September 15, 2000

SRC Research Report

165

A Practical, Robust Method for Generating Variable Range Tables

Caroline Tice
and
Prof. Susan L. Graham

COMPAQ

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

http://www.research.compaq.com/SRC/

Compaq Systems Research Center

SRC's charter is to advance the state of the art in computer systems by doing basic and applied research in support of our company's business objectives. Our interests and projects span scalable systems (including hardware, networking, distributed systems, and programming-language technology), the Internet (including the Web, e-commerce, and information retrieval), and human/computer interaction (including user-interface technology, computer-based appliances, and mobile computing). SRC was established in 1984 by Digital Equipment Corporation.

We test the value of our ideas by building hardware and software prototypes and assessing their utility in realistic settings. Interesting systems are too complex to be evaluated solely in the abstract; practical use enables us to investigate their properties in depth. This experience is useful in the short term in refining our designs and invaluable in the long term in advancing our knowledge. Most of the major advances in information systems have come through this approach, including personal computing, distributed systems, and the Internet.

We also perform complementary work of a more mathematical character. Some of that lies in established fields of theoretical computer science, such as the analysis of algorithms, computer-aided geometric design, security and cryptography, and formal specification and verification. Other work explores new ground motivated by problems that arise in our systems research.

We are strongly committed to communicating our results; exposing and testing our ideas in the research and development communities leads to improved understanding. Our research report series supplements publication in professional journals and conferences, while our technical note series allows timely dissemination of recent research findings. We seek users for our prototype systems among those with whom we have common interests, and we encourage collaboration with university researchers.

A Practical, Robust Method for Generating Variable Range Tables

Caroline Tice and Prof. Susan L. Graham

September 15, 2000

Prof. Susan L. Graham works in the Electrical Engineering and Computer Science Department at the University of California, Berkeley. She can be reached by email at 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* [3, 7]. 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* [3]. 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.

Contents

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 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 where variable values reside during various portions of the 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 requests to see the value of a variable during the execution of the program.

There is a nice set of relationships that exists between source programs and their corresponding *unoptimized* binary programs that makes the task of collecting and generating symbol table information fairly straightforward. For example, whenever the value of a source variable is updated, the new value is always written to the variable's single home location in memory. Compiler optimizations, however, destroy these relationships, making the task of determining a variable's current location much harder. Once a program has been optimized, 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 portions of the computation where the variable is not live, not being anywhere at all. Determining where to look for a variable's value is known as the *data location problem*.

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. We are assuming a single thread of control in this paper.

The concept of a variable range table is not new; it has been around 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. This means that 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 directly 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 source for the optimizations. This makes it simpler to find and fix errors in the location-collection code. Also the code is more efficient than previous approaches, because the data is collected only once, not collected and then updated continuously as optimizations are performed.

The rest of this paper is organized as follows: In Section 2 we give an overview of our approach and explain the role of key instructions. Section 3 presents the important algorithmic details for collecting the data and assembling it into the variable range table. We briefly describe our implementation experiences in Section 4, and in Section 5 we present our conclusions.

2 Using the Optimized Binary

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, we can obtain completely accurate and correct information as to the locations of all the program variables, only performing the analyisis *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.

2.1 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 source (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 (in those cases where code has been duplicated, a single atom may have multiple corresponding key instructions). 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, out of the set of instructions generated from the atom, the key instruction is the first instruction encountered during program execution that causes a source-level-visible state change to the program. By definition every atom must have such an instruction.

Because we are concerned here with assignments to source variables, we will 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.[1] Therefore the key instruction for an assignment atom is the instruction that writes the right-hand side of the assignment to the variable's location in memory; or, if the write to memory has been optimized away, it is the final instruction that calculates the right-hand side and leaves it in a register. In those cases where 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.

[1]This discussion assumes there are no constructs in the source, such as a swap statement, that simultaneously update multiple source variables.

2.2 Identifying key instructions

We are assuming that the compiler keeps accurate information throughout the compilation process as to the original source position (file, line, and column position) from which every piece of internal representation was generated. Thus in the final phase, every instruction has associated with it information as to the source position(s) from which it was originally generated. The compiler has to maintain at least this basic information to make source-level debugging of the optimized target 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 out the set of instructions generated from that atom.[2] Key instructions for assignment atoms, however, are much harder to identify. In fact they cannot be identified at all, if one only 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 sort out 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 process, when it first parses the program and generates the appropriate internal representation. Therefore it makes sense to take advantage of the compiler and have the front end tag the piece of internal representation that will eventually become the key instruction(s) for an assignment atom. These tags can then be propagated through the compiler, and when the instructions are generated, the key instructions for assignment atoms are already tagged as such.

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

We claimed earlier that our approach to collecting variable location data is almost entirely independent of the optimization implementations. It is not completely independent because the key instruction tags do have to be propagated through the various optimization phases. We feel this is acceptable for two reasons. First it is far simpler and requires far less bookkeeping to propagate the one-bit tag indicating that a piece of internal representation corresponds to an assignment key instruction, than it does to track and record information about variable names and current variable storage locations. Second, and more importantly, we are assuming 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 taking advantage of an existing mechanism, rather than requiring a new, separate implementation.

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].

3 Algorithmic Details

Our algorithm for collecting the variable location information has two phases.

3.1 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 first phase collects that information for us. It 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 representation of the program. Every time an assignment is encountered, it is checked to see if the assignment is to a source-level variable.[3] 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.

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

3.2 Performing the dataflow analysis

The second phase, 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". 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 passed on during 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. Our dataflow algorithm can be found in Appendix A. Appendix B shows our algorithm for examining instructions within basic blocks.

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 table collected during the first phase. 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 records, 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 addresses the compiler assigned to any 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.)

The actions outlined above do not take pointer variables into consideration. Pointers require a little more work, e.g., modifying the code for memory load and store instructions to check if the operand being used to obtain the memory location corresponds to a live location for a source variable (dereferencing a variable in the live location tuple set); and modifying the code for mathematical operator instructions to correctly take into consideration operands that correspond to the locations in the set of live variable locations (i.e. using a variable in the live location tuple set as part of an address calculation for a pointer access.) The actual details can become somewhat complicated, but the basic ideas are straightforward.

3.3 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.

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. Any location tuples for a given variable and location that have consecutive ranges need to be combined. For example, the location tuples ⟨x, $r4, 1, 7⟩ (translated as "variable `x` is in register 4 from address one through address seven") and ⟨x , $r4, 7, 10⟩ ("variable `x` is in register 4 from address seven through address ten") can be combined into ⟨x, $r4, 1, 10⟩ ("variable `x` is in register 4 from address one through address ten").

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. 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. 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.[4] Thus an additional benefit of this approach is that we can automatically determine not only the correct locations of variables, but also the

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

address ranges for which variables are uninitialized or evicted.

4 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 variables 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.

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 1. 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`.[5] 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.

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.

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

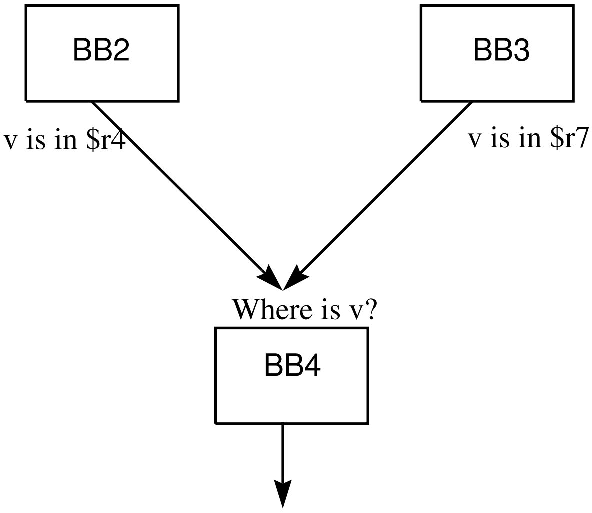

Figure 1: Inconsistency problem during dataflow analysis

This choice was reasonable because the problem in Figure 1 typically arises only when v is dead on entry to BB4. Otherwise the compiler would require definitive information as to where it could find the variable.[6] 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 to illustrate this, 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

[6]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.

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.

5 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 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 code for the optimizations; 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 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 dataflow analysis between basic blocks

Let N be the set of nodes in the graph, where each node corresponds to a basic block.

```
/* Initialize data in nodes; n.visited is the number of times this algorithm has
processed a node */

forall n ∈ N do
   n.visited ← 0
   n.locations ← Ø
od

/* Initialize variables; loopCnt counts the number of times the outer loop of this
algorithm iterates; topsOfLoops is a set containing the basic blocks in the graph
that correspond to tops of loops. */

EntryBB.locations ← initialSet
loopCnt ← −1
topsOfLoops ← Ø

/* Outer loop: This loop is necessary because it is impossible, in the presence of
loops, to process all the nodes in the graph such that all of a node's predecessors
are processed before the node itself. If the graph contains no loops, this outer
loop will iterate once; otherwise it will iterate until no changes occur in the
location set for any basic block. */

do {
   workset ← topsOfLoops ∪ { EntryBB }
   topsOfLoops ← Ø
   loopCnt ← loopCnt + 1
   changes ← false

   /* Inner loop: Process all nodes in the graph once, in an order such that, except
   for the tops of loops, no node is processed until all of its predecessor nodes
   are processed. */

   do {
      currentBB ← (B s.t. (B ∈ workset) ∧ (∀ C ∈ predecessor(B), C.visited > 0))
      if currentBB = Null
         currentBB ← an element in workset /* All nodes in worklist must be tops of loops */
         topsOfLoops ← topsOfLoops ∪ { currentBB }
      fi
      workset ← workset \ { currentBB }
```

```
    /* Initial set is union of all predecessor sets */

    currentSet ← currentBB.locations ∪ {⋃_{p∈predecessor(currentBB)} p.locations}

    /* Final set is initial set minus kill set union gen set; Appendix B shows
    algorithm for generating kill & gen sets */

    currentSet ← (currentSet \ Kill(currentBB)) ∪ Gen(currentBB)
    if (currentBB.locations ≠ currentSet)
      changes ← true
    currentBB.locations ← currentSet
    currentBB.visited ← currentBB.visited + 1

    /* Add successors to workset only if their visit count is less than the loop count;
    this prevents loop bottoms adding loop tops again (infinitely). */

    workset ← workset ∪ { B | (B ∈ successor(currentBB)) ∧ (B.visited ≤ loopCnt) }
  } while workset ≠ Ø
} while changes = true
```

B Algorithm for generating, passing, and killing location tuples within a basic block

```
currentSet ← ⋃_{p∈predecessor(currentBB)} p.locations
i ← 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 ← name_lookup(i.source_position)
    forall l in currentSet
      if (l.varname = varname)
        kill(l)
      fi
    endfor
    newRecord ← gen (varname, i.dest, i.address, undefined)
```

```
    currentSet ← currentSet ∪ { newRecord }
  else if (i is a register copy)
    forall l in currentSet
      if (l.location = i.source)
        newRecord ← gen (l.varname, i.dest, i.address, undefined)
        currentSet ← currentSet ∪ { 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 ≠ Ø)
```

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", Tech. Report 164, Compaq Systems Research Center, Palo Alto, CA, Sept. 2000

[6] C. Tice, "Non-Transparent Debugging of Optimized Code", Ph.D. Dissertation, Tech. 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.