A Practical, Robust Method for Generating Variable Range Tables*

Caroline Tice[1] and Susan L. Graham[2]

[1] Compaq Systems Research Center
caroline.tice@compaq.com
[2] University of California, Berkeley
graham@cs.berkeley.edu

Abstract. In optimized programs the location in which the current value of a single source variable may reside typically varies as the computation progresses. A debugger for optimized code needs to know all of the locations – both registers and memory addresses – in which a variable resides, and which locations are valid for which portions of the computation. Determining this information is known as the *data location problem.* Because optimizations frequently move variables around (between registers and memory or from one register to another) the compiler must build a table to keep track of this information. Such a table is known as a *variable range table.* Once a variable range table has been constructed, finding a variable's current location reduces to the simple task of looking up the appropriate entry in the table.
The difficulty lies in collecting the data for building the table. Previous methods for collecting this data depend on which optimizations the compiler performs and how those optimizations are implemented. In these methods the code for collecting the variable location data is distributed throughout the optimizer code, and is therefore easy to break and hard to fix. This paper presents a different approach. By taking advantage of *key instructions*, our approach allows the collection of all of the variable location data in a single dataflow-analysis pass over the program. This approach results in code for collecting the variable location information that is easier to maintain than previous approaches and that is almost entirely independent of which optimizations the compiler performs and of how the optimizations are implemented.

1 Introduction

A correct, accurate symbol table is critical for the interactive source-level debugging of optimized code. The symbol table contains information about all the

* This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Contract No. F30602-95-C-0136 and Grant No. MDA972-92-J-1028, by the National Science Foundation under Infrastructure Grant Nos. EIA-9802069 and CDA-9401156, by an endowment for the Chancellor's Professorship Award from the University of California, Berkeley, and by Fellowship support from the GAANN. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

symbols (names) that occur in the source program, including object names, type names, local and global variables, subroutine names and parameters. Symbol tables also keep information about the declared types of program variables and about how to map between locations in the source program and locations in the target program. An especially important piece of information kept in the symbol table is information about the places where variable values reside during program execution. Debuggers need accurate information about variable locations, both in memory and in registers, in order to determine where to look when a user asks to see the value of a variable during the execution of the program.

The relationship that exists between a source program and its corresponding *unoptimized* binary program makes it fairly straightforward to collect and generate symbol table information. For example, whenever the value of a source variable is updated, the new value is written to the variable's home location in memory. Compiler optimizations destroy these relationships, making the task of determining a variable's current location much harder. In an optimized program, the location that contains a variable's correct current value can vary as the computation proceeds, sometimes being one of several different registers; sometimes being one of several locations in memory; sometimes being a constant value encoded directly in the instructions; and sometimes, during parts of the computation where the variable is not live, not being anywhere at all. Determining where to find a variable's value is known as the *data location problem* [3,7].

In order to correctly capture information about variables' locations, the compiler needs to construct a *variable range table*, such as the one described by Coutant, Meloy, and Ruscetta [3]. This table associates, with every variable in the source program, a list of "entries". Each entry in the list gives a location in which that variable resides during some portion of the computation. Each location entry in the table also has associated with it the range(s) of addresses in the binary program for which the entry is valid.

The idea of a variable range table is not new; it has existed for over a decade. The difficulty is that, until now, there has been no easy way to collect this information. The standard approach is to track every source variable through every optimization performed by the compiler, recording each location as the variable gets moved around. Thus the data evolves as the optimizations are performed, and when the optimizations are complete the data has been "collected".

This approach has many disadvantages. The greatest of these is that the code for collecting the variable range table information is distributed throughout the code for the optimizations. Consequently these implementations are very fragile, because any change to the optimizations (adding an optimization, removing an optimization, or changing the implementation of an optimization) can break this code. In addition errors in the code for collecting variable location data are very hard to track down and fix, because the code is not all in one place.

In this paper we present a completely different technique for collecting this important information. By using dataflow analysis on the optimized binary program we are able to collect the variable location information in one pass. Our approach is independent of which optimizations the compiler performs and of how the optimizations are implemented, although it does rely on accurate information about key instructions, as explained later. Our approach allows all the source code for collecting the location information to be in one place, rather than distributed throughout the optimizer source code in the compiler. This makes it simpler to find and fix errors in the location-collection code. The code is potentially more efficient than previous approaches, because the data is collected only once, not collected and then updated continually as optimizations are performed.

Our approach builds on the realization that the final optimized binary program must contain all the required information as to where variables are located. The optimized binary program encodes the final results of all the optimizations, including the low-level optimizations such as instruction scheduling, register allocation, and spilling. Therefore by performing dataflow analysis on the final optimized binary program and discovering where the location of a variable changes, we can obtain completely accurate and correct information as to the locations of all the program variables, only performing the analysis *once*, and without having to trace through each optimization individually.

So why has no one done this before? In order to perform the dataflow analysis on the instructions, one needs to know which instructions correspond to assignments to source program variables, and, for those instructions, which variable is receiving the assignment. As explained in the next section, this critical piece of information cannot be determined from the instructions by themselves. But we can take advantage of *key instructions* in order to obtain this vital information.

The rest of this paper is organized as follows: In Section 2 we describe our method for constructing the variable range table for an optimized program. We briefly describe our implementation experiences in Section 3, and in Section 4 we present our conclusions.

2 Using the Optimized Binary

The key to knowing where the current value of a variable resides at runtime is identifying the places in the target program where the value of a source variable gets stored, either because the variable has been copied to a new location, or because it has been given a new value. In this section, we explain how key instructions are used to find those places in the target program, and how dataflow analysis on those places is used to construct the variable range table.

2.1 The Role of Key Instructions

The notion of key instructions was originally introduced in order to solve a different problem relating to debugging optimized code, the *code location problem* [3,5,7]. This problem is determining how to map between locations in the source program and corresponding locations in the optimized target program. This mapping is needed for implementing many basic debugger functions, such as single-stepping and setting control breakpoints. The locations in the source program that are of interest are those that result in a source-level-visible state change (i.e. changing either the value of a source variable or the flow of control through the program). Each piece of program source code (e.g., statement, expression, subexpression, etc.) that causes a single potential source-level-visible state change is called an *atom*. Some source statements contain only a single atom, but others contain multiple atoms. Every atom in the source program has a corresponding *key instruction* in the optimized target program.[1] Intuitively, the key instruction for any atom is the single instruction, in the set of instructions generated from that atom, that most closely embodies the semantics of the atom. In particular, of the set of instructions generated for the atom, the key instruction is the first instruction reached during program execution that causes a source-level-visible state change to the program. By definition every atom that has not been eliminated by the optimizer must have such an instruction.

Because we are concerned here with assignments to source variables, we focus on assignment atoms and their key instructions. Since there is a one-to-one mapping between atoms and visible state changes in the source, any atom can assign a value to at most one source variable.[2] Therefore the key instruction for an assignment atom is the instruction that writes the value of the right-hand side of the assignment to the variable's location in memory; or, if the write to memory has been eliminated, it is the final instruction that evaluates the right-hand side into a register. If optimizations cause code for an atom to be duplicated, such as unrolling a loop some number of times, the atom will have one key instruction for each copy of the duplicated code that remains in the final program.

2.2 Internal Representation

```
i = start - 1;          LOAD start; LOADI 1; SUB; STORE i;
while (!(done)) {       LOADI 0; LOAD done; WHILE NEQ;
                        LOOP_BODY
  i++;                  LOAD i; LOADI 1; ADD; STORE i;
  if (i >= j)           LOAD i; LOAD j; IF GEQ;
                        TRUEBR
    done = 1;           LOADI 1; STORE done;
                        FALSEBR
}                       END_LOOP

 (a) C code             (b) Early Internal Representation
```

Fig. 1. Sample C code and its equivalent early internal representation

Throughout this paper we refer to the *early internal representation* and to the *final internal representation* generated by the compiler. The *early internal*

[1] If code has been duplicated, a single atom may have multiple key instructions.

[2] For simplicity, this discussion assumes there are no constructs in the source, such as a swap statement, that simultaneously update multiple source variables.

representation is a very high-level set of instructions. While slightly "below" the level of the programming language (it uses explicit instructions for many things that are implicit in the programming language), it has constructs such as `if`, `while-do`, and `switch`. It also uses variable names to refer to variables that occur in the program source. Figure 1 shows a small piece of C code and how the code might be translated into this early internal representation. The early internal representation uses postfix operator conventions.

As the compilation proceeds this high-level representation gets gradually "lowered". During this lowering process it is common for a single high-level instruction to be broken into multiple lower-level instructions. At the end of the compilation process the program is in the *final internal representation*, from which the binary code is directly generated. In the compiler we used, this internal representation is the assembly code for the RISC instruction set of the MIPS processor. Figure 2 shows some common instructions from this set.

Type	Example	Semantics
Register Copy	`or $16, $4, $0`	or $4 with $0 (always constant 0) and store in $16. This is the same as to copying $4 into $16
	`mov $4, $16`	copy $4 into $16
Load Address	`lda $8, 24($sp)`	$8 gets the address '$sp + 24'
Load	`ld  $17, 32($sp)`	$17 gets contents of memory at location '$sp + 32'
Store	`sd  $31, 8($sp)`	value in $31 is written to memory at location '$sp + 8'
Addition	`addiu $sp, $sp, -80`	$sp gets the contents of $sp added to -80 (an address)
	`add  $16, $4, $11`	$16 gets the sum of the contents of $4 and $11
Subtraction	`subiu $sp, $sp, 80`	$sp gets the contents of $sp minus 80 (an address)
	`sub $7, $8, $5`	$7 gets the value of the contents of $8 minus the contents of $5

Fig. 2. Examples of instructions in the final internal representation

2.3 Identifying Key Instructions

We assume that the compiler keeps accurate information throughout the compilation process as to the source position (file, line, and column position) from which every piece of internal representation, both early and late, was generated. In the final representation, every instruction has associated with it the source position(s) from which it was generated. The compiler must maintain this basic information to make source-level debugging of the optimized program feasible.

Using the source position associated with each instruction, we can identify the set of all instructions generated from any given atom. Once we have this set for each atom we need to find the key instruction within the set. Identifying key instructions for control flow atoms is not too difficult: The key instruction for a control flow atom is the first conditional branching instruction in the set of instructions generated from that atom.[3] Key instructions for assignment atoms, however, are much harder to identify. In fact they cannot be identified at all, if one has only the set of instructions generated from the atom.

There are three factors that contribute to this difficulty. First, one cannot identify assignment key instructions by the opcode of the instruction. An assignment key instruction might be a store instruction, but it could as easily be an add, subtract, multiply, or any other operator instruction (if the store to memory has been optimized away). Second, one cannot use variable names to determine the key instruction for an assignment, because there are no variable names in the instructions. Third, one cannot rely on the position of the instruction to identify assignment key instructions. While it is true that, in the absence of code duplication, the key instruction for an assignment atom will be the last non-nop instruction for the atom, optimizations such as loop unrolling may result in an assignment atom having multiple key instructions within a single basic block, making it impossible to use instruction position to find them all. In fact the only way to identify all the key instructions for an assignment atom is to perform a careful semantic analysis of both the instructions and the original source atom.

Luckily the compiler *does* perform such an analysis early in the compilation, when it parses the program and generates the early internal representation. It makes sense to take advantage of the compiler and have the front end tag the piece of early internal representation that will become the key instruction(s) for an assignment atom. These tags can then be propagated appropriately through the compiler as the representation is lowered, instructions are broken down, and optimizations are performed. When the final instructions are generated, the key instructions for assignment atoms are already tagged as such.

We claimed earlier that our approach to collecting variable location data is mostly independent of the optimization implementations. It is not completely independent because the key instruction tags have to be propagated through the various optimization phases. We feel this is acceptable for two reasons. First it is far simpler to propagate the one-bit tag indicating that a piece of internal representation corresponds to an assignment key instruction, than to track and record information about variable names and current variable storage locations. Second, and more importantly, we assume that key instructions will be tagged anyway, as a necessary part of solving the code location problem, without which one cannot implement such basic debugger functions as setting control breakpoints and single-stepping through the code. Therefore we view our use of key instructions to collect the variable range table information as exploiting an existing mechanism, rather than requiring a new, separate implementation.

[3] By “first” we mean the first instruction encountered if one goes through the instructions in the order in which they will be executed.

For more information about key instructions, including formal definitions, algorithms for identifying them, and an explanation of how they solve the code location problem, see Tice and Graham[5].

2.4 Collecting Names of Assigned Variables

Recall that, in order to perform the dataflow analysis on the binary instructions, we need to know which instructions assign to source variables, and which variables receive those assignments. The key instruction tags on the instructions identify the assignment instructions, but we still need to know the name of the variable that receives each assignment. The phase to collect that information is executed fairly early in the back end of the compiler, before variable name information has been lost. In this phase a single pass is made over the early representation of the program. Every time an assignment is encountered, it is checked to see if the assignment is to a source-level variable.[4] If so, the name of the variable and the source position of the assignment atom are recorded in a table. Thus at the end of this pass we have created a table with one entry for every assignment atom in the source program. Each entry contains the name of the variable receiving the assignment and the source position of the assignment atom. We attempt to make the variable name information collected in this phase as precise as possible, by recording assignments to components of variables such as fields or array elements.

2.5 Performing the Dataflow Analysis

A forward dataflow analysis is performed at the very end of the compilation process, after *all* optimizations (including instruction scheduling) have finished. The data items being created, killed, etc. by this dataflow analysis are *variable location records*, where a variable location record is a tuple consisting of a variable name, a location, a starting address, and an ending address. For the rest of this paper we will refer to such a tuple as a *location tuple.*

```
Location Tuple Format:
      < name , location, starting address, ending address>

Example Location Tuples:

      1: < ‘x’,  $4,  0x1000A247,  undefined >
      2: < ‘p.foo’, Mem[$sp + $8], 0x1000A216, 0x1000A4BC >
```

Fig. 3. Location Tuples

[4] The key instruction tags can be used to tell whether or not an assignment is to a source variable.

Figure 3 shows the format of a location tuple and gives two examples. Example 1 is a location tuple for variable *x*. It indicates that the value for *x* can be found in register four ($4) while the program counter is between 0x1000A247 (the starting address for this tuple) and some undefined ending address. The ending address is undefined because the tuple shown in Example 1 is still live in the dataflow analysis. Once the tuple has been killed, the ending address will be filled in, as explained below. Example 2 is a tuple indicating the value for *p.foo* can be found in the memory address resulting from adding the contents of register eight ($8) to the stack pointer ($sp) when the program counter is between 0x1000A216 and 0x1000A4BC.

When a location tuple is created, it receives the name of the source variable being updated or moved and the new current location for the variable (either a register or a memory address). The starting address is the address of the current instruction, and the ending address is undefined. When a location tuple is killed during the dataflow analysis, the ending address in the location tuple is filled in with the address of the killing instruction. Killed location tuples are not propagated further by the dataflow analysis, but they are kept. At the end of the dataflow analysis, all location tuples will have been killed. The data in these location tuples is then used to construct the variable range table.

We perform the dataflow analysis at the subroutine level. The initial set of location tuples for the dataflow analysis contains one location tuple for each formal parameter of the subroutine. (The compiler knows the initial locations for the parameters). It also contains one location tuple for each local variable to which the compiler has assigned a home location in memory. These local variable location tuples start with a special version of the variable name that indicates that the variable is uninitialized. When the variable is first assigned a value, a new location tuple is created for it. This allows us to determine those portions of the program where a variable is uninitialized, which in turn makes it easy for the debugger to warn a user who queries the value of an uninitialized variable.

In most respects the details of our dataflow algorithm are quite standard [2]. But the rules for creating and killing location tuples inside a basic block deserve comment. Although we have focused so far on instructions that update the values of source-level variables, in fact *every* instruction in the basic block must be examined carefully when performing the dataflow analysis. The following paragraphs enumerate the different types of instructions that require some kind of action and explain what actions are appropriate. Appendix A gives our algorithm for examining instructions within basic blocks.

In Figure 2 the reader can see all the various types of instructions discussed below except for key instructions. A key instruction is simply a regular instruction with an extra bit set.

Key Instructions. First we need to determine if the key instruction is for an assignment atom or not. We can determine this by comparing the source position associated with the key instruction with the source positions in the name table collected earlier (see Section 2.4). If the source position is not in the table, we know the key instruction is not for an assignment atom, so we treat it like any other instruction. If the source position is found in the table, then we use the table to get the name of the variable receiving the assignment. We kill any other location tuples for that variable. We also kill any location tuple whose location is the same as the destination of the assignment instruction. Finally we generate a new location tuple for the variable, starting at the current address, and with a location corresponding to the destination of the assignment instruction.

Register Copy Instructions. If the instruction copies contents from one register to another, we first kill any location tuples whose locations correspond to the destination of the copy. Next we check to see if the source register is a location in any of the live location tuples. If so, we generate a new location tuple copying the variable name from the existing location tuple. We make the starting address of the new location tuple the current address, and set the location to the destination of the register copy. This can, of course, result in the same variable having multiple live location tuples, at least temporarily.

Load Instructions. For load instructions, we check the memory address of the load to see if it corresponds to any of the home addresses the compiler assigned to any local variables or to any location in a live location tuple (i.e. a spilled register). In either case we generate a new location tuple for the variable, starting at the current address, and using the destination of the load as the location. We also kill any existing live location tuples whose locations correspond to the destination of the load.

Store Instructions. First we kill any location tuples whose locations correspond to the destination of the store. Next we check the address of the store to see if it corresponds to an address the compiler assigned to a variable, in which case we generate a new location tuple for the variable. We also generate a new location tuple if the source of the store corresponds to a location in any live location tuple (i.e. a register spill).

Subroutine Calls. For subroutine calls we kill any location tuple whose location corresponds to subroutine return value locations.

All Other Instructions. If the instruction writes to a destination, we check to see if the destination corresponds to any location in any live location tuple. Any such location tuples are then killed. (The corresponding variable's value has been overwritten.)

2.6 Handling Pointer Variables

The actions outlined above do not take pointer variables into consideration. Pointers require slightly more work, but the basic ideas are fairly straightforward. In order to correctly gather and maintain location information for pointer variables, we take the following actions, in addition to those already described above. The following explanations use C syntax in particular, but the concepts and actions are generalizable for other languages.

Load Instructions. If the base register for a load from memory instruction corresponds to the location in any live location tuple T, and the load offset is zero, we generate a new live location tuple. The location in the new location tuple is the destination of the load instruction. The variable name in the new tuple,

V_1, is a version of the variable name V that occurs in T, modified as follows. If V begins with at least one ampersand ('&'), we remove one ampersand from the front of V to construct V_1. If V does not have at least one ampersand at the beginning, we add an asterisk ('*') to the front of V to construct V_1.

Store Instructions. If the base register for a store to memory instruction corresponds to the location in any live location tuple T, and the store offset is zero, we generate a new location tuple. The location in the new tuple is "Memory[$R]", where "$R" is the base register of the store instruction. The variable name in the new tuple, V_1, is a version of the variable name V that occurs in T, modified as follows. If V begins with an ampersand, remove one ampersand from the front of V to construct V_1. Otherwise add one asterisk to the front of V to construct V_1. In addition to creating the new location tuple, we also need to look for any live location tuples whose variable names could be constructed either by removing one or more ampersands from V or by adding one or more asterisks to V. Any such tuples must be killed, since the value stored in the location represented by our new location tuple will invalidate them.

Load Address Instructions. If the instruction set includes special instructions for loading addresses into registers, these instructions also require special actions. If the "base register + offset" used in the address calculation corresponds to a memory address in any live location tuple, or to the home location the compiler assigned to any local variable, we generate a new location tuple. The location in the new tuple is the destination of the load address instruction. The variable name is '&V', where V is the name of the variable whose home location or live location tuple location corresponds to the "base register + offset" used in the address calculation.

Integer Addition/Subtraction Instructions. If one operand of an integer addition or subtraction instruction is a register containing a memory address, treat the instruction exactly like the load address instructions, as explained above, using the other operand as the "offset". A register contains a memory address if it is one of the special registers that always address memory, such as $sp or $gp, or if it is the location in a live location tuple whose variable name begins with an ampersand.

2.7 Building the Variable Range Table

Once the dataflow analysis is complete, we have a large amount of data that needs to be combined into the variable range table. To make the range table as small as possible, we combine and eliminate location tuples wherever this is possible and reasonable. We also compute, for every variable, all the address ranges for which the variable does not have any location. Figure 4 traces an example of this process.

Figure 4(a) shows an example of raw location tuples collected for a variable x. In order to consolidate the data we first sort the location tuples by variable name. For each variable, the location tuples for that variable need to be sorted by location (Figure 4(b)). Any location tuples for a given variable and location that have consecutive ranges need to be combined. For example, the location tuples

```
<x, $6, 64, 92>
<x, $11, 24, 28>
<x, M[$sp+48], 84, 100>
<x, $11, 28, 36>
<x, $6, 32, 48>
<x, $11, 12, 24>

      (a)
```

```
<x, M[$sp+48], 84, 100>
<x, $6, 64, 92>
<x, $6, 32, 48>
<x, $11, 24, 28>
<x, $11, 28, 36>
<x, $11, 12, 24>

      (b)
```

```
<x, M[$sp+48], 84, 100>
<x, $6, 64, 92>
<x, $6, 32, 48>
<x, $11, 12, 36>

      (c)
```

```
<x, $11, 12, 36>
<x, $6, 32, 48>
<x, $6, 64, 92>
<x, M[$sp+48], 84, 100>

      (d)
```

```
<x, $11, 12, 32>
<x, $6, 32, 48>
<x, $6, 64, 84>
<x, M[$sp+48], 84, 100>

      (e)
```

```
<x, uninit, 0, 12>
<x, $11, 12, 32>
<x, $6, 32, 48>
<x, evicted, 48, 64>
<x, $6, 64, 84>
<x, M[$sp+48], 84, 100>
      (f)
```

Fig. 4. Trace for constructing variable range table information from raw location tuple data.

⟨x, \$4, 1, 7⟩ (translated as "variable `x` is in register 4 from address one through address seven") and ⟨x , \$4, 7, 10⟩ ("variable `x` is in register 4 from address seven through address ten") can be combined into ⟨x, \$4, 1, 10⟩ ("variable `x` is in register 4 from address one through address ten"). Figure 4(c) shows consecutive ranges for x in register 11 combined into a single tuple.

After all such consecutive location tuples have been combined, we sort all of the location tuples for each variable (regardless of location) by starting address (Figure 4(d)). For some address ranges a variable may be in multiple locations, while for other address ranges a variable may not be in any location. For those address ranges where a variable has multiple locations, we select one location (usually the one with the longest address range) to be the location we will record in the variable range table. We also shorten other ranges, to eliminate overlap. This is illustrated in Figure 4(e), where the ranges of x in registers \$r11 and \$r6 were shortened. This is an implementation decision, rather than a feature of our algorithm. Admittedly by not recording all locations for a variable we are losing some information. If the debugger for optimized code were to allow users to update the values of variables, then the information we are losing would be critical. We are assuming, for this work, that users are not allowed to update source variables from inside the debugger once a program has been optimized, as such an update could invalidate an assumption made by the compiler, which in turn could invalidate an optimization, and thus make the program start behaving incorrectly. Updating variables after optimizations have been performed is an open research issue, and is beyond the scope of this paper. For those address ranges for which a variable does not have any location, we create an "evicted" record for the variable and put that in the variable range table.[5] Thus an additional benefit of this approach is that we can automatically determine not only the correct locations of variables, but also the address ranges for which variables are uninitialized or evicted. Figure 4(f) shows the final set of location tuples for x, which will be written to the variable range table.

For location tuples whose variable names are of the form `array[exp]`, or which begin with one or more asterisks, we do not create any eviction records. Also for variables of the latter type, we do not keep location tuples whose location is of the form "Memory[exp]", if there are also tuples, covering the same address ranges, whose location is "exp", and whose name is constructed by removing all the asterisks from the front of the variables in question. For example, if we know that pointer variable p is in register sixteen, there is no point in keeping the information that **p* can be found in Memory[$16]. On the other hand, knowing that **p* can also be found in register seven is useful, so such information is kept.

3 Implementation Experience

We implemented a prototype of our approach for building a variable range table inside the SGI Mips-Pro 7.2 C compiler, a commercial compiler that optimizes aggressively. This compiler consists of nearly 500,000 lines of C and C++ source code. The symbol table format used is the DWARF 2.0 standard format [4]. Since we were implementing an entire solution for debugging optimized code, we modified the compiler to identify key instructions as well as to generate the variable range table information. We also modified a version of the SGI dbx debugger to use the resulting symbol table for debugging optimized code.

Modifying the compiler to collect the variable names, to perform the dataflow analysis and to write the variable range table into the symbol table took roughly 2 months, and required modifying or writing approximately 1,500-2,000 lines of code. Modifying the debugger to use the variable range table to look up a variable's value took about 2 weeks and 300-500 lines of code. As these numbers show, it is quite easy to adapt existing compilers and debuggers to create and use this variable range table. The numbers here do not include the time it took to implement the key instruction scheme, which took roughly five months and involved 1,500-2,000 lines of code. A large part of the five months was spent becoming familiar with the compiler code.

Once our prototype was fully implemented, we made it available for use to some software engineers at SGI and to some computer science graduate students at the University of California, Berkeley. In the end, sixteen people tested our prototype and gave us feedback on it. Overall the feedback was positive, several people expressing the wish they had had this tool a few months earlier, as it would have helped them find bugs they had been working on. For more details on our user feedback, see Tice [6].

[5] Evicted variables are part of the *residency problem* identified by Adl-Tabatabai and Gross [1].

Prior to the implementation of our prototype, the SGI compiler, in common with most commercial compilers, did nothing in particular to facilitate debugging optimized code. Its default mode was to turn off optimizations if debugging was requested. Although it was possible to override this default through the use of certain compilation flags, the debugging information was collected and stored in the symbol table in exactly the same manner as if no optimizations were being performed. The result of this was that data in the symbol table for optimized programs was often incomplete, incorrect, and/or misleading, particularly with respect to the locations of variables.

When implementing our dataflow analysis we encountered a small problem. As in standard dataflow analysis algorithms, the in-set for each basic block was constructed by joining all out-sets of the predecessor basic blocks. Occasionally when joining sets of locations from predecessor blocks we found inconsistencies between the predecessors. Two different basic blocks might indicate the same variable being in two conflicting locations, as shown in Figure 5. At the end of basic block BB2, variable `v` is in register `$r4`, while at the end of basic block BB3, `v` is in register `$r7`.[6] The question is where, at the beginning of basic block BB4, should we say that `v` is? The answer would appear to be "it depends on the execution path". However the execution path is something we cannot know at compile time, when we are constructing the variable range table.

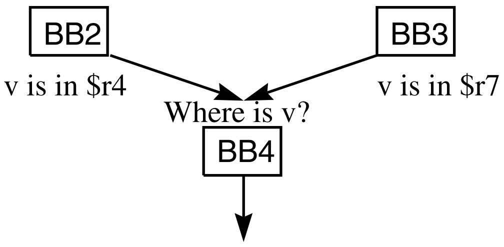

Fig. 5. Inconsistency problem during dataflow analysis

To deal with this problem, we chose to "kill" both of the conflicting location tuples at the end of the basic blocks from which they came. If there are no location tuples for the variable that do not conflict at this point, the range table will indicate the variable as being *evicted*, i.e. not residing anywhere. This choice was reasonable because the problem in Figure 5 typically arises only when `v` is dead on entry to BB4. Otherwise the compiler would require definitive information as

[6] Note that it would be fine if BB2 and BB3 both reported that `v` was stored both in `$r4` and `$r7`.

to where it could find the variable.[7] Since the variable is dead, killing its location tuple (evicting it, in effect) seems a reasonable, simple solution.

An alternative approach would be to tag the conflicting location tuples in some special way. In the final variable range table all the conflicting alternative locations could be entered, leaving the correct resolution up to the debugger or the debugger user.

Our simple approach was particularly attractive in the context of Optdbx [6], the debugger for which it was built, because Optdbx performs eviction recovery, capturing the values of variables before the compiler overwrites them. This technique allows the debugger to recover the correct variable value in these cases. Using our example as illustration, whenever execution reaches either the end of basic block BB2 or basic block BB3, the debugger will notice that `v` is about to be evicted and so will cache the current value of `v` in a special table. If the user suspends execution at the top of basic block BB4 and asks to see the value of `v`, the debugger will first look in the range table, where it will see that `v` is currently evicted. It will then get the value of `v` from its special table.

The preceding discussion illustrates one of the added benefits of using dataflow analysis to construct the variable range table: one can obtain (with no extra work) exact information about variable evictions, uninitialized variables, and variables that have been optimized away. (The last case would be those variables for which no location tuples were created.) Users of our debugger especially liked its ability to indicate these special cases.

4 Conclusion

In this paper we have presented a new method for collecting the variable location information to be incorporated into a variable range table. This is necessary for solving the data location problem, allowing debuggers of optimized code to determine where variable values reside during execution of the program. By taking advantage of key instruction information, our method uses dataflow analysis techniques to collect all the variable location information in a single pass over the final optimized internal representation of the program.

Our approach has many advantages over previous approaches. It is potentially faster than previous approaches, because it collects the data in one dataflow pass, rather than building an initial set of data and evolving the set as optimizations are performed. Unlike previous approaches the person implementing this technique does not need to understand how all of the optimizations in the compiler are implemented. In previous approaches the source code for collecting the variable location data has to be distributed throughout the optimizer code; any time an optimization is added, removed, or modified, the code for collecting the variable location data must be modified as well. By contrast our approach is completely independent of which optimizations the compiler performs and of

[7] It is possible to construct pathological cases for which this assumption is false; however we do not believe such pathological cases ever actually arise inside compilers, and therefore we feel it is reasonable to make this assumption.

how those optimizations are implemented. The code for performing the dataflow analysis is all in one place, making it easier to write, to maintain, and to debug.

A further benefit of using dataflow analysis on the final representation of the program to collect the variable location data is that this also allows for easy identification of those portions of the target program for which any given variable is either uninitialized or non-resident. Previous variable range tables have not contained this information at all.

Finally by implementing these ideas within an existing commercial compiler and debugger, we have shown that these ideas work and that they can be retrofitted into existing tools without too much effort.

A Algorithm for Generating, Passing, and Killing Location Tuples within a Basic Block

```
currentSet $\leftarrow \bigcup_{p \in predecessor(currentBB)}$ p.locations
i $\leftarrow$ first instruction for basic block
do {
  if (i has a destination)
    forall l in currentSet
      if (l.location = i.destination)
        kill(l)
      fi
    endfor
  fi

  if (i is an assignment key instruction)
    varname $\leftarrow$ name_lookup(i.source_position)
    forall l in currentSet
      if (l.varname = varname)
        kill(l)
      fi
    endfor
    newRecord $\leftarrow$ gen (varname, i.dest, i.address, *undefined*)
    currentSet $\leftarrow$ currentSet $\cup$ { newRecord }
  else if (i is a register copy)
    forall l in currentSet
      if (l.location = i.source)
        newRecord $\leftarrow$ gen (l.varname, i.dest, i.address, *undefined*)
        currentSet $\leftarrow$ currentSet $\cup$ { newRecord}
      fi
    endfor
  else if (i is a function call)
    forall l in currentSet
      if (l.location = return value register)
        kill(l)
      fi
    endfor
  else if (i is a memory read)
```

```
        if (i.memory_location is "home" address of variable)
          newRecord ← gen (name of var, i.dest, i.address, undefined)
          currentSet ← currentSet ∪ { newRecord }
        else
          forall l in currentSet
            if (l.location = i.mem_loc)
              newRecord ← gen (name of var, i.dest, i.address, undefined)
              currentSet ← currentSet ∪ { newRecord }
            fi
          endfor
        fi
      else if (i is a memory write)
        forall l in currentSet
          if (l.location = i.source)
            newRecord ← gen(l.varname, i.mem_loc, i.address, undefined)
            currentSet ← currentSet ∪ { newRecord }
          fi
        endfor
      fi

      i ← i.next_instruction
    } while (i ≠ ∅)
```

References

1. A. Adl-Tabatabai and T. Gross, "Evicted Variables and the Interaction of Global Register Allocation and Symbolic Debugging", *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, January 1993, pp. 371-383
2. A. Aho, R. Sethi, and J. Ullman, "Compilers Principles, Techniques, and Tools", Addison-Wesley Publishing Company, 1986
3. D. Coutant, S. Meloy, and M. Ruscetta, "DOC: A Practical Approach to Source-Level Debugging of Globally Optimized Code", In *Proceedings of the 1988 PLDI Conference*, 1988
4. J. Silverstein, ed., "DWARF Debugging Information Format", Proposed Standard, UNIX International Programming Languages Special Interest Group, July 1993
5. C. Tice and S. L. Graham, "Key Instructions: Solving the Code Location Problem for Optimized Code", Research Report 164, Compaq Systems Research Center, Palo Alto, CA, Sept. 2000
6. C. Tice, "Non-Transparent Debugging of Optimized Code", Ph.D. Dissertation, Technical Report UCB//CSD-99-1077, University of California, Berkeley, Oct. 1999.
7. P. Zellweger, "High Level Debugging of Optimized Code", Ph.D. Dissertation, University of California, Berkeley, Xerox PARC TR CSL-84-5, May 1984.