# Finding Basic Block and Variable Correspondence

Iman Narasamdya and Andrei Voronkov

University of Manchester
{in, voronkov}@cs.man.ac.uk

**Abstract.** Having in mind the ultimate goal of translation validation for optimizing compilers, we propose a new algorithm for solving the problem of finding basic block and variable correspondence between two (low-level) programs generated by a compiler from the same source using different optimizations. The essence of our technique is interpretation of the two programs on random inputs and comparing the histories of value changes for variables. We describe an architecture of a system for finding basic block and variable correspondence and provide experimental evidence of its usefulness.

## 1 Introduction

Verifying the optimizing phase of a compiler has become crucial as developers have been relying on this phase to produce high performance code. However, proving the correctness of the optimizing phase is infeasible due to its size, its sophisticated algorithms and data structures, as well as ongoing evolution and modification. *Translation validation* [7] is an alternative but feasible approach to compiler correctness, which can be applied to the optimizing phase [6,10,8]. The idea of translation validation is as follows: instead of proving the correctness of the optimizing phase for *every* possible program, prove *for a single program* that the program and its optimized version are semantically equivalent.

In this paper we take the following view of translation validation. We have two programs $P$ and $P'$, and each of them is a result of compiling the same source program, but unlike $P$, the compilation of $P'$ involves the optimizing phase. Both programs are written in the same intermediate language. We call the program $P$ the *original program*, and the program $P'$ the *optimized program*.

Knowing that a variable in $P$ corresponds to a variable in $P'$ gives us a valuable information that can be used to prove the equivalence of $P$ and $P'$ automatically. Intuitively, a variable $x_1$ in the original programs corresponds to a variable $x_2$ in the optimized program if they have the same values at some control blocks for all possible runs of the two programs on the same input values. We shall formulate the right notion of correspondence in a formal way later.

In this paper *block* is a sequence of instructions that is always entered at the beginning and exited at the end. With this definition, we consider a *program point* as a block consisting of one instruction. A block is called a *basic block* if the sequence of instructions is maximal.

C. Hankin and I. Siveroni (Eds.): SAS 2005, LNCS 3672, pp. 251–267, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Consider the following simple programs:

```
P :                          P′ :
w1 := n;                     x1 := n;
while w1 > 0 do              while x1 > 0 do
  w2 := w1 − 1;                x1 := x1 − 2;
  w1 := w2 − 1;                call f(x1)
  call f(w1)                 od
od
```

In these programs $n$ is an argument and $w_1, w_2, x_1$ are local variables. Suppose that the function $f$ does not have any side effect. If we can establish that the variable $w_1$ in $P$ corresponds to the variable $x_1$ in $P'$, then we can verify that the two programs are equivalent without generating loop invariants. Indeed, using this information we can check that the two programs will perform the same sequence of calls of $f(\ldots)$ by the following kind of reasoning. First, the values of $w_1$ and $x_1$ coincide at the entries of the two loops. Second, if they coincide at some iteration of the loops, they also coincide at the next iteration. Third, if they coincide at some iteration of the loops, the function $f$ will be called with the same arguments in both programs. Finally, if the loop exit condition $w_1 \leq 0$ is satisfied in $P$, the loop exit condition $x_1 \leq 0$ is also satisfied in $P'$.

Note that this reasoning also *proves* that $w_1$ in $P$ and $x_1$ in $P'$ correspond to each other. However, before performing this reasoning we must in some way *guess* that $w_1$ in the original program corresponds to $x_1$ in the optimized program. Moreover, we also have to establish some correspondence between control blocks in the two programs: blocks in which the corresponding variables always have the same values. This paper deals with "guessing" a basic block and variable correspondence. We do not yet consider the VC generation from a given correspondence or proving the VCs.

By only knowing that one program is an optimized version of the other, it is not trivial to construct automatically a basic block and variable correspondence. Optimizations can change the structure of a program, for instance, while-do loops are transformed to do-while loops to be able to move loop invariants. Optimizations might also include eliminating existing branches and introducing new branches to the program. In this paper we do not try to address any *reordering transformation*, that is any transformation that changes the order of execution of code, without adding or deleting any executions of any statement [5].

This paper proposes a new technique in constructing a basic block and variable correspondence between $P$ and $P'$. The idea of this new technique is to execute $P$ and $P'$ separately with the same initial store, also called a *memory state* here. The values stored in the memory are generated randomly upon demand. For example, when a program accesses an uninitialized memory location, we can create a new piece of memory and fill it with a random value. Furthermore, while executing the programs, the values assigned to each variable and the blocks in which these assignments occur are recorded. If the sequences of *value changes* of two variables are the same, then the variables probably correspond to each other, and the block in which the changes occur might also correspond to each other.

The problem of finding a basic block and variable correspondence between $P$ and $P'$ is a hard problem. No single technique is able to cover all possible optimizations

applied to the source program. The emphasis of our work here is to develop a *cheap* technique that could help to find a basic block and variable correspondence. This correspondence in turn can help us generate a verification condition which is sufficient to prove the equivalence of $P$ and $P'$. Our technique is considerably cheap for the following reasons. First, our technique amounts to building an interpreter to perform program executions. As the language in which $P$ and $P'$ are written is usually simple, the interpreter is easy to develop. Moreover, it does not take a sophisticated algorithm to determine the sameness of value changes between two records. Another advantage of our new technique is that it needs only the code of the original and the optimized programs but no further information from the optimizing phase. Therefore, it can be applied to verify the optimizing phase of different compilers without instrumenting them any further.

The remainder of this paper is organized as follows. Section 2 gives an overview of some recent existing techniques in constructing basic block and variable correspondences. Section 3 states formally the problem of finding basic block and variable correspondence. In Section 4 the idea of the new technique is discussed. Afterwards, in Section 5, we discuss the syntax and semantics of an intermediate language used throughout this paper . Section 6 discusses a *randomized interpreter* used to evaluate programs written in the intermediate language. Finally, section 7 describes some experimental results. An extended version of this paper is available at `http://www.cs.man.ac.uk/~voronkov/sas_fullpaper.ps`.

## 2 Related Work

One technique related to translation validation is *Necula's technique* [6]. In this technique, each of the original and the optimized programs is firstly evaluated symbolically into a series of mutually recursive function definitions. A basic block and variable correspondence is inferred by a scanning algorithm that traverses the function definitions. For example, when the scanning algorithm visits a branch condition $e$ in the original program, it determines whether $e$ is eliminated due to the optimizations. If it is eliminated, then the information collected is either $e = 0$ or $\neg e = 0$, depending on which branch of $e$ is preserved in the optimized program. If $e$ is not eliminated, then it corresponds to another branch condition $e'$ in the optimized program. The information collected is either $e = e'$ or $e = \neg e'$, depending on the correspondence of $e$'s and $e'$'s branches. This shows that, besides symbolic evaluation, Necula's technique has to solve some equalities to determine which branches are eliminated and also to determine the correspondence between branches in the two programs. Moreover, to find a basic block correspondence Necula's technique uses some heuristics which are specific to the *GNU C compiler*. This limits the applicability of Necula's technique to verifying other compilers.

Another translation validation technique is *VOC* [11]. We overview VOC for structure preserving transformations only. Such transformations admit a mapping between some program points in $P$ and $P'$. In VOC a basic block and variable correspondence is represented by a mapping from some blocks in $P'$ to some blocks in $P$, and also by a data abstraction. The domain and range of the block mapping form sets of *control*

*blocks*. VOC chooses the first block of each loop body as a control block. The data abstraction is constructed as follows. For each block $B_i$ in $P'$, and for every path from block $B_j$ leading to $B_i$, a set of equalities $v = V$ is computed, where $v$ and $V$ are variables in $P$ and $P'$ respectively. The equalities are implied by invariants reaching $B_j$, transition system representing the path from $B_j$ to $B_i$ and its counterpart in $P$, and the current constructed data abstraction. This requires the implementation of VOC to use a prover to generate a data abstraction. Moreover, an implementation of VOC for *Intel's ORC compiler*, *VOC-64*, tries the variable equalities for every pair of variables except for the temporaries introduced by the compiler. This trial is performed by scanning the symbol table produced by the compiler [2]. However, not every compiler provides the symbol table as a result of compilation, thus this limits the applicability of VOC-64.

A quite recent translation validation technique is *Rival's technique* [9]. The technique provides a unifying framework for the certification of compilation and of compiled programs. Similarly to Necula's technique, the framework is based on a symbolic representation of the semantics of the programs. Rival's technique extracts basic block and variable correspondence from the standard debugging information if no optimizations are applied. However, when some optimizations are involved in the compilation, the optimizing phase has to be instrumented further to debug the optimized code and generate the correspondence between the original and the optimized programs. One technique to automatically generate such a correspondence is due to *Jaramillo* et. al [4]. In this technique, the optimized programs initially starts as an identical copy of the original one, so that the mapping starts as an identity. As each transformation is applied, the mapping is changed to reflect the effects of the transformation. Thus, in this technique, one needs to know what and in which order the transformations are applied by the optimizing phase.

## 3 Basic Block and Variable Correspondence

In this section we formalize the problem we are trying to solve. We will only be dealing with *programs* divided into blocks. A concrete notion of program will be defined later in Section 5. We assume that every program defines a transition relation with two kinds of transition: (i) transitions $(\beta_1, \sigma_1) \to (\beta_2, \sigma_2)$; (ii) transitions $(\beta_1, \sigma_1) \to \sigma_2$, where $\beta_1, \beta_2$ are blocks and $\sigma_1, \sigma_2$ are stores. Intuitively, the second kind of transition brings the program to a terminal state. The *run* of such a program is either an infinite sequence $(\beta_0, \sigma_0), (\beta_1, \sigma_1), \ldots$ or a finite sequence $(\beta_0, \sigma_0), (\beta_1, \sigma_1), \ldots, (\beta_n, \sigma_n), \sigma_{n+1}$. Here $\beta_0, \beta_1, \ldots$ is the sequence of blocks visited in this run and $\sigma_i$ is the store when the run reaches $\beta_i$.

Let $\bar{b}$ be a sequence of distinct blocks in $P$ and $R$ be a run. Denote by $R|_{\bar{b}}$ the subsequence of $R$ consisting of the blocks occurring in $\bar{b}$.

Let $P$ and $P'$ be two programs, $\bar{b} = b_1, \ldots, b_k$ be a sequence of distinct blocks in $P$ and $\bar{b}' = b'_1, \ldots, b'_k$ be a sequence of distinct blocks in $P'$ of the same length. Let also $\bar{x} = x_1, \ldots, x_m$ be a sequence of variables[1] in $P$ and $\bar{x}' = x'_1, \ldots, x'_m$ be a sequence of variables in $P'$, also of the same length. In the sequel we will refer to $\bar{b}$ and $\bar{b}'$ as *control blocks* and to $\bar{x}$ and $\bar{x}'$ as *control variables*.

[1] For simplicity, we consider variables as memory locations.

We say that there is a *block and variable correspondence* between $(\bar{b}; \bar{x})$ and $(\bar{b}'; \bar{x}')$ if, for every pair of runs $R = (\beta_0, \sigma_0), (\beta_1, \sigma_1), \ldots$ and $R' = (\beta'_0, \sigma'_0), (\beta'_1, \sigma'_1), \ldots$ of the programs $P$ and $P'$, respectively, on the same inputs and the same initial store, (that is, $\beta_0 = \beta'_0$ and $\sigma_0 = \sigma'_0$) the following conditions hold. Let

$$R|_{\bar{b}} = (\beta_{i_0}, \sigma_{i_0}), (\beta_{i_1}, \sigma_{i_1}), \ldots \qquad R'|_{\bar{b}'} = (\beta'_{j_0}, \sigma'_{j_0}), (\beta'_{j_1}, \sigma'_{j_1}), \ldots$$

Then $R|_{\bar{b}}$ and $R'|_{\bar{b}'}$ have the same length and for every non-negative integer $n$ the following conditions hold:

1. $\beta_{i_n} = b_\ell$ if and only if $\beta'_{j_n} = b'_\ell$, for all $\ell$;
2. $\sigma_{i_n}(x_1) = \sigma'_{j_n}(x_1), \ldots, \sigma_{i_n}(x_m) = \sigma'_{j_n}(x_m)$;
3. $\sigma_{i_{n+1}}(x_1) = \sigma'_{j_{n+1}}(x_1), \ldots, \sigma_{i_n+1}(x_m) = \sigma'_{j_n+1}(x_m)$.

That is, in $R$ and $R'$ the control blocks are visited in the same order, and the values of the control variables at the entries and exits of the visited control blocks are the same.

Our main goal is to find, in a fully automatic way, a correspondence between program points and variables of $P$ and $P'$. Note that we always have a correspondence when $\bar{b}$ is an empty sequence. Likewise, we always have a correspondence when $\bar{x}$ is an empty sequence. As a consequence, there is no largest correspondence. However, we are interested in correspondences in which $\bar{b}$ is "as large as possible", and similarly for $\bar{x}$.

The definition of basic block and variable correspondence above allows us to trade variable correspondence for block correspondence and vice versa. Consider the following programs with $n$ as their arguments:

```
Program P:                      Program P':
b0 : if n ≤ 0 then              b0' : if n > 0 then
        b2 : x1 := n                    b2' : x2' := 1
             x2 := 0                  else
     else                               b3' : x2' := 0
        b3 : x1 := n            b4' : x1' := n
             x2 := 1                  x3' := x2'
b4 : x1 := n
     x3 := x2
```

The program $P'$ can be obtained by applying dead code elimination to $P$. If we can establish that $x_1$, $x_2$, and $x_3$ in $P$ correspond to their primed counterparts in $P'$, we could only construct a block correspondence between $b_0$ and $b'_0$, and also between $b_4$ and $b'_4$. The block $b_2$ does not correspond to the block $b'_3$ since the values of $x_1$ and $x'_1$ after executing these blocks are different. When we sacrifice the correspondence between $x_1$ and $x'_1$, we obtain a larger block correspondence, that is between $b_2$ and $b'_3$, and also between $b_3$ and $b'_2$. The resulting block correspondence is crucial if we have to establish a branch correspondence.

We can introduce variations on the basic block and variable correspondence problem. For example, if a variable is initialized inside a block, we can restrict the definition to its value at the block exit only. We can change the definition so that a single block in one of the programs will correspond to several blocks in another program. This will help us to cope with such optimizations as *loop invariant hoisting*. Likewise, we can

consider the *single static assignment* (or *SSA*) [1] form of programs in which a variable may change its value only inside a single basic block. The technique we discuss in this paper is equally applicable to these modifications.

## 4 Random Interpretation

In this section we introduce the technique of *random interpretation* that allows one to address the block and variable correspondence problem. The idea of the technique is to evaluate both the original program and its optimized version separately with the same initial randomly generated memory state. A memory state can be thought of as a function mapping memory locations to the content of the memory at these locations. While evaluating each program, we record the values assigned to each variable and the program points at which the assignments occur. This record forms a *history* of values assigned to a variable. Let us define the notion of history formally.

As usual, we say that a block $b$ *defines* a variable $x$ if $b$ contains an assignment to $x$. Consider a run $R$ of a program $P$ and let $x$ be a variable occurring in $P$. Let $\bar{b}$ be the set of all blocks in $P$ defining $x$ and $R|_{\bar{b}} = (\beta_0, \sigma_0), (\beta_1, \sigma_1), \ldots$. Then $\beta_0, \beta_1, \ldots$ are the only blocks in this run which may change the value of $x$. We call the *history of $x$ in $R$* the sequence of pairs

$$(v_0, \beta_0), (v_1, \beta_1), \ldots \tag{1}$$

where each $v_i$ is the value of $x$ *at the exit* of $\beta_i$. Now, given the history $h_x$ of $x$ in $R$ of the form (1) we call the *value change sequence of $x$ in $R$* any subsequence of (1)

$$(v_{j_0}, \beta_{j_0}), (v_{j_1}, \beta_{j_1}), \ldots$$

that can be obtained from (1) by repeated applications of the following transformation: if the sequence contains two consecutive pairs with the same value:

$$\ldots, (v, \beta_i), (v, \beta_{i+1}), \ldots,$$

then remove either $(v, \beta_i)$ or $(v, \beta_{i+1})$ from it. This transformation is applied until there are no such pairs.

Let $h = (v_0, \beta_0), \ldots$ and $h' = (v'_0, \beta'_0), \ldots$ be two sequences of value changes. We write $h \lhd h'$ if the sequence of values $v_0, v_1, \ldots$ is a prefix of $v'_0, v'_1, \ldots$. In other words, the length of $h$ is smaller than or equal to the length of $h'$ and for all $k$ such that $(v_k, \beta_k)$ occur in $h$ we have $v_k = v'_k$. We write $h \lhd\rhd h'$ if $h \lhd h'$ and $h' \lhd h$. We will use the same notation for histories. Let $h$ and $h'$ be two histories. Then we write $h \lhd h'$ if the relation $\lhd$ holds on the sequences of value changes corresponding to $h$ and $h'$, and similar for $\lhd\rhd$ in place of $\lhd$. For example if the history of a variable $x$ in a run $R$ is

$$h = (1, b_1), (2, b_2), (2, b'_2), (3, b_3), (4, b_4), (5, b_5),$$

then there are two value change sequences of $x$ in $R$:

$$h_1 = (1, b_1), (2, b_2), (3, b_3), (4, b_4), (5, b_5) \text{ and}$$
$$h_2 = (1, b_1), (2, b'_2), (3, b_3), (4, b_4), (5, b_5).$$

Obviously $h \lhd\rhd h_1$ and $h \lhd\rhd h_2$ (the relation $\lhd\rhd$ always holds between a history and the value change sequence obtained from this history).

Consider another example. Assume that the histories of variables $x$ and $x'$ in runs of $P$ and $x'$ in $P'$ with the same initial store are, respectively

$$h = (1, b_1), (2, b_2), (3, b_3), (4, b_4), (5, b_5), \text{ and}$$
$$h' = (1, b'_1), (2, b'_{2,1}), (2, b'_{2,2}), (3, b'_3), (4, b'_4), (5, b'_5).$$

Then we have $h \lhd\rhd h'$. This suggests that there may be a correspondence between $(b_1, b_2, b_3, b_4, b_5; x)$ and $(b'_1, b'_{2,1}, b'_3, b'_4, b'_5; x')$ and also between $(b_1, b_2, b_3, b_4, b_5; x)$ and $(b'_1, b'_{2,2}, b'_3, b'_4, b'_5; x')$.

We are going to use the notions of history and sequence of value changes in translation validation as follows:

1. Run an interpreter several times on $P$ and $P'$ on a randomly generated store, record histories of variables and *guess* a block and variable correspondence using the corresponding histories.
2. *Prove* that the guessed correspondence is, indeed, a correspondence;
3. *Using* this correspondence, *prove* the equivalence of $P$ and $P'$.

This paper is only concerned with the first part of this process. Note that in the first part we only *guess* a correspondence, verification of this correspondence is not described here. Since we only guess a correspondence, we will refer to it in the sequel as a *guessed correspondence*.

On interpreting a program, if the content of a memory location is used without any prior initialization, then that memory location is initialized with a random value. This is the reason for calling this technique *random interpretation*. Moreover, since there is no guarantee that random interpretation terminates, we abort it after some number of steps.

We guess a correspondence by making several runs of the two programs and memorizing histories of variables. These histories are then compared using the relation $\lhd\rhd$ for terminated runs and $\lhd$ for aborted ones.

Of course, after guessing a correspondence the verification may fail, after which we can try to guess another correspondence. We think that our technique can nicely complement other existing techniques for the following reasons.

(i) The notion of correspondence (as formalized here, or similar notions) seems to be fundamental in all approaches to translation validation. If one wants to implement a validating compiler, then the compiler should produce a correspondence. However, translation validation of third-party compilers requires some way of finding a correspondence. (ii) Our technique for guessing a correspondence does not use symbolic evaluation or proofs and is, therefore, cheap. Moreover, the space of all possible correspondences is huge, while our techique normally guesses a reasonably-sized correspondence after a small number of runs. (iii) Other techniques can be combined with ours. For example, if each of the two programs contains a single copy of a call of the same function, we can require that every correspondence includes the blocks with the function calls. In general, one can combine random interpretation with a symbolic interpretation.

One can argue that the relationship between histories of variables and correspondences is loose. For example, one might argue that the same variable in the optimized program may correspond to several variables in the original one. To tackle this problem, the original program $P$ and the optimized program $P'$ are transformed into their SSA forms prior to interpreting them. In SSA form a block in which a variable can changes its value is uniquely identified by this variable, therefore the notion of correspondence can be simplified by using only sequences of variables. Moreover, in SSA form, instead of using value change sequences, we can simply consider the histories of variables to guess a variable correspondence.

## 5 The Intermediate Language *IL* and the Memory Model

**The Syntax of *IL*.** The original program and its optimized version are written in the same low-level intermediate language, which is subsequently called *IL*. This section discussed the syntax and semantics of this language.

Figure 1 describes the syntax of *IL*. Instructions consist of move instruction, jump instruction, conditional jump instruction, and return instruction. Every instruction has a unique label $l_1$, and all of them but jump have the label $l_2$ of the next instruction. We sometimes denote an instruction by its label. At this early stage, the *IL* language does not include function calls. Expressions in the *IL* language are of the following forms: integer constant, register[2], global variable, escaping memory location[3], binary arithmetic operation, and relational operation.

**The Chunk Memory Model.** Hitherto, a memory has usually been modelled as an infinitely long array. In this model its is assumed that memory locations for global variables do not overlap with each other and with other memory locations. Similarly to memory locations for registers. Moreover, each stack frame is represented by a finite sub-array of the infinitely long array.

To prove program equivalences, we sometimes have to provide evidence that updating a variable does not change the values of other variables. For instance, updating a global variable does not affect the values of other global variables, that is two global variables $g_1$ and $g_2$ never refer to the same memory location. In the *IL* language global variables $g_1$ and $g_2$ are written as $[g_1]$ and $[g_2]$, where $g_1$ and $g_2$ are memory addresses. In the above memory model, to provide evidence that the memory locations do not overlap, we have to show that there exist integers $g_1$ and $g_2$ such that for every integer $n$, $g_1 + n \neq g_2$. This statement is obviously false.

In this section a new memory model is proposed. One may think of this model as corresponding to the memory model of C. The model describes a memory as a collection of chunks. A *chunk* is a finite, contiguously allocated set of objects. In this memory model a register can be considered as a chunk of size one. Thus, there is no difference between registers and ordinary memory pieces. The values stored in chunks can either be constants or references to some chunks. Furthermore, in this paper we

[2] In this paper the notions of register and temporary are used interchangeably.

[3] An escaping memory location is a memory location whose address can be taken.

| | | |
|---|---|---|
| *Instruction* | $i$ | $::= l_1 : mi\, l_2 \mid l_1 : \texttt{jump}(o) \mid l_1 : \texttt{cjump}(rel, l)\, l_2$ <br> $\mid l_1 : ret$ |
| *Move Instruction* | $mi$ | $::= lval \leftarrow e$ |
| *Left Value* | $lval$ | $::= r \mid [a]$ |
| *Return* | $ret$ | $::= \texttt{return} \mid \texttt{return}\, r$ |
| *Expression* | $e$ | $::= o \mid rel$ |
| *Relational Operation* | $rel$ | $::= o_1\, rel\, o_2$ |
| *Relational Operator* | $rop$ | $::= > \mid \geq \mid < \mid \leq \mid = \mid \neq \mid \ldots$ |
| *Arithmetic Operation* | $arith$ | $::= o_1\, bop\, o_2$ |
| *Arithmetic Operator* | $bop$ | $::= + \mid - \mid * \mid / \mid \ldots$ |
| *Operand* | $o$ | $::= n \mid r \mid g \mid [a] \mid arith$ |
| *Address* | $a$ | $::= r \mid g \mid a + ao \mid ao + a \mid a - ao$ |
| *Address Offset* | $ao$ | $::= r \mid n \mid ao_1\, bop\, ao_2$ |
| *Integer Constant* | $n$ | $::= \mathcal{Z}$ |
| *Register* | $r$ | $::= \mathtt{r}_1 \mid \mathtt{r}_2 \mid \ldots$ |
| *Global Name* | $g$ | $::= \langle identifiers \rangle$ |
| *Label* | $l$ | $::= \mathcal{N}$ |

**Fig. 1.** The intermediate Language *IL*

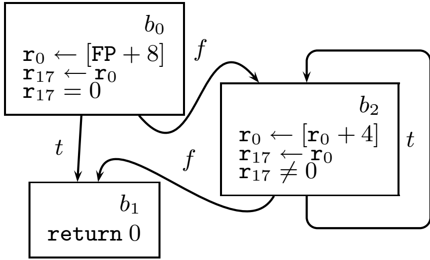


**Fig. 2.** The linked list traversal program of Example 1

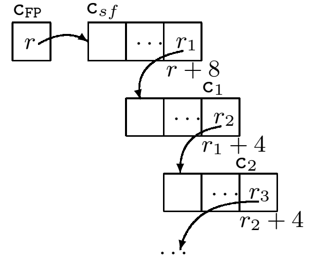


**Fig. 3.** The chunk-based memory after interpreting the program in Figure 2

focus on intra-procedural optimizations, and thus there is one chunk dedicated to representing the current stack frame. Memory locations on this particular chunk are often called *stack memory locations*. The new memory model is called *the chunk memory model*. This model can be used to provide evidence that some variables never refer to the same location. For example, such information can be provided by a language standard (pointers to two incompatible types cannot coincide, memory allocation operator always returns a new piece of memory etc.).

*Example 1.* Consider the program in Figure 2. Here, FP denotes the frame pointer. In a higher-level language the program can be viewed as a program that walks over a linked list. With this understanding, the memory selection $[\mathtt{r}_0 + 4]$ denotes the next element of a node in the linked list.

The memory after interpreting the program is depicted in Figure 3. The chunk $\mathtt{c}_{\mathtt{FP}}$ represents the frame pointer, and the chunk $\mathtt{c}_{sf}$ is the one dedicated to representing the current stack frame. The length of the chained chunks depends on

- the random initial value $r_1$ assigned to the chunk $\mathtt{c}_{sf}$ at the address $r + 8$, where $r$ is the random initial value for the frame pointer $\mathtt{FP}$ and $r_1$ is a reference to the chunk $\mathtt{c}_1$; and
- the random initial value $r_{i+1}$ assigned to the chunk $\mathtt{c}_i$ at address $r_i + 4$, where $r_{i+1}$ is a reference to the chunk $\mathtt{c}_{i+1}$, for $i > 1$.

**Formal Semantics of *IL*.** First we introduce a notion of values. Evaluation of an expression results in a value, which can either be an integer constant or a reference to a chunk. The following definition describes values formally:

$$\mathit{Value}\ v\ ::=\ n \mid \mathit{ref};\ \ \mathit{Reference}\ \mathit{ref}\ ::=\ (c, n);\ \ \mathit{Chunk}\ c\ ::=\ \mathtt{c}_1 \mid \mathtt{c}_2 \mid \ldots$$

A *reference* is a pair $(c, n)$, where $c$ denotes the chunk to which the reference points and $n$ denotes an index of the chunk. References are considered as memory location.

To describe the semantics, we introduce two operations on memory states. Denote by $M$, $R$, and $V$ the sets of all memory states, references, and values, respectively. The functions

$$\mathtt{sel} : M \times R \to V \ \text{ and } \ \mathtt{upd} : M \times R \times V \to M$$

access (respectively, update) memory states. The value $\mathtt{sel}(m, r)$ is the content of the memory state $m$ at the address $r$, where $r$ is a reference. The memory state $\mathtt{upd}(m, r, v)$ is obtained from the memory state $m$ by updating $m$ at the address $r$ with the value $v$.

To define the dynamic semantics of IL, we consider registers and names of global variables as references $(c, 0)$ for some chunk $c$. The association between them and such references is formalized using the notion of *static environment*. The environment consists of two partial injective functions: $Env_n$, which maps names of global variables and registers to references; and $Env_l$, which maps integers to instructions.

The dynamic semantics of *IL* is specified in terms of *operational semantics* given by a simultaneous inductive definition of the following relations:

$$\begin{array}{l} \mathit{Instruction\ interpretation} : (m, i) \mapsto (m', i'); \\ \mathit{Expression\ interpretation} :\ (m, e) \mapsto v. \end{array}$$

The instruction interpretation $(m, i) \mapsto (m', i')$ means the following: interpreting the instruction $i$ with the memory state $m$ yields a new memory state $m'$, and the interpretation is followed by interpreting the instruction $i'$ with the new memory state $m'$. Likewise for the expression interpretation, but the evaluation does not change the memory state.

Due to lack of space, we only show some rules in the inductive definition. First, memory can only be accessed through references. In Example 1 the evaluation of $\mathtt{FP}+8$ in $[\mathtt{FP} + 8]$ must result in a reference. The following rule describes this situations:

$$\frac{(m, a) \mapsto \mathit{ref} \quad \mathtt{sel}(m, \mathit{ref}) = v}{(m, [a]) \mapsto v}$$

Arithmetic operations on references are only applied to the index. Adding two references and subtracting a reference from a constant do not make any sense. Thus, the rules describing binary operations have to distinguish the evaluation results of their operands, for example:

$$\frac{(m, o_1) \mapsto (c, n_1) \quad (m, o_2) \mapsto n_2}{(m, o_1 + o_2) \mapsto (c, n_1 + n_2)} \quad \frac{(m, o_1) \mapsto (c, n_1) \quad (m, o_2) \mapsto (c, n_2)}{(m, o_1 - o_2) \mapsto n_1 - n_2}$$

The right-hand side rule above specifies that subtracting a reference from another reference is allowed as long as both references point to the same chunk.

Rules for instructions can be described similarly. For example, assigning a value to a register is described by the following rule for move instructions:

$$\frac{Env_n(r) = ref \quad (m, e) \mapsto v \quad Env_l(l_2) = i}{(m, l_1 : \ r \leftarrow e\ l_2) \mapsto (\mathtt{upd}(m, ref, v), i)}$$

The memory location where the value is stored is obtained by looking up $Env_n$.

## 6 Randomized Interpreter

This section discussed a randomized interpreter that imitates the work of the interpreter except that the memory state is generated using random number generator.

**The Algorithm of the Randomized Interpreter.** Consider again the program in Figure 2. The content of memory location denoted by $\mathtt{FP} + 8$ is used but without any prior initialization, so the value of evaluating $[\mathtt{FP} + 8]$ for the first time is not known. To denote the unknown value resulting from expression evaluation, we extend the set of values by introducing an *undefined value* $\bullet$. Initially, when the randomized interpreter evaluates a program, every location in the memory state contains $\bullet$. The behavior of the randomized interpreter can be defined by extending the semantics of *IL* to work with the undefined value. However, the extension is not so straightforward for two reasons. First, reading a location containing $\bullet$ results in a change of memory since the location will be filled with a random generated value. Second, *IL* has no type information, so the randomized interpreter does not know if the location should be initialized with a constant or a reference.

The first problem is solved by changing the relation of expression evaluation to allow updating memory states, that is $(m, e) \mapsto_r (m', v)$. To solve the second problem, we introduce a new kind of value called *conditional value*. This kind of value is denoted by $(ref_1 \ ? \ n : ref_2)$, which means that the *definite value* is $n$ if the content of $ref_1$ was firstly initialized to a reference, or otherwise $ref_2$. Both $n$ and $ref_2$ are often called the *definite forms* of the conditional value. Furthermore, we also introduce a function $rand$ that generates random pairs $(ref, n)$, where $ref$ is a reference to a new chunk. The content of a newly created chunk is $\bullet$ everywhere.

A conditional value might become definite at some point during an interpretation. For example, multiplication can only be applied to integer operands. Thus, in multiplying a conditional value $(ref \ ? \ n_1 : ref_1)$ with an integer $n_2$, the conditional value gets

definite to the integer $n_1$, and the content of $ref$ is known to be firstly initialized with a reference. To capture this, we introduce a new operation on memory:

$$\texttt{def} : M \times R \times \{0, 1\} \to M.$$

The operation $\texttt{def}(m, ref, b)$ returns a new memory state $m'$ obtained from $m$ by updating with a definite value $defv$ every reference $ref'$ in $m$ at which $\texttt{sel}(m, ref')$ is $(ref \mathbin{?} defv_1 : defv_2)$ for some definite values $defv_1$ and $defv_2$, such that, if $b$ is one, then $defv$ is equal to $defv_1$, otherwise $defv_2$.

Due to lack of space, we only show some rules describing the algorithm of the randomized interpreter. Suppose, on evaluating $[a]$ with a memory state $m$, $a$ evaluates to a reference $v_a$, but the memory selection $\texttt{sel}(m, v_a)$ yields the undefined value. The interpreter then generates a random conditional value as described by the following rule:

$$\frac{(m, a) \mapsto_r (m', ref_a) \quad \texttt{sel}(m', ref_a) = \bullet \quad rand() = (ref, n)}{(m, [a]) \mapsto_r (\texttt{upd}(m', ref_a, (ref_a \mathbin{?} ref : n)), (ref_a \mathbin{?} ref : n))}$$

If $v_a$ is a conditional value, then at least one of its definite forms is a reference, and in turn, $v_a$ becomes definite to one of its definite references, as shown in the following rule:

$$\frac{(m, a) \mapsto_r (m', (ref' \mathbin{?} ref_a : n_a)) \quad \texttt{sel}(\texttt{def}(m', ref', 1), ref_a) = v \quad v \neq \bullet}{(m, [a]) \mapsto_r (\texttt{def}(m', ref', 1), v)}$$

In the case that both definite forms are references, the interpreter has to make a random choice, which can easily described by non-deterministic rules. For instructions, the following rule describes the algorithm for interpreting move instructions:

$$\frac{(m, a) \mapsto_r (m', v_a) \quad (m, e) \mapsto_r (m'', v) \quad (m'', a) \mapsto_r (m'', ref) \quad Env_l(l_2) = i}{(m, l_1 : \; [a] \leftarrow e \; l_2) \mapsto_r (\texttt{upd}(m'', ref, v), i)}$$

The address $a$ of $[a]$ above is evaluated twice in order to make the order of interpretation irrelevant.

Again, due to limited space, we omit the proofs of soundness and completeness of the interpreter with respect to the *IL* semantics.

**Escaping Memory Locations in SSA.** In order to preserve SSA property we have to ensure that each memory word is written only once. In many compiler textbooks, memory is considered as a "variable". We assume to have two new expressions to the *IL* syntax, one is `store` expression for creating a new value (of the entire memory), and the other is `load` which is similar to `sel` but occurs in the syntactic level.

Consider an excerpt of a program below and its SSA form:

$$\begin{array}{l} \texttt{r}_1 \leftarrow [\texttt{r}_2] \Rightarrow \texttt{r}_1 \leftarrow \texttt{load}(M_0, \texttt{r}_2) \\ [\texttt{r}_2] \leftarrow \texttt{r}_1 \Rightarrow M_1 \leftarrow \texttt{store}(M_0, \texttt{r}_2, \texttt{r}_1) \end{array}$$

This SSA form does not conform to the *IL* semantics. Recall that in the chunk memory model registers are part of the memory. Thus, the first instruction above updates memory $M_0$, but the second instruction uses the old $M_0$ as an argument of `store`.

An alternative solution to this problem is to leave the assignment $[a] \leftarrow e$ intact, but dynamically create and evaluate a new temporary $t$ and an assignment $t \leftarrow e$ whenever the former assignment is evaluated. In detail, for every assignment $[a] \leftarrow e$, suppose $a$ and $e$ evaluate to $ref$ and $v$ respectively, we create an assignment of $v$ to some chunk. The chunk is *associated* with $ref$ and the point where $[a] \leftarrow e$ occurs.

First, we introduce a new environment $Env_p$ which maps pairs of references and instruction labels (program points) to references. The resulting reference is said to be *associated* with the pair of reference and label given as arguments to $Env_p$. The following rule describes formally the above solution to the SSA problem:

$$\frac{(m,a) \mapsto ref \quad (m,e) \mapsto v \quad Env_l(l') = i' \quad Env_p(ref,l) = ref'}{m, l_1 :\ [a] \leftarrow e\, l_2 \mapsto (\mathtt{upd}(\mathtt{upd}(m, ref, v), ref', v), i')}$$

Note that, for the sake of clarity, the above rule assumes that we do not deal with any conditional value.

Introducing a new temporary for every escaping variable yields many histories to be analyzed. To reduce the number of histories, first we do not record values stored in memory locations of the current stack frame. These memory locations represent variables in the program. Since we dynamically create a new temporary every time we write to a stack memory location, the resulting temporaries are the SSA form of the variables represented by these memory locations. This condition also holds when a stack memory location represents more than one variable in the program.

Second, it is not necessary to create a new temporary if the escaping memory location is not represented by any stack memory location. That is, for any instruction $l_1 :\ [a] \leftarrow e\, l_2$ where $a$ evaluates to $(c, n)$ for some index $n$, but chunk $c$ does not represent the stack, we do not create a new temporary. Again, for simplicity, the following definition assumes that we do not deal with any conditional value:

$$\frac{(m,a) \mapsto ref \quad ref = (\mathtt{c}_{sf}, n) \quad (m,e) \mapsto v \quad Env_l(l_2) = i' \quad Env_p(ref,l_1) = ref'}{m, l_1 :\ [a] \leftarrow e\, l_2 \mapsto (\mathtt{upd}(\mathtt{upd}(m, ref, v), ref', v), i')}$$

$$\frac{(m,a) \mapsto ref \quad ref = (c, n) \quad c \neq \mathtt{c}_{sf} \quad (m,e) \mapsto v \quad Env_l(l_2) = i'}{m, l_1 :\ [a] \leftarrow e\, l_2 \mapsto (\mathtt{upd}(m, ref, v), i')}$$

The chunk $\mathtt{c}_{sf}$ is the chunk representing the current stack frame.

Third, recall that writing to a memory location like $(c, n)$ above gives rise to a program's side-effect. To be equal, the original program $P$ and the optimized program $P'$ must have the same side-effects. That is, for every instruction $l_1 :\ [a] \leftarrow e\ l_2$ in $P$ there is a corresponding instruction $l'_1 :\ [a'] \leftarrow e'\ l'_2$ in $P'$ such that evaluations of $a$ and $a'$ yield the same sequence of references, and also evaluations of $e$ and $e'$ yield the same sequence of values. For this purpose, we statically add a new instruction $l_0 :\ r \leftarrow a\, l_1$ before the instruction $l_1$, and a new instruction $l'_0 :\ r' \leftarrow a'\ l'_1$ before the instruction $l'_1$, where $r$ and $r'$ are new registers. Thus, in order to be equivalent, $r$ and $r'$ must correspond to each other. Moreover, having $r$ and $r'$, we do not have to create histories for the memory locations referred by $a$ and $a'$ above. Hence, the number of histories to be analyzed decreases.

The data produced by the randomized interpreter are then processed by an analyzer. The analyzer implements an algorithm for examining the value change sequences of all

variables, and guessing a basic block and variable correspondence. A full description of the analyzer is given in the extended version of this paper.

## 7 Experimental Results

This section describes the results of some experiments that have been conducted. The compiler used in the experiments is the *GNU C compiler* (*GCC*) 3.3.3. In every compilation, the compiler is instructed to dump the RTL, which is the intermediate representation used by the GCC, after performing the machine dependent reorganization. Then, the RTL dump is translated into the *IL* language, which in turn is interpreted by the randomized interpreter.

More precisely, in each experiment, programs are compiled twice, the first compilation is performed without any optimization (`O0`), and the second one with the (`O3`)-level optimization. The latter optimizations typically include *constant folding*, *copy* and *constant propagation*, *dead code* and *unreachable code elimination*, *algebraic simplification*, *local and global common subexpression elimination* followed by *jump optimization*, *partial redundancy elimination*, *loop invariant hoisting*, *induction variable elimination* and *strength reduction*, *branch optimization*, *loop inversion*, *loop reversal*, and *loop unrolling*.

For more extensive experiments we ran the randomized interpreter on the source code of GCC 3.3.3. The interpreter is developed incrementally, and the current implementation only supports 4-byte integer mode. The interpreter at the moment could interpret $299$ functions out of $5,714$ functions which comprise the core of GCC. We are still developing the interpreter further to make it able to interpret all functions in the source of GCC. Most of the GCC functions that can be interpreted by the randomized interpreter are small functions; in average $25$ lines of code.

Table 1 shows the result of running the interpreter on the source of GCC. We divide the table into several columns based on the size of the functions. Information that we obtain from the experiments is the number of points and variables in the correspondence, the number of visited variables during the interpretations, percentage of code coverage, visited branches, and time statistics. In the experiments the interpreter is set to execute at most $10,000$ lines of code.

For small functions whose sizes are less than $10$ lines of code, the original and the optimized versions are almost the same. Table 1 shows that the interpreter could cover almost all code and visit all branches in these functions. Thus, the analyzer could produce a high percentage of block correspondence. For functions whose sizes are greater than $10$ but less than $50$ lines of code, the interpreter could still cover a large portion of the functions and also visit most of their branches. The code coverage is important since more lines of code that can be covered, the clearer the behaviors of the functions can be described, and the more block correspondence can be produced. Moreover, most of control variables are those used in the conditional expressions in the branches, so the more branches are visited the more point correspondence can be produced.

For functions, whose sizes are greater than $50$ lines of code, the interpreter has problem with covering all code in these functions. Table 1 shows that more than $80\%$ of code is not covered. This causes the percentage of point correspondence small. The code

**Table 1.** The result of running the randomized interpreter and the analyzer on the source of GCC

| | $loc \leq 10$ | | $10 < loc \leq 25$ | | $25 < loc \leq 50$ | |
|---|---|---|---|---|---|---|
| | $P$ | $P'$ | $P$ | $P'$ | $P$ | $P'$ |
| Blocks in correspondence | 83.77% | 81.92% | 35.05% | 29.56% | 17.92% | 20.43% |
| Variables in correspondence | 91.97% | 93.16% | 76.55% | 73.72% | 87.90% | 86.04% |
| Number of visited variables | 12.28 | 10.46 | 26.2 | 18.61 | 19.33 | 20.4 |
| Code coverage | 90.16% | 88.82% | 50.37% | 51.58% | 30.71% | 24.65% |
| Visited branches | 97.74% | 96.62% | 72.66% | 75.99% | 41.21% | 43.33% |
| Interpretation time | $0.056s$ | $0.022s$ | $0.170s$ | $0.090s\%$ | $0.155s$ | $0.046s$ |
| Analysis time | $0.110s$ | $0.102s$ | $0.579s$ | $0.617s$ | $1.083s$ | $0.653s$ |

| | $50 < loc \leq 100$ | | $loc > 100$ | |
|---|---|---|---|---|
| | $P$ | $P'$ | $P$ | $P'$ |
| Blocks in correspondence | 11.24% | 9.06% | 8.06% | 6.03% |
| Variables in correspondence | 90.04% | 67.01% | 95.23% | 67.08% |
| Number of visited variables | 19.0 | 18.5 | 21.0 | 20.0 |
| Code coverage | 14.61% | 15.81% | 8.83% | 10.67% |
| Visited branches | 23.81% | 25.25% | 14.34% | 17.34% |
| Interpretation time | $0.004s$ | $0.002s$ | $0.004s$ | $0.001s$ |
| Analysis time | $0.001s$ | $0.002s$ | $0.002s$ | $0.001s$ |

coverage problem has long been known in program testing, that is to produce test cases that could cover all code in the function. We plan to tackle this problem by interpreting the function and its optimized version from points that are known to correspond to each other, and also combining our technique with symbolic interpretation to produce random inputs that can cover all code in these functions.

## 8 Discussion and Conclusion

Compared to the technique proposed in other papers, our technique is cheap since it requires no theorem proving or symbolic evaluation. Moreover, our technique can give reasonable results even in cases where other techniques do not work. For example, in the case of the loop reversal optimization our technique can still find correspondence between variables before and after the loop. But the price to pay is that the guessed correspondence has to be verified by a theorem prover. In most examples we studied such a verification was trivial. However, to get a full understanding of the technique, our system has to be combined with a VC generator and VC checker.

There are also examples when our technique may not be appropriate. For example, using random values may be inappropriate when one of the branches is "hard" to reach. Gulwani and Necula [3] propose a very interesting technique, also based on random inputs, for solving this problem, but it is only applicable to a very special class of programs. We believe that our technique can be improved both by "correcting" random values as in [3] and also by mixing symbolic interpretation with the random one.

The definition of basic block and variable correspondences in Section 3 does not capture some optimizing transformations. For example, consider the following programs:

| $P$ : | $P'$ : |
|---|---|
| $x_1 \ := \ 0;$ | $x_1 \ := \ 1;$ |
| **do** | **do** |
| $\quad i \ := \ i + 1;$ | $\quad i \ := \ i + 1;$ |
| $\quad x_1 \ := \ 1;$ | $\quad x_2 \ := \ i + x_1;$ |
| $\quad x_2 \ := \ i + x_1;$ | **while** $i < n$ |
| **while** $i < n$ | $x_3 \ := \ x_1;$ |
| $x_3 \ := \ x_1;$ | |

$P'$ is obtained by applying loop invariant hoisting and dead code elimination to $P$. The loop bodies of the two programs correspond to each other. Moreover, the loop body of $P$ also corresponds to the assignment $x_1 \ := \ 1$ in $P'$ since an instance of this assignment is executed many times in $P$. However, our definition of correspondence does not capture this case. Indeed, it is hard to give a simple but general formal definition of basic block and variable correspondence that can capture all existing optimizing transformations. Our definition can be extended further to capture more transformations. For example, by adding some properties into the definition to allow a block to correspond to more than one other block in the runs will capture the correspondence of the above programs. Although our definition of correspondence does not capture the above case, in the SSA form, the randomized interpreter and the analyzer can produce the correspondence of the loop body of $P$ and the assignment $x_1 \ := \ 1$ in $P'$.

We are still improving the definition of correspondence. The definition we provide in this paper is considerably simple and easy-to-understand, but nonetheless captures the notion of correspondence needed to prove the equivalence of two programs. Particularly in the above example, without the correspondence of $x_1$, but as long as we can establish the correspondence of variables $i$, $n$, $x_2$, and $x_3$, the equivalence of $P$ and $P'$ can easily be proved. Indeed, the randomized interpreter and the analyzer can establish such a correspondence.

## References


1. B. Alpern, M.N. Wegman, and F.K. Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages (POPL 1988)*, pages 1–11, 1988.
2. Yi Fang. Personal communication over emails, 2005.
3. S. Gulwani and G.C. Necula. Global value numbering using random interpretation. In N.D. Jones and X. Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 342–352. ACM Press, 2004.
4. Clara Jaramillo, Rajiv Gupta, and Mary Lou Soffa. Capturing the effects of code improving transformations. In *IEEE PACT*, pages 118–123, 1998.
5. Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

6. George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN Conference on Principles of Programming Languages Design and Implementation (PLDI)*, pages 83–95, June 2000.
7. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. *LNCS*, 1384, 1998.
8. Amir Pnueli, Lenore Zuck, Yi Fang, Benjamin Goldberg, and Ying Hu. Translation and run-time validation of optimized code. *ENTCS*, 70(4):1–22, 2002.
9. Xavier Rival. Symbolic transfer function-based approaches to certified compilation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–13. ACM Press, 2004.
10. Robert van Engelen, David B. Whalley, and Xin Yuan. Automatic validation of code-improving transformations. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 206–210. Springer-Verlag, 2001.
11. L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. Voc: A methodology for the translation validation of optimizing compilers. *j-jucs*, 9(3):223–247, March 2003.