concept of semantic consistency of register allocation based on a simplified register allocation model. The soundness of the approach is proved. However, the simplified model does not include advanced register allocation functions, like register coalescing. It also does not consider the case that two live ranges of the same variable are allocated to different registers at a merge point. Although the author implements the technique, experimental data is not presented.

By focusing on register 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 approach is precise and fast (average compile-time overhead is 8%). Our approach reports causal information about errors in the register allocator's output code, including value flow error types and locations. This information can be used to help track down the bug in the register allocator that caused the error.

5.3 Dynamic Checking of Optimization Outputs

Jaramillo et al. [2002] proposed to dynamically check the register allocator's output code with annotations. Annotations are inserted by the compiler in both the input and output code of the register allocator. Programs with and without register allocation are run at the same time and the corresponding values from the two programs are compared at the annotated instructions. Errors are reported when the values differ. However, unlike SARAC, this dynamic approach can not guarantee 100% coverage of a program; some errors along a specific path may not be detected. With this approach, the program must be run many times under different test data inputs to ensure a good coverage. Although they described this approach, it is not implemented and evaluated.

5.4 Data Flow Analysis in Related Areas

Data flow analysis has been used to check/verify program properties, which are related to our approach. The work by Dwyer and Clarke [1994] used data flow analysis to verify concurrent code properties. However, their work does not check the correctness of register allocation. Data flow analysis has also been applied to source-level debugging, including debugging code optimized with register allocation [Adl-Tabatabai and Gross 1993, 1996; Jaramillo et al. 2002; Wismueller 1994]. Jaramillo et al. [2002] used data flow analyses, including reachability and postdominance, to determine where and what annotations to use in their comparison checking techniques for optimized code. Adl-Tabatabai and Gross [1993, 1996] considered the aspects of global register allocation on the residency problem, which determines if a variable is in its assigned register at a breakpoint. Wismueller [1994] also utilized data flow analysis to ensure the correct debugger behavior with optimized programs, including register allocation. Compared with SARAC, these approaches use data flow analysis to relate a runtime value to a source variable, allowing the source program to be debugged. SARAC is used to check the correctness of the register allocator's output code.

6. SUMMARY AND FUTURE WORK

This article describes SARAC, which is a new approach to catch and identify bugs in register allocation. The approach statically checks that the value flow of the input code to the register allocator is maintained in the output code, given that the register allocator performs limited code edits (including register coalescing). SARAC is accurate and fast. The approach can be extended to handle rematerialization and register aliasing. A prototype tool (*ra-analyzer*) shows that our approach has minimal compile-time and memory overhead. The

performance and memory usage of *ra-analyzer* scale well with function body size, number of operands, and CFG complexity.

A goal for future work is to make *ra-analyzer* stand-alone 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 types and architectural constraints can be a common error source in a register allocator. Another issue is how to connect the tool to different compilers and intermediate representations. A final issue in making SARAC stand-alone is to develop a way to describe machine-dependent information to the tool about the registers.

REFERENCES

ADL-TABATABAI, A. AND GROSS, T. 1993. Evicted variables and the interaction of global register allocation and symbolic debugging. In *Proceedings of the Symposium on Principles of Programming Languages*.

ADL-TABATABAI, A. AND GROSS, T. 1996. Source-Level debugging of scalar optimized code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

BARTHE, G., GREGOIRE, B., KUNZ, C., AND REZK, T. 2006. Certificate translation for optimizing compilers. In *Proceedings of the 13th International Static Analysis Symposium*.

BLECH, J. O. AND GREGOIRE, B. 2008. Certifying code generation runs with Coq: A tool description. In *Proceedings of the 7th International Workshop on Compiler Optimization Meets Compiler Verification*.

BENITEZ, M. E. AND DAVIDSON, J. W. 1988. A portable global optimizer and linker. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

BERNSTEIN, D., GOLUMBIC, M., MANSOUR, Y., PINTER, R., GOLDIN, D., KRAWCZYK, H., AND NAHSHON, I. 1989. Spill code minimization techniques for optimizing compilers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

BRADLEE, D. G., EGGERS, S. J., AND HENRY, R. R. 1991. Integrating register allocation and instruction scheduling for RISCs. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*.

BRIGGS, P., COOPER, K. D., AND TORCZON, L. 1992. Rematerialization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

BRIGGS, P., COOPER, K. D., AND TORCZON, L. 1994. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst. 3*, 16, 428–455.

CHAITIN, G. J. 1982. Register allocation and spilling via graph coloring. In *Proceedings of the Symposium on Compiler Construction.*

CHOW, F. C. AND HENNESSY, J. L. 1990. The priority-based register allocation coloring approach. *ACM Trans. Program. Lang. Syst. 4*, 12, 501–536.

COOPER, K. D. AND LU, J. 1997. Register promotion in C programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

CPU2000 BENCHMARK. Standard Performance Evaluation Corporation (SPEC). http://www.spec.org.

DAVIDSON, J. W. AND FRASER, C. W. 1984. Register allocation and exhaustive peephole optimization. *Softw. Pract. Exper. 14*, 9, 857–865.

DOR, N., ADAMS, S., DAS, M., AND YANG, Z. 2004. Software validation via scalable path-sensitive value flow analysis. In *Proceedings of the International Symposium on Software Testing and Analysis*.

DWYER, M. B. AND CLARKE, L. A. 1994. Data flow analysis for verifying properties of concurrent programs. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*.

GCC. GCC homepage. http://gcc.gnu.org/.

GEORGE, L. AND APPEL, A. W. 1996. Iterated register coalescing. *ACM Trans. Program. Lang. Syst. 3*, 18, 300–324.

GUPTA, R., SOFFA, M. L., AND STEELE, T. 1989. Register allocation via clique separators. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

JARAMILLO, C. S., GUPTA, R., AND SOFFA, M. L. 2002. Verifying optimizers through comparison checking. In *Proceedings of the International Workshop on Compiler Optimization Meets Compiler Verification*.

LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. 1997. MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*.

LEE, K. D. 2003. A formally verified register allocation framework. *Electron. Not. Theor. Comput. Sci. 82,* 3.

LERNER, S., MILLSTEIN, T., AND CHAMBERS, C. 2003. Automatically proving the correctness of compiler optimizations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

LERNER, S., MILLSTEIN, T., RICE, E., AND CHAMBERS, C. 2005. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Proceedings of the Symposium on Principles of Programming Languages*.

LEROY, X. 2006. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *Proceedings of the Symposium on Principles of Programming Languages*.

LI, G., OWENS, S., AND SLIND, K. 2007. Structure of a proof-producing compiler for a subset of higher order logic. In *Proceedings of the 16th European Symposium on Programming (ESOP'07)*. Lecture Notes in Computer Science, vol. 4421. Springer, 205–219.

MCNERNEY, T. M. 1991. Verifying the correctness of compiler transformations on basic blocks using abstract interpretation. In *Proceedings of the ACM/SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*.

MIBENCH. University of Michigan. http://www.eecs.umich.edu/mibench/.

NANDIVADA, V. K., PEREIRA, F. M. Q., AND PALSBERG, J. 2007. A framework for end-to-end verification and evaluation of register allocators. In *Proceedings of the 14th International Static Analysis Symposium*.

NECULA, G. C. 2000. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

NECULA, G. C. AND LEE, P. 1998. The design and implementation of a certifying compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

PEREIRA, F. M. Q. 2006. Static validation of register allocation. http://compilers.cs.ucla.edu/ralf/publications/Pereira06b-tech-report.pdf.

PINTER, S. S. 1993. Register allocation with instruction scheduling: A new approach. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

PNUELI, A., SIEGEL, M., AND SINGERMAN, F. 1998. Translation validation. In *Proceedings of the 4th Conference on Tools and Algorithms for Construction and Analysis of Systems*.

POLETTO, M. AND SARKAR, V. 1999. Linear scan register allocation. *ACM Trans. Program. Lang. Syst. 5*, 21, 895–913.

RINARD, M. C. 1999. Credible compilation. Tech. rep. MIT-LCS-TR-776, Massachusetts Institute of Technology. March.

RIVAL, X. 2004. Symbolic transfer function-based approaches to certified compilation. In *Proceedings of the Symposium on Principles of Programming Languages*.

SANTHANAM, V. AND ODNERT, D. 1990. Register allocation across procedure and module boundaries. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

SMITH, M. D. AND HOLLOWAY, G. Machine SUIF. http://www.eecs.harvard.edu/hube/research/machsuif.html.

SMITH, M. D., RAMSEY, N., AND HOLLOWAY, G. 2004. A generalized algorithm for graph-coloring register allocation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

STEFFEN, B., KNOOP, J., AND RUTHING, O. 1990. The value flow graph: A program representation for optimal program transformations. In *Proceedings of the 3rd European Symposium on Programming (ESOP)*. Lecture Notes in Computer Science, vol. 432. Springer, 389–405.

WISMUELLER, R. 1994. Debugging of globally optimized programs using data flow analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Received April 2007; revised October 2009; accepted October 2009