A New Data-Location Tracking Scheme for the Recovery of Expected Variable Values

Le-Chun Wu* Wen-mei W. Hwu†

*Department of Computer Science
†Department of Electrical and Computer Engineering and
The Coordinated Science Laboratory
University of Illinois
Urbana, IL 61801
Email: {lcwu, hwu}@crhc.uiuc.edu

Technical Report IMPACT-98-07

September 1998

1 Introduction

The run-time location of a user variable may be altered by optimizations such as register allocation. The variable value may be in different places (constant, register, or memory) at different points of execution. Or it may not exist at all due to code deletion and register (and memory location) reuse. To allow the user to access the value of a variable, the debugger has to know what location holds the value of the variable at breakpoints.

Coutant et al. [1] proposed a data structure called *range* to communicate to the debugger the location information of variables in different ranges of the binary program. A variable has a set of range records which are calculated based on the live ranges of the variable at compile-time. For example, in Figure 1(a), there are three live ranges for variable a in the sample program.

The range records for variable a is shown in Figure 1(b). By comparing the address of an object code location with the *Low Address* and *High Address* of each range record, the debugger can decide where to get the variable value at this object code location. If the address is not in any one of the range records, which means the variable value is not available at this point, the debugger has to inform the user of this fact.

Range information calculated based on the live ranges of variables is considered conservative because the

Low Addr	High Addr	Location	Type
2004	2004	3	constant
2012	2016	r1	register
2028	2028	r5	register

Low Addr	High Addr	Location	Type
2004	2008	3	constant
2012	2024	r1	register
2028	2028	r5	register

Figure 1: (a) An example code segment (b) range records for variable a (c) extended range records for variable a

fact that a variable is not live doesn't mean its value is not available. For example, in the program shown in Figure 1, variable a is not live from address `2020` to `2024`, but its value is still in register `r1`. Adl-Tabatabai and Gross [2] have proposed a framework using data-flow analysis which can be used to extend the range of a value location to the point where the value is killed. The extended ranges of variable a for the example in Figure 1(a) is shown in Figure 1(c).

While the data-location information generated by the aforementioned techniques is sufficient for the debugger to determine if a variable value can be found in any place, it becomes insufficient (or even inaccurate) when the debugger attempts to recover the expected variable values at breakpoints, as illustrated in Figure 2. Figure 2(a) shows the original code of a sample program. Figure 2(b) shows the optimized code where instruction $I5$ (a definition of variable b) is moved up. The ranges for variable b constructed using Adl-Tabatabai's approach is depicted in Figure 2(c). Suppose a breakpoint is set at statement S whose anchor point is at $I4$. When the user requests for b's value at this breakpoint, the debugger in our scheme will compare the address of the anchor point $I4$ with the range records for b[1]. Using the range records shown in Figure 2(c), the debugger would think b's value is in register $r3$, while in fact the value of b at the breakpoint should come from the definition at $I1$ (in register $r1$) in order to provide expected behavior. Hence the range information we desire in order to provide the expected value of b should be the one depicted in Figure 2(d) where the first range is extended to cover instruction $I4$.

To solve the problem mentioned above and provide data-location information more fitting to our new debugging strategy, a new data-location tracking scheme is proposed. Our scheme keeps track of information about the definitions (assignments) of source variables during compilation and register allocation, and then calculates variable range information using the definition information preserved. The range information will be constructed in a way such that the original execution order relationship between the definitions of a

[1]Our debugging scheme does not map a source breakpoint to a single object location as most of the traditional debuggers do. Therefore we use the address of the anchor point of a source breakpoint to compare with the range records because anchor points preserve the original execution order and reaching conditions of the source statements.

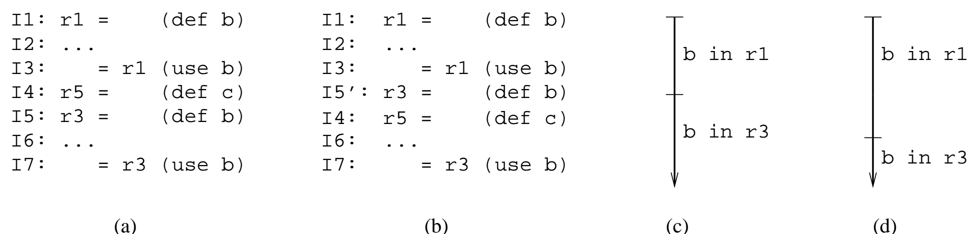

Figure 2: (a) Original code (b) Optimized code (c) range records for variable b using previous techniques (d) range records for variable b desired

variable and the breakpoints is reflected.

Recovering the expected values of non-current variables caused by code deletion is also handled by our data-location tracking scheme. Most of the time when an assignment of a variable is deleted, the value assigned to the variable can still be recovered using the values of other variables. Our scheme tracks where or how to obtain the value of a variable whenever an assignment of a variable is deleted. Besides a constant, a register, or a memory location, the location of a variable in our scheme can also be an arithmetic expression consisting of various storage locations. Although the concept of using the values of other variables to recover a deleted variable value has been mentioned in some of the previous work [3, 1], our scheme provides a more general and systematic approach which can handle almost all kinds of code deletion and recover the expected values of deleted variables whenever it is possible.

In the following subsections, we discuss how our scheme preserves the variable definition information and calculates the range information.

2 Variable definition information

For each source variable, the compiler keeps a set of definition records each of which corresponds to a source definition (assignment) of the variable. A definition record contains the following information:

Definition type The type of the definition includes *original*, *moved*, *equivalent*, and *deleted*.

Value expression The information about where or how to obtain the value of the variable. It may be a constant, a register, a memory location, or an arithmetic expression.

Actual definition points The instruction (machine location) that moves the source value of the variable to a storage location, or the instructions whose destinations constitute the value expression of the variable when the original definition instruction is deleted.

Effective definition points The instructions (machine locations) from which the source definition should take effect based on the semantics of the original program.

Source location of the definition This information includes the source location and the execution order of the definition (assignment).

Before any optimization is performed, for a source definition D of variable V, the instruction I that moves the source value of V to a storage location serves as both the actual and the effective definition points of D. The type of the definition is *original* and the value expression is the destination of I.

Since only code deletion and code movement will affect the values of source variables, we will discuss how the variable definition information is maintained in these two cases.

1. *Code deletion.*

   When an instruction I, which is an actual definition point of a definition D of variable V's, is deleted,

   (a) If the value of the destination of I, dst, can be obtained through the expression E which is available at I, and I is the only actual definition point of D which defines dst,

      i. dst in the value expression of D is replaced by E,

      ii. the instructions which define the operands of E replace I and become the actual definition points of D, and

      iii. the type of D becomes *equivalent* if it hasn't been so.

   (b) Otherwise, the type of D becomes *deleted* and there is no actual definition point and value expression for D.

   If the type of D is *original*, the anchor points of the source definition corresponding to D will become the effective definition points of D. This issue will be discussed later.

   Figure 3 shows a code example where instruction $I4$, a definition of variable y, is deleted because it is dead. We can see y's value can be recovered by the expression $r1+r2$ $(a+b)$. Therefore $r1+r2$ becomes the value expression of the deleted definition, instructions $I1$ and $I2$ become the actual definition points, and the type of the definition becomes *equivalent*. If instruction $I5$ is the anchor point of the source definition, $I5$ becomes the effective definition point.

2. *Code movement*

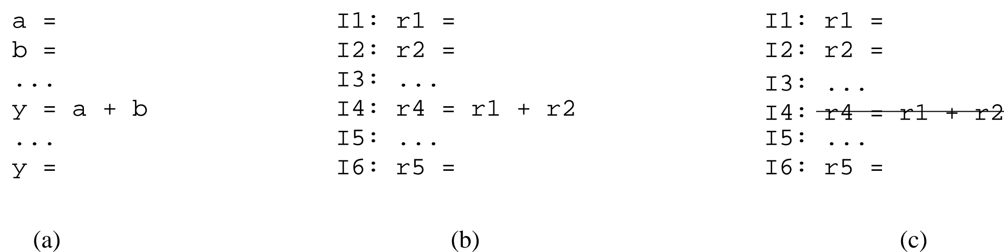

Figure 3: (a) Original source code (b) Original assembly code (c) Optimized assembly code

When an instruction I, which is an actual definition point of a definition D of variable V's, is moved to a place *control-equivalent* to its original location [2],

(a) I's new location replaces its old location as an actual definition point of D.

(b) If the type of D is *original*, it is changed to *moved*.

Also if the type of D is *original*, the anchor points of the source definition corresponding to D will become the effective definition points of D.

Refer back to Figure 2. After instruction $I5$, which is a definition of variable b, is moved, the new location of $I5$ becomes the actual definition point of the definition, and the definition type becomes *moved*. If $I6$ is the anchor point of the definition, it becomes the effective definition point.

When an instruction I is moved to a place which is not *control-equivalent* to its original location as shown in Figure 4, if the type of D is *equivalent*, we handle the movement as we did for the simple code movement mentioned above. Even though the movement of I might render the value expression of D unavailable at some points, the compiler will calculate the availability of the value expression before the range information is calculated.

However, when D's type is either *original* or *moved*, how we handle it depends on the movement of the code which can be classified into four cases.

Case 1 I is moved up to a place which dominates but is not post-dominated by I's old location as shown in Figure 4(a). This case usually happens when speculative code motion is performed. We treat this case as if I is deleted, while the value of V defined by I can be found in the destination of I'. Therefore the type of D becomes *equivalent* and I' becomes the actual definition point of D. This way we can ensure the range records for V will be created (see Section 3) such that the

[2]Two machine locations are *control-equivalent* if one dominates and is post-dominated by the other.

value of V at breakpoint 1, 2, 3, and 5 will not be from I', while the expected value of V can be obtained at breakpoint 4 as long as $r4$ is not re-defined before reaching the breakpoint.

Case 2 I is moved up to a place which does not dominate but is post-dominated by I's old location. If this case happens due to partial redundancy elimination as depicted in Figure 4(b), we treat this case as if I is moved to two new places, J and I'. Therefore the type of D becomes *moved* and both J and I' become the actual definition points of D. This way we can ensure the range records created for V will not be affected by I' (see Section 3) and the expected value of V can be obtained at breakpoint 3 as long as the value of $r4$ defined by J and I' are both available at the breakpoint.

However, if I is moved up due to other reasons, to be conservative, we treat the case as if I is being deleted and the value of V defined by I cannot be recovered. That is, the type of D becomes *deleted* and there is no actual definition point and value expression for D.

Case 3 I is moved down to a place which does not post-dominate but is dominated by I's old location as shown in Figure 4(c). This case usually happens when partial dead code elimination is performed. We treat this case as if I is deleted, while the value of V defined by I can be found in the destination of I'. Therefore the type of D becomes *equivalent* and I' becomes the actual definition point of D. This way we can ensure the range records for V will be created (see Section 3) such that the value of V defined by the definition(s) prior to I will not be seen by breakpoint 1, 2, 3, and 5, while the expected value of V can be obtained at breakpoint 4.

Case 4 I is moved down to a place which post-dominates but is not dominated by I's old location. If this case happens due to tail merging as depicted in Figure 4(d), we treat it as a simple code movement. Therefore the type of D becomes *moved* and I' becomes the actual definition point of D.

However, if I is moved down due to other reasons, to be conservative, we treat the case as if I is being deleted and the value of V defined by I cannot be recovered. That is, the type of D becomes *deleted* and there is no actual definition point and value expression for D.

After all the optimizations have been done, the compiler will check the definition records of each variable. If the type of a definition is not *original*, the anchor points of the source assignment statement which corresponds to the definition become the effective definition points of the definition. Since we use the address of the anchor point of a source breakpoint to compare with the range records, using anchor points of a moved or deleted source assignment as the effective definition points (which will later on become the start

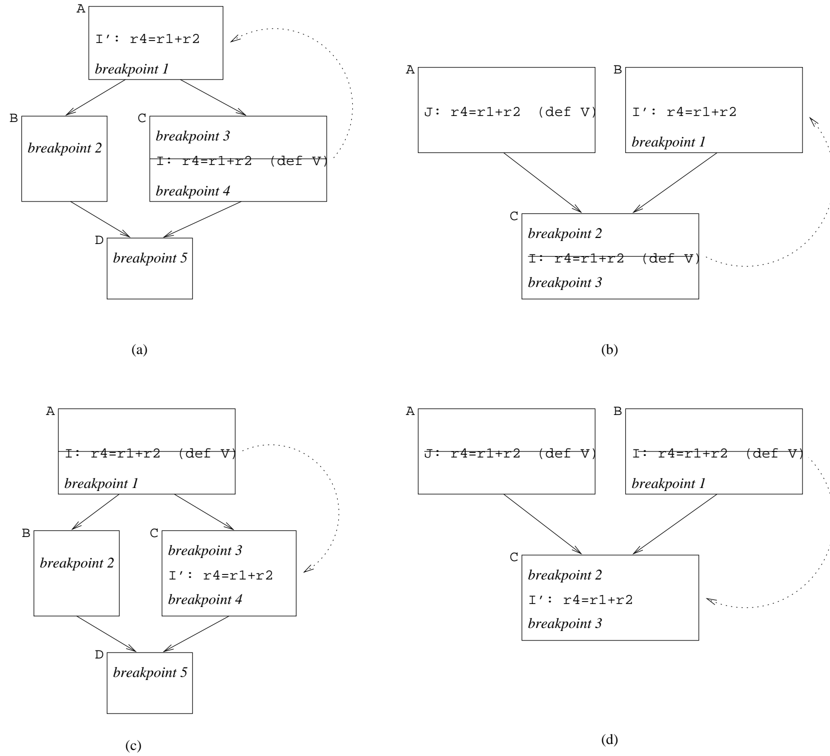

Figure 4: Code moved to a non-control-equivalent place (a) Case 1 (b) Case 2 (c) Case 3 (d) Case 4

points of range records) will ensure the correct coverage of the ranges with regard to each source breakpoint.

3 Range calculation

Besides low address (start address), high address (end address), and location information (which is extended to handle more complicated value expression), a range record in our scheme also includes information about source execution order of the source assignment corresponding to the range. The reason why the execution order information is required will be discussed later.

Range calculation is based on the variable definition information preserved during optimization. For a source variable V, range records are created for each of its definitions, D, only when

1. the type of D is not *deleted*, and

2. the destinations of D's actual definition points are all available at some points between D's effective definition points and the places where D is killed if the type of D is *equivalent*.

For example, in the optimized code shown in Figure 3(c), if either $r1$ or $r2$ is re-defined before instruction

$I5$ (which is the effective definition point of a definition of y) is reached, no range record will be created for this definition of y.

Range records created for D may start at the following locations:

1. The effective definition points of D if
   (a) the type of D is *original* or *moved*, or
   (b) the type of D is *equivalent* and all of D's actual definition points will be reached before its effective definition points without traversing back edges.
2. The earliest location where all the destinations of D's actual definition points are available if
   (a) the type of D is *equivalent*, and
   (b) some of D's actual definition points will be reached after its effective definition points without traversing back edges.

   For example, in Figure 4(c), suppose after optimization the effective definition point of V's definition is the instruction immediately following I in basic block A. Since the actual definition point of the definition, I', is reached after the effective definition point, the range record corresponding to the definition will start from the point immediately following I'.
3. The beginning of a basic block which can be reached by definition D if
   (a) D is available on all paths leading to the basic block, or
   (b) there is a definition of V's on every path leading to the basic block and all V's definitions which reaches the basic block have the same value expression.

   For example, in Figure 5(a), suppose $I1$ and $I2$ are two definitions of variable y which reaches basic block C. Since these two definitions have the same value expression, $r1$, a range record for variable y will be created starting from the beginning of basic block C. On the contrary, in Figure 5(b), suppose $I1$ and $I2$ are two definitions of variable y which reaches basic block C. Since their value expressions are different, no range record for variable y will start from the beginning of basic block C.

Range records for D may end at the following locations:

1. The effective definition points of other definitions of V (including those definitions with type *deleted*).

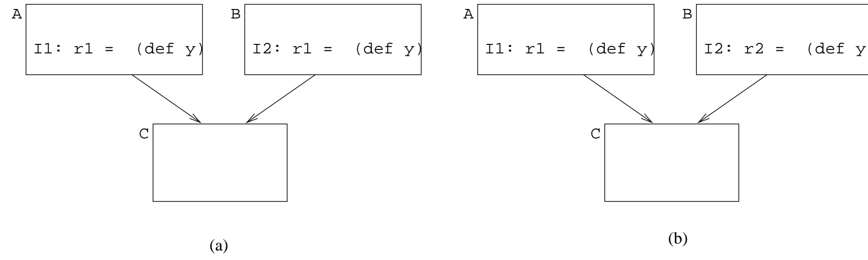

Figure 5: (a) Definitions with the same value expression (b) Definitions with different value expressions

```
I1: r1 =    (def a)
I2:    = r1
I3: r1 =    (def b)
I4:    = r1
I5: r3 =    (def c)
I6: r1 =    (def b)
I7: ...
I8:    = r1
```

(a)

```
I1:  r1 =    (def a)
I2:     = r1
I3:  r1 =    (def b)
I4:     = r1
I6': r1 =    (actual def b)
I5:  r3 =    (def c)
I7:  ...     (effective def b)
I8:     = r1
```

(b)

Figure 6: (a) Original code (b) Optimized code

2. The location where any of the source operands in D's value expression is re-defined (killed), unless the instruction which re-defines the operand is an actual definition point of a moved definition of V.

   For example, Figure 6(b) shows an optimized code where instruction $I6$ is moved up after optimization. The range record for variable a will end at $I3$ because $r1$, which is the value expression of a's definition, is re-defined by $I3$. However, the range record for the first definition of b does not end at $I6'$ because $I6'$ is the actual definition point of a moved definition of b. The reason is because if the range record for the first definition of b ends at $I6'$, since the range record for the second definition of b won't start until $I7$ (the effective definition point of the second definition of b), instruction $I5$ will not be covered by any range record of b. However, we would like $I5$ to be covered by the range record of the first definition of b because our selective execution would provide a correct value of $r1$ even though $r1$ is modified prematurely by $I6'$.

3. The end of a basic block.

The range records constructed for D are assigned D's value expression and source execution order information.

Note that the reason why a range record for a definition D of variable V starts from instruction I, an effective definition point of D, instead of the instruction immediately following I is because I might serve as an anchor point of multiple source statements. Some of these statements might have smaller execution

order than definition D and therefore we should exclude I from the range record for D. Some might have larger execution order than D and I should be included in the range record. Without knowing where the breakpoints will be set in advance, we will need to always include I in the range record. At debug-time, when the address of an anchor point of a source breakpoint matches the start address of a range record, the debugger will use the source execution order information of the range record to decide if this range should cover the breakpoint or not.

Based on the the concept and intuition we described above and the algorithm Adl-Tabatabai proposed to determine the residency of variables [4], a data-flow algorithm to calculate the range information of variables is presented in Appendix A. Note that the range records for a variable V calculated by our algorithm is automatically *coalesced* so that the adjacent ranges of V which have the same value expression are merged into a bigger range to reduce the number of the range records (and therefore the size of the debugging information).

References

[1] D. Coutant, S. Meloy, and M. Ruscetta, "DOC: A practical approach to source-level debugging of globally optimized code," in *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 125–134, June 1988.

[2] A. Adl-Tabatabai and T. Gross, "Evicted variables and the interaction of global register allocation and symbolic debugging," in *Conference Record of the 20th Annual ACM Symposium on Principles of Programming Languages*, pp. 371–383, January 1993.

[3] J. Hennessy, "Symbolic debugging of optimized code," *ACM Transactions on Programming Languages and Systems*, vol. 4, pp. 323–344, July 1982.

[4] A. Adl-Tabatabai, *Source-Level Debugging of Globally Optimized Code.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, 1996.

A Data-flow algorithm for range calculation

To facilitate the data-flow algorithm, instructions will be first annotated with variable definition attributes (each of which include a variable name and a value expression) in the following way:

For each definition, D, of variable V,

- if I is an effective definition point of D, I will be annotated with a definition attribute of V with value expression E. However, the value expression in the attribute will be set to null when

   1. the type of D is *deleted*, or

2. the type of D is *equivalent* and some of D's actual definition points will be reached after its effective definition points without traversing back edges, or
3. the type of D is *equivalent* and the destinations of D's actual definition points are not all available at any point between D's effective definition points and the places where D is killed.

Or,

- if I is the earliest location where all the destinations of D's actual definition points are available when
  1. the type of D is *equivalent*, and
  2. some of D's actual definition points will be reached after its effective definition points without traversing back edges, and
  3. the destinations of D's actual definition points are all available at some points between D's effective definition points and the places where D is killed,

I will be annotated with a definition attribute of V with value expression E.

The data-flow algorithm is operated on *variable definition pair* $< V, E >$ where V is a source variable and E is a value expression. That $< V, E >$ is available at a point of the program means V's value is obtained through E when the execution stops at that point. Therefore we have

$AvailVarExprIn[I] = \{< V, E > \mid$ the value of V is obtained through E at the point before executing $I\}$

$AvailVarExprOut[I] = \{< V, E > \mid$ the value of V is obtained through E at the point after executing $I\}$

Also we define $ValueExpression[V]$ to be the set of value expressions which V's value can be obtained from at different points of the program.

The value of a variable V is obtained through a value expression E at the point before executing I only when V's value is obtained through E after all of I's predecessor. Therefore

$$AvailVarExprIn[I] = \bigcap_{J\ is\ a\ predecessor\ of\ I} AvailVarExprOut[J]$$

We define $AvailVarExprGen[I]$ to be the set of variable definition pairs made available by instruction I and $AvailVarExprKill[I]$ to be the set of variable definition pairs killed by I.

$$AvailVarExprOut[I] = AvailVarExprGen[I] \cup (AvailVarExprIn[I] - AvailVarExprKill[I])$$

The $AvailVarExprGen$ set and $AvailVarExprKill$ set for instruction I are defined as follows:

- If I is annotated with a variable definition attribute of variable V with value expression E,
  - $< V, E > \in AvailVarExprGen[I]$ if E is not null
  - $\forall E' \in ValueExpression(V), < V, E' > \in AvailVarExprKill(I)$
- If I has a destination L, for each value expression E of every variable V which has L as one of its operands (unless I is an actual definition point of a moved definition of V),
  - $< V, L > \in AvailVarExprKill[I]$

After the data-flow analysis is done, a range record for variable V with value expression E starts from instruction I if $< V, E >$ is first seen in $AvailVarExprOut[I]$, and ends at instruction J if $< V, E >$ is not in $AvailVarExprOut[J]$ any more.