Practical and Accurate Low-Level Pointer Analysis

Bolei Guo Matthew J. Bridges Spyridon Triantafyllis Guilherme Ottoni
Easwaran Raman David I. August
Department of Computer Science, Princeton University
{bguo, mbridges, strianta, ottoni, eraman, august}@princeton.edu

Abstract

Pointer analysis is traditionally performed once, early in the compilation process, upon an intermediate representation (IR) with source-code semantics. However, performing pointer analysis only once at this level imposes a phase-ordering constraint, causing alias information to become stale after subsequent code transformations. Moreover, high-level pointer analysis cannot be used at link time or run time, where the source code is unavailable.

This paper advocates performing pointer analysis on a low-level intermediate representation. We present the first context-sensitive and partially flow-sensitive points-to analysis designed to operate at the assembly level. As we will demonstrate, low-level pointer analysis can be as accurate as high-level analysis. Additionally, our low-level pointer analysis also enables a quantitative comparison of propagating high-level pointer analysis results through subsequent code transformations, versus recomputing them at the low level. We show that, for C programs, the former practice is considerably less accurate than the latter.

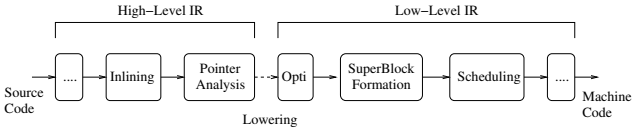

Figure 1. Traditional compiler organization

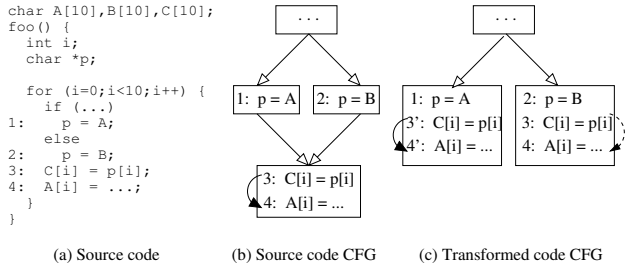

Figure 2. Conservative propagation of dependence information

1 Introduction

Pointer analysis is an important tool for modern optimizing compilers. The aliasing information obtained from pointer analysis enables aggressive code optimizations, such as redundant store elimination and load/store reordering. Traditionally, pointer analysis is performed very early in the compilation process, on a high-level intermediate representation (IR) containing source-level semantic information. The result of the analysis, often in the form of dependence edges, is annotated on the IR and must be conservatively maintained by subsequent code transformations.

Unfortunately, performing pointer analysis only once on a high-level IR imposes a phase-ordering constraint upon the compilation process, as illustrated in Figure 1. The fact that pointer analysis takes high-level IR as input requires that it be performed before the code lowering phase of the compiler. Subsequent code transformations must conservatively propagate the analysis results, potentially diluting their precision. Ultimately, less precise alias information will limit the effectiveness of memory optimizations and other optimizations such as scheduling. For example, consider the C code shown in Figure 2(a) and the control-flow graph corresponding to its loop body in Figure 2(b). Because p at instruction 3 may point to either array A or array B, the high-level analysis correctly determines that instructions 3 and 4 may access the same memory location. This memory dependence is illustrated by the arc between instructions 3 and 4 in Figure 2(b). After superblock formation [1] is performed on this code, the control-flow graph illustrated in Figure 2(c) is obtained. Instructions 2, 3 and 4 are grouped into a superblock. Two new instructions (3' and 4') are created through tail duplication. Clearly, there is no longer any memory dependence between instructions 3 and 4 in Figure 2(c). However, the superblock formation algorithm, without additional knowledge of how to perform

memory disambiguation, conservatively propagates the dependence relation to the transformed code, illustrated by the dashed arc in Figure 2(c). This conservative, unnecessary dependence prevents the scheduler from moving instruction 4 before 3, for example. Conceivably, code transformations such as superblock formation can be augmented to incrementally update pointer analysis results in a fully precise way. However, this would mean that each code transformation would have to incorporate a pointer analysis component comparable in both implementation difficulty and computational complexity to the original pointer analysis algorithm.

Additionally, high-level pointer analysis is only applicable when the source code is available. Therefore, tasks such as binary re-optimization, link-time optimization, and runtime optimization cannot benefit from it.

To overcome these limitations, this paper argues for performing pointer analysis on a low-level IR. Not only can this eliminate the conservative dependence propagation problem, but it is also an important step toward fully source-language independent memory disambiguation for binaries. In this paper, we present the first context-sensitive and partially flow-sensitive points-to analysis that operates at the assembly level, which we call the VELOCITY Low-Level Pointer Analysis (VLLPA). We compare the accuracy of VLLPA to that of a state-of-the-art high-level pointer analysis and find that our analysis matches the high-level analysis closely. Additionally, for the C programming language, we identify possible accuracy loss that comes with the loss of source language information. Finally, we quantitatively characterize the impact of conservatively propagating memory dependences through optimizations by comparing the stale dependence information originating from the IMPACT compiler's [2] high-level analysis against that computed by our low-level analysis after code transformations.

In summary, the contributions of this work are:

- The first context-sensitive and partially flow-sensitive low-level points-to analysis algorithm.
- A characterization of the precision loss of the points-to algorithm due to translation from a high-level IR to a low-level IR for the C programming language.
- A quantitative analysis of the conservative memory dependence propagation problem.

We discuss related work in Section 2. Section 3 presents the VLLPA algorithm. Section 4 explores the information lost in the IR lowering process. Section 5 compares the accuracy of our analysis to that of the IMPACT compiler's high-level pointer analysis. Section 6 dissects the conservative dependence propagation problem in detail. Section 7 describes the impact of different pointer analyses on optimization effectiveness. Finally, we conclude in Section 8.

2 Related Work

There is a wealth of literature on pointer analysis [3], though very few papers focus on low-level program representations. Below, we discuss the most salient aspects of memory analysis algorithms and explain the approach taken in this paper. Table 1 summarizes several related algorithms with respect to these aspects.

Input Type Most pointer analysis algorithms operate on high-level IRs that preserve information from the source code [2, 4, 5, 6]. As explained in Section 1, performing pointer analysis at the high level leads to conservative dependence propagation. Low-level pointer analysis has been performed before [7, 8] and requires different assumptions.

Language Features Pointer analysis algorithms often support only a subset of the source language's features [6, 9, 10]. The language features supported can dramatically affect the algorithm's complexity. For example, Steensgaard's algorithm is almost linear when applied to a simple language [9], but becomes exponential when aggregate fields are added to the language [5]. The IMPACT compiler's pointer analysis algorithm [2] can handle all features of C, including function pointers and pointer arithmetic. VLLPA handles all assembly language features, thus all features of any higher level language.

Context Sensitivity Context-insensitive algorithms treat the whole program as a single interprocedural control flow graph and suffer from the problem of *unrealizable paths* where values from one call site can be propagated via the callee to another call site [11, 9, 8]. Conversely, context-sensitive algorithms consider only paths along which calls and returns are properly matched [2, 7, 4, 6]. This can be achieved by computing a *transfer function* that summarizes the effects of each procedure and applying it wherever the procedure is called [6]. Context-sensitive algorithms are more precise, but are usually slower and much more complicated to implement. The algorithm presented in this paper not only uses transfer functions to separate the effects of a called procedure at different call sites, it also ensures that the concrete values propagated from different call sites to the callee can be considered independently when computing aliases within the callee.

Flow Sensitivity Flow-insensitive algorithms ignore the order of statements and need only maintain a single points-to relation for the whole procedure [2, 9, 11]. On the other hand, flow-sensitive algorithms follow the program control flow, maintaining different analysis solutions at different program points [6]. Naturally, flow-sensitive algorithms are more precise, but also

more time- and memory-intensive. VLLPA is partially flow-sensitive; it tracks the values of registers according to their position in the flow graph, but maintains only a single points-to set for each memory location.

Modularity Some pointer analysis algorithms, especially context-sensitive ones such as [4, 10], require the whole program to be memory-resident during analysis. Other algorithms are modular, in the sense that only part of the program and/or the analysis information needs to be present in memory at each point [2, 6, 12]. The algorithm presented in this paper is modular.

Store Abstraction The runtime store can be abstracted using a *store-based* or a *path-based* (*storeless*) approach. The store-based approach uses a finite number of nodes in an abstract store graph to represent a potentially infinite number of runtime locations [6]. The path-based approach names runtime locations by how they are accessed from program variables [13], described using an abstraction called *access path*. As a single memory location may be reachable via multiple access paths, it may have multiple names. Path-based approaches are less appropriate for low-level analyses because program variables are not readily distinguishable at the low level. Hence VLLPA is store-based.

Heap Modeling Heap-allocated objects can either be named according to their allocations sites or modeled in greater detail by shape analysis [14], which is generally more expensive to compute. There is a spectrum of granularity at which heap objects can be distinguished based on their allocation sites. Certain algorithms [12] merge all heap allocation sites together, not distinguishing between heap objects at all, while other algorithms [4] differentiate heap blocks allocated at the same allocation site based on the different call chains reaching that site. There is an obvious trade-off between analysis time/memory consumption and precision. VLLPA limits the allocation context information to at most two consecutive call edges. This is sufficient to provide good precision while preventing too many abstract heap blocks from being produced.

Few techniques for low-level pointer analysis have been developed. Most commercial compilers, such as GCC, only apply a simplistic intraprocedural memory disambiguation technique called *instruction inspection*, which considers two memory references to be non-conflicting based on ad-hoc rules such as "same base register with no intervening redefinitions and different offsets".

Two dataflow-based interprocedural low-level pointer analyses have been proposed. Debray et al. [7] propose a flow-sensitive, context-insensitive alias analysis of executable code that suffers from three major sources of imprecision. First, it does not keep track of any memory content. Second, if two different definitions of a register reach the same join point, then it will widen the value of that register to `ANY`, which denotes all possible addresses. Third, memory addresses are distinguished only by their lower-order bits. As a result, this algorithm can only provide information for 35%-60% of all memory operations.

Balakrishnan et al. [8] propose a flow-sensitive algorithm for analyzing memory accesses in x86 executables. Their analysis computes an over-approximation of values stored in memory locations in addition to registers, but it, like the Debray's algorithm, has the drawback of being context-insensitive. Empirical studies show that flow sensitivity, in the absence of context sensitivity, does not significantly improve precision [15]. Neither of these two works attempts to match the accuracy of their analyses against that of high-level analysis.

In this paper, we will compare our low-level analysis to the IMPACT compiler's high-level pointer analysis [2]. IMPACT's powerful algorithm is one of the few context-sensitive algorithms that can handle large programs from the SPEC benchmark suites and that deal with all features of C. The algorithm uses a path-based memory model and is context-sensitive but flow-insensitive.

3 Low-level Points-to Analysis

Broadly, our proposed algorithm is an iterative context-sensitive and partially flow-sensitive algorithm for low-level code. It combines elements from the Relevant Context Inference (RCI) algorithm [6] and the IMPACT compiler's Modular Interprocedural Pointer Analysis algorithm [2], appropriately modified to work on low-level code.

To achieve context sensitivity, the algorithm computes a *transfer function* for each procedure, which summarizes the effects of a call to that procedure on memory. Obviously, a procedure's transfer function will need to incorporate the transfer functions of its callees. This is done through bottom-up propagation of transfer functions from callees to callers. To handle calls through function pointers, we adopt the technique proposed in [12]: the algorithm starts with an underestimated call graph, which is augmented during the course of the analysis, as possible targets for indirect procedure calls are discovered. The rest of this section presents a detailed description of the algorithm.

3.1 Algorithm Outline

The algorithm consists of four major phases, as shown in Figure 3. Phases 0 to 2 are applied iteratively until a fixed point is reached, then Phase 3 is applied.

Phase 0: A call graph is constructed. If indirect calls are present, only targets that have already been identified

Technique	IR Level	Feature Coverage	Context Sensitive	Flow Sensitivity	Time Complexity	Modular	Store Abstraction
Debray et al. [7]	Assembly	Complete	No	Reg	Exp	Yes	Store-based
Balakrishnan et al. [8]	Assembly	Complete	No	Reg/Mem	Exp	Yes	Store-based
VLLPA	Assembly	Complete	Yes	Reg	Exp	Yes	Store-based
IMPACT [2]	Source	Complete	Yes	No	Exp	Yes	Storeless
RCI [6]	Source	Partial	Yes	Reg/Mem	Exp	Yes	Store-based
Landi & Ryder [10]	Source	Partial	No	Reg/Mem	Exp	No	Equivalence
Emami et al. [12]	Source	Complete	Yes	Reg/Mem	Exp	No	Store-based
Wilson & Lam [4]	Source	Complete	Yes	Reg/Mem	Exp	No	Storeless
Steensgaard [9]	Source	Partial	No	No	Linear	Yes	Storeless
Steensgaard [5]	Source	Complete	No	No	Exp	Yes	Storeless

Table 1. Summary of some Pointer Analysis Algorithms

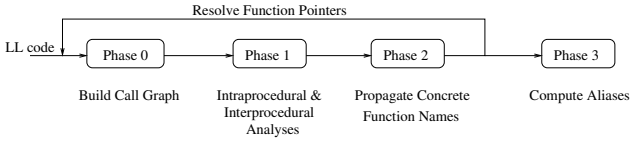

Figure 3. Algorithm Phases

will be taken into account. In the first iteration, all indirect calls are presumed to have no targets. The constructed call graph is divided into strongly connected components (*SCCs*). The SCCs form a directed acyclic graph, referred to as the *SCC-DAG* from here on.

Phase 1: The SCC-DAG is traversed in reverse topological order. Analysis is performed on each procedure assuming unknown initial values for parameters, global variables, and all memory locations reachable from them. This produces a *transfer function* that summarizes the points-to relationships observable by callers of that procedure. By traversing the SCC-DAG in reverse topological order, it is guaranteed that the transfer functions of a procedure's non-recursive callees (not in the same SCC with the caller) are available when the procedure is analyzed. The transfer functions of procedures belonging to the same SCC depend cyclically on each other, and are therefore computed simultaneously using fixed-point iteration.

Phase 2: The SCC-DAG is then traversed in topological order to propagate concrete values of function pointers. This makes the partially resolved call graph more accurate for the next iteration. If any function pointer targets are changed, the algorithm starts over at phase 0, otherwise the algorithm terminates with phase 3.

Phase 3: After the iteration terminates, the SCC-DAG of the complete call graph is traversed once more, in topological order, to propagate all concrete pointer values.

This algorithm differs from both the RCI algorithm [6] and IMPACT's algorithm [2]. A major difference from RCI is that RCI resolves virtual method calls at the beginning of its algorithm through hierarchy analysis. This allows it to avoid the iterative loop over phases 0 to 2, but can lead severe inaccuracies. The two algorithms also handle aliases between a procedure's unknown initial values in very different ways (see Section 3.5). The differences from [2] arise from flow sensitivity. By being flow-insensitive, IMPACT's algorithm can compute the transfer function of a procedure independent of the transfer functions of its callees. As a result, the transfer functions can be produced in an initial prepass phase, and need not be computed iteratively. This is not possible for a flow-sensitive algorithm.

Like the other two analyses, our algorithm is modular, in the sense that during any phase the whole program body need not be memory resident. Only the procedures belonging to the same SCC and the transfer functions need to be in memory simultaneously.

3.2 Store Abstraction

Any static memory analysis algorithm has to represent an unbounded set of runtime memory addresses using a finite set of abstract names. For this purpose our algorithm uses a store-based memory model. According to this model, a program's memory is divided into a set of *abstract structures*, each with a unique name. Each structure can correspond to multiple memory blocks at runtime. An abstract structure is created for each global variable in the program's data segment. The activation frame of a procedure can be represented by one or more abstract structures, according to the calling convention. In our experimental framework there are three such structures for each procedure: incoming parameter space (IP), outgoing parameter space (OP), and local variable space (LV). Finally, heap objects allocated locally within a procedure or any of its descendants in

the call graph are named according to the contexts in which they are allocated. We use the first two call edges at the top of a call chain reaching a static allocation site as the context information. A unique abstract structure is created to represented all heap blocks allocated through call chains reaching a static allocation site that share the common two starting call edges.

Unknown initial values (UIVs) are needed to represent memory blocks accessible by a procedure, but not created by either the procedure or its callees. UIVs will be created for the memory blocks reachable, either directly or indirectly, through parameters or global variables. For a procedure parameter or global variable A, we will use the UIV $[A]$ to represent the memory block pointed to by A. If the field at offset o of a UIV U is a pointer, then a new UIV, named $U@o$, will be needed to represent the possible value of that field. The parameter or global variable from which a UIV is reachable will be referred to as the *base* of the UIV. For example, A is the base of UIVs $[A]$, $[A]@4$, $[A]@8@16$ etc. UIVs are created lazily, whenever they are needed.

Much like C structures, abstract structures have *fields*, distinguished by their offsets from the start of the structure. Therefore an *abstract address* has the form $\langle S, o\rangle$, where S is a structure name and o is an offset. Our algorithm does not differentiate between different array elements. Instead, the offset of the first array element is used as a representative of all elements in the array. If the analysis detects that the field starting at offset o of a structure S is being accessed as an array with element size l, then the element $\langle S, o\rangle$ will be used to represent all elements of the form $\langle S, o + l \times i\rangle$. This will be presented in more detail in Section 3.4.

3.3 Algorithm Highlights

This section describes aspects of VLLPA that distinguish it from existing high-level pointer analysis algorithms.

- One particular difficulty with analysis at the low level arises from the fact that common memory operations, such as array and field accesses, do not appear explicitly in the code. Rather, the analysis has to infer whether a memory operation "looks like" a field and/or array access by examining pointer arithmetic and offset calculations, which can often span several instructions. How VLLPA does this by back tracing and pattern matching is described in Section 3.4.1.

- The number of UIVs in a procedure can be infinite in the presence of recursive data types. To represent them in a finite manner, high-level analysis algorithms such as [6] and [2] generally collapse UIVs with the same data type. Not only is type information not present in the low-level, it is also not trustworthy for the C programming language anyway. VLLPA limits the number of UIVs by observing the points-to sets they belong to. Details are given in Section 3.4.2.

- Since UIVs are essentially access paths, it is possible for two UIVs to refer to the same runtime location. To handle aliases between UIVs context-sensitively, previous store-based analyses either generate different transfer functions for different alias contexts [4] or tag points-to pairs with the alias contexts in which they hold [6]. VLLPA can summarize a procedure assuming no aliases between its UIVs and apply the transfer function properly to account for such aliases should they exist at a call site. Such aliases are rare in practice, hence the overhead is small. This is explained in Section 3.5.

3.4 Intraprocedural Analysis

This section explains in greater detail the intraprocedural aspect of Phase 1. The goal of the intraprocedural analysis stage is to summarize a procedure's effects on memory through a *transfer function*. This analysis calculates points-to sets for registers and memory locations by iterating over the procedure's flow graph until a fixed point is reached. The analysis first transforms the procedure's low-level representation to Static Single Assignment (SSA) form. Since in this form every register is assigned exactly once, we need only maintain a single points-to set for each register, instead of a separate set per register for each program point. This flow sensitivity with respect to registers allowed by SSA does not, however, come with excessive growth of the number of registers, because code growth from SSA is linear in practice, implying a linear growth of registers. Moreover, SSA is done only for pointer analysis; the original code is not modified. Our algorithm is *not* flow-sensitive with respect to pointers in memory, hence we use a single points-to set for each abstract memory location as well.

More formally, the solution to the intraprocedural analysis problem consists of the following three elements.

1. $\mathcal{R}$ – A mapping from registers to sets of abstract addresses, which records the memory locations each register may point to.

2. $\mathcal{M}$ – A mapping from abstract addresses to sets of abstract addresses, which records points-to sets for pointer fields stored in memory.

3. $\mathcal{I}$ – A set of unknown initial values (UIVs) used by the procedure.

Together, $\mathcal{M}$ and $\mathcal{I}$ form the procedure's transfer function.

3.4.1 Analyzing Individual Instructions

In the absence of type declarations, it is not clear which values, either in registers or in memory, represent pointers. Values resulting from arithmetic operations other than addition and subtraction and those residing in floating-point registers are initially assumed to be non-pointers. In all other cases the analysis has to assume that a register or a memory field may be a pointer until it is proven to be a non-pointer, for example, by the fact that it is involved in arithmetic operations with another non-pointer.

In general, the analysis directly modifies $\mathcal{R}$, $\mathcal{M}$, and/or $\mathcal{I}$ only when it encounters instructions that may create or modify pointer values. Such instructions include loads, stores, additions, subtractions, and ϕ-functions created by SSA. Pseudo-code for the actions taken at these instructions is provided in Figure 4. Below we discuss some of the above instruction types in more detail.

Load: $\mathbf{r1 = mem[r2]}$ A load is handled in a straightforward way. The only complication arises if $r2$ points to a UIV's field $\langle U, o\rangle$. In general, memory reachable through a UIV's field should also be represented by a UIV. Since UIVs are lazily created, that second UIV may not exist yet. In this case a fresh UIV named $U@o$ is created.

Store: $\mathbf{mem[r1] = r2}$ Note that, for an abstract address $\langle S, o\rangle \in \mathcal{R}(r1)$, the values in $\mathcal{R}(r2)$ will be *added* to the points-to set $\mathcal{M}(\langle S, o\rangle)$. In the related work this is usually referred to as a *weak update*. A *strong update*, on the other hand, would involve replacing the contents of $M(\langle S, o\rangle)$ with those of $\mathcal{R}(r2)$. Strong updates are possible only under certain conditions, which require a quite complicated mechanism to keep track of (see, for example, the mechanism employed in [6]). Our algorithm eschews strong updates completely for the sake of simplicity and efficiency.

Add: $\mathbf{r1 = r2 + r3}$ When translated to low-level code, non-trivial memory operations, such as array accesses or accesses of fields within arrays of structures, will be translated to a series of register-to-register adds. Therefore it is especially important that a low-level memory analysis algorithm handle this type of instruction correctly. Typically, one of the registers $r2$ or $r3$ will represent a pointer value, serving as the base, whereas the other register will represent an offset. Since we cannot know in advance which register is the pointer, we have to treat the two operands symmetrically, by first assuming that $r2$ is the address base, then making the same assumption for $r3$, and finally taking the union of the results. The pseudo-code of Figure 4 deals only with the first case; the second case is symmetric to it.

If $r2$ indeed carries a pointer value, its points-to set can be retrieved from $\mathcal{R}$. The only problem left is to trace the integer value held in $r3$, which is assumed to have the form $i \times l + c$, where i is a non-constant value, l is the (constant) size of array elements, and c is a constant displacement. The displacement c will be non-zero if the array is a structure field, and/or the elements of the array have themselves fields. The analysis examines the single definition of $r3$ and attempts to pattern-match it to the form $i \times l + c$. This pattern matching is performed by the functions **infer_offset** and **infer_stride**, whose pseudo-code is given in Figure 5. If the pattern matching fails, then we conservatively set $l = 1$ and $c = 0$, essentially meaning that $r3$ can contain all possible offsets. In either case, for every abstract address $\langle S, o\rangle \in \mathcal{R}(r2)$, the element $\langle S, o+c\rangle$ will be added to $\mathcal{R}(r1)$, and the stride l will be recorded appropriately by calling **set_stride**.

3.4.2 Ensuring Termination

Each analysis action described in the previous section can only add new arcs in $\mathcal{R}$ and $\mathcal{M}$ and new elements to $\mathcal{I}$. Therefore these three elements increase monotonically in each iteration. To prove that the analysis terminates, it is sufficient to make sure that there is an upper bound to the solution. This can be done by ensuring that only a finite number of abstract addresses is created.

The number of abstract structures in the global data segment, the activation frame, and those dynamically created are obviously finite. However, as presented up to now, our algorithm could potentially create an unbounded number of UIVs. For example, consider the procedure `f` shown in Figure 6(a). This procedure traverses a linked list whose head is represented by the UIV $[P0]$. It is easy to see that successive iterations will add the values $\langle [P0], 0\rangle$, $\langle [P0]@4, 0\rangle$, $\langle P0@4@4, 0\rangle$, ... to $\mathcal{R}(r1)$. This problem is solved in high-level pointer analyses by variations of the algorithm proposed in [16], which collapses sequences of field accesses beginning and ending with the same type when a new UIV needs to be created. In our example, the access paths `l->next`, `l->next->next`, etc. will be assigned the same UIV as `l`, since all of them are of type `T*`.

VLLPA does not make any use of type information. However, it is possible to combine UIVs *after* they are created by observing the points-to sets they belong to. In the previous example, once it is observed that the register $r1$ can point to either $[P0]$ or $[P0]@4$, it is reasonable to assume that these two structures are used interchangeably. Therefore, it is likely that little information will be lost if $[P0]@4$ is merged with $[P0]$. As a result of the merge, $\langle [P0], 4\rangle$ will now point back to $[P0]$. The next iteration of the analysis will not produce any new UIVs, since the only field that is loaded is already pointing to a UIV.

As a general rule, whenever the points-to set of a register r contains the abstract addresses $\langle U_1, 0\rangle$, $\langle U_2, 0\rangle$, ...,

`r1 = mem[r2]`	`mem[r2] = r1`	`r1 = r2+c`	`r1 = r2+r3`	`r = φ(r1…rn)`

$s2 := \mathcal{R}(r2)$
$s1 := \emptyset$
for $\langle S, o\rangle \in s2$ **do**
  if $S \in I$ **and**
  $\mathcal{M}(\langle S, o\rangle) \cap I = \emptyset$ **then**
  $\mathcal{I} \cup= S@o$
  $\mathcal{M}(\langle S, o\rangle) \cup= \langle S@o, 0\rangle$
  $s1 \cup= \mathcal{M}(\langle S, o\rangle)$
$\mathcal{R}(r1) := s1$
	$s1 := \mathcal{R}(r1)$
	$s2 := \mathcal{R}(r2)$
	for $\langle S, o\rangle \in s2$ **do**
	  $\mathcal{M}(\langle S, o\rangle) \cup= s1$
		$s2 := \mathcal{R}(r2)$
		$s1 := \emptyset$
		for $\langle S, o\rangle \in s2$ **do**
		  $s1 \cup= \langle S, o{+}c\rangle$
		$\mathcal{R}(r1) := s1$
			$s1 := \emptyset$
			$s2 := \mathcal{R}(r2)$
			if $s2 \neq \emptyset$ **then**
			  c := infer_offset($r3$)
			  l := infer_stride($r3$)
			  for $\langle S, o\rangle \in s2$ **do**
			  $s1 \cup= \langle S, o{+}c\rangle$
			  set_stride(S, o, l)
			…
			$\mathcal{R}(r1) := s1$
				$s1 := \mathcal{R}(r1)$
				…
				$sn := \mathcal{R}(rn)$
				$\mathcal{R}(r) := \bigcup_{i=1}^{n} si$

Figure 4. Transfer functions of individual instructions

infer_offset(`r`):

```
if r = r' + c then
  return c
return 0
```

(a) infer_offset

infer_stride(`r`):

```
if r = r' + c then
  return infer_stride(r')
if r = r' * c then
  return c
if r = r' << c then
  return 2^c
return 1
```

(b) infer_stride

set_stride(S, o, l):

for $\langle S, o'\rangle \in domain(\mathcal{M})$ **do**
  if $o' > o$ **then**
    $\mathcal{M}(\langle S, o + (o' - o)\%l\rangle) \cup= \mathcal{M}(\langle S, o'\rangle)$

(c) set_stride

Figure 5. Pseudo-code of intraprocedural analysis subroutines

$\langle U_n, 0\rangle$, where $U_1, U_2, \ldots, U_n$ are all UIVs with the same base, we merge $U_2, \ldots, U_n$ with U_1. To see why this rule results in a finite number of UIVs, consider that UIVs are created *only* at load instructions, as seen in Section 3.4.1. As soon as a UIV U is created by a load $r1 = mem[r2]$, the abstract address $\langle U, 0\rangle$ will be added to $\mathcal{R}(r1)$. Therefore, for every UIV U there must exist a register r, such that $\langle U, 0\rangle \in \mathcal{R}(r)$. After applying the above rule, each register points-to set will contain at most one UIV from each base (since multiple UIVs with the same base would have been combined by the above rule). Therefore there are at most $M \times N$ UIVs, where M is the number of base UIVs and N is the number of registers.

Now that the number of abstract structures has been bounded, we must still make sure that the number of offsets used for each abstract structure is also bounded. To see why this is a problem, consider the function `g` in Figure 6(c). This function traverses an array residing in structure $[P0]$. It is easy to see that the abstract addresses $\langle [P0], 0\rangle$, $\langle [P0], 4\rangle$, $\langle [P0], 8\rangle$, … will be successively added to $\mathcal{R}(r1)$. A high-level memory analysis algorithm, such as [2], can deal with this problem by observing that A (i.e. $[P0]$) is declared as an array, and therefore all its elements are represented by a single element. Unfortunately, this information is not available at the low level. However, we can use the fact that $[P0]$ appears with offsets 0, 4, 8, etc. in the same points-to set as a hint that $[P0]$ is an array with elements of size 4. As a general rule, whenever two abstract addresses of the form $\langle S, o1\rangle$ and $\langle S, o2\rangle$ appear simultaneously in a register's points-to set, where $o1 < o2$, it is assumed that $\langle S, o1\rangle$ is the start of an array with element size $o2 - o1$. Since there is no differentiation of elements of the same array, $\langle S, o1\rangle$ is used to represent $\langle S, o2\rangle$ (as well as any other address of the form $\langle S, o1 + k \cdot (o2 - o1)\rangle$. This can be accomplished by calling the function **set_stride** from Section 3.4.1.

If there are N registers in the procedure, the above rule ensures that at most N different offsets can be in use for each abstract structure. Since the number of abstract structures is bounded, the number of abstract addresses is also bounded, and the algorithm's termination is ensured.

3.5 Interprocedural Analysis

This section describes the interprocedural aspect of Phase 1. Suppose that a procedure F is calling a procedure G. Let M_G and I_G be the memory map and the set of initial values of the callee respectively. These two sets comprise the callee's transfer function. Also, let M_F and I_F be the memory map and UIV set of the caller. When analyzing the call from F to G, the goal is to augment M_F and I_F according to the contents of M_G and I_G respectively.

Obviously, the UIVs of the callee must be translated to appropriate values in the caller. In general, a *set* of abstract addresses in the caller's context will be bound to each UIV

```
typedef struct T {
  int data; T* next;
} T;

f(T* l) {
  while (l != NULL)
    ...
    l = l->next;
}
```

(a) List: source

```
f:
LOOP:
  r1 = φ (param0, r2)
  br r1 == 0 EXIT
  ...
  r2 = mem[r1+4]
  jump LOOP
EXIT:
```

(b) List: low-level

```
int A[100];

g() {
  int *a = A;
  while (...) {
    ... = *a;
    ...
    a++;
  }
}
```

(c) Array: source

```
A:
  reserve 400
g:
  r1 = A
LOOP:
  r2 = φ (r1, r4)
  r3 = mem[r2]
  ...
  r4 = r2 + 4
  br (...)  LOOP
```

(d) Array: low-level

Figure 6. Termination over loops

referenced by the callee. Let β be this mapping from callee UIVs to sets of caller abstract addresses. Let $\mathcal{B}$ be a generalization of β that maps callee abstract addresses to sets of caller abstract addresses as follows:

$$\mathcal{B}(\langle S, o\rangle) = \begin{cases} \{\langle T, o+p\rangle \mid \langle T, p\rangle \in \beta(S)\} & \text{if} \quad S \in I_G \\ \{\langle S, o\rangle\} & \text{otherwise} \end{cases}$$

We initialize β by adding actual-to-formal binding for the base UIVs, i.e. parameters and global variables. Then we proceed to augment β and M_F so that the following constraints are met:

- The bindings to callee UIVs must be consistent with the relative positions of the UIVs. That is, for two UIVs $U_1, U_2 \in I_G$, the following must be true:
  If $U_2 = U_1 @ o_1$ **and** $\langle T_1, p_1\rangle \in \beta(U_1)$ **and** $\langle T_2, p_2\rangle \in \mathcal{M}_F(\langle T_1, p_1 + o_1\rangle)$, **then** $\langle T_2, p_2\rangle \in \beta(U_2)$.
- Points-to arcs in the callee must translate to corresponding points-to arcs in the caller. That is, for every two abstract addresses $\langle S_1, o_1\rangle$ and $\langle S_2, o_2\rangle$ in the callee, the following must be true:
  If $\langle S_2, o_2\rangle \in \mathcal{M}_G(\langle S_1, o_1\rangle)$ **and** $\langle T_1, p_1\rangle \in \mathcal{B}(\langle S_1, o_1\rangle)$ **and** $\langle T_2, p_2\rangle \in \mathcal{B}(\langle S_2, o_2\rangle)$, **then** $\langle T_2, p_2\rangle \in \mathcal{M}_F(\langle T_1, p_1\rangle)$.

We ensure that β and $\mathcal{M}_F$ obey these constraints by augmenting β and $\mathcal{M}_F$ through simultaneous iteration, rather than first computing β and then proceeding to update $\mathcal{M}_F$. This is necessary in order to handle aliases between UIVs, as can be seen in the example in Figure 7. In this example the two parameters of `g` have the same value. This means that instruction 2, referencing the address $\langle[\texttt{P0}], 0\rangle$, may load the address of global variable `B`, which was stored at $\langle[\texttt{P0}], 0\rangle$ by instruction 1. Therefore after the call to `g` in `f`, global variable `C` may point to either `A` or `B`. The translation process goes as follows: At first $\langle\texttt{LV}, 0\rangle$ is bound to both $[\texttt{P0}]$ and $[\texttt{P1}]$, and $\langle\texttt{A}, 0\rangle$ is bound to $[\texttt{P1}]@0$, marked as (1) in Figure 7. These mappings will trigger the addition of the points-to arcs $\langle\texttt{LV}, 0\rangle \rightarrow \langle\texttt{B}, 0\rangle$ and $\langle\texttt{C}, 0\rangle \rightarrow \langle\texttt{A}, 0\rangle$ to $\mathcal{M}_{\mathcal{F}}$ (2). If we reexamine the bindings after these changes to $\mathcal{M}_F$, $\langle\texttt{B}, 0\rangle$ will be now be bound to the UIV $[\texttt{P0}]@0$ (3). Only now can the points-to arc $\langle\texttt{C}, 0\rangle \rightarrow \langle\texttt{B}, 0\rangle$ be added to $\mathcal{M}_F$ (4). Since aliases among UIVs are rarely encountered in practice, the iterative computation of β and $\mathcal{M}_F$ terminates quickly.

Note that the algorithms in [6] and [4] handle aliases between UIVs in significantly more complicated ways. We can handle aliases in the way described above because of our algorithm's flow insensitivity with respect to memory and because we do not perform strong updates.

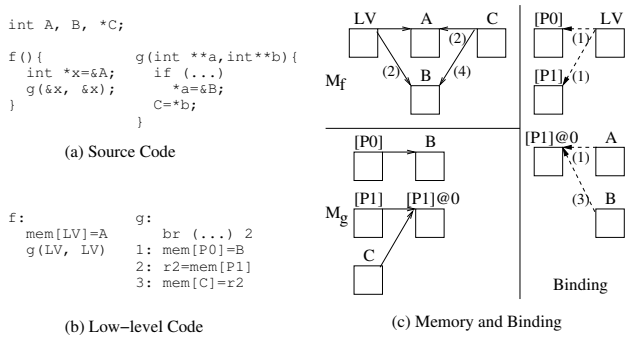

Figure 7. Example of interprocedural analysis in the presence of aliases

4 Information Lost in IR Translation

One would expect that a low-level pointer analysis algorithm would be less accurate than a high-level one, due to information lost during IR lowering, such as type information and high-level semantics. For the C programming language, we find only one specific piece of lost information to be relevant to pointer analysis. This is the start location and the length of statically allocated arrays, which include arrays statically allocated on the stack and arrays that are fields within structures. This information is specified by array declarations and is typically not propagated during lowering.

In some cases, the location and the length of a statically allocated array are necessary to distinguish accesses to the

array from accesses to other data in the same memory region. The example in Figure 8 illustrates this problem. The local variables of procedure `foo` include an array `buf` of 50 bytes and a scalar variable `x`. Figure 8(c) shows one possible layout of the local-variable (LV) section of `foo`, in which `x` is placed above `buf`. Now consider instructions `s1` and `s2` in Figure 8(b): the former accesses `buf` in a loop, whereas the latter accesses `x`. A high-level memory analysis can easily conclude that there is no memory dependence between these two instructions, since they access different local variables.

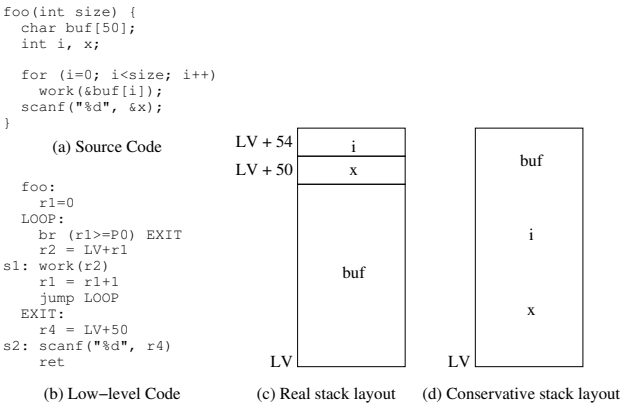

Figure 8. A C program, the corresponding low-level, unannotated code, and its real and conservative memory layouts.

On the other hand, low-level pointer analysis can determine the stride of `s1`'s accesses, but not its bounds. Therefore it has to assume that `s1` may access the whole LV memory area, including the position allocated to `x`. This is illustrated in Figure 8(d), where boundaries between the stack variables are not discernible. This will result in a spurious memory dependence between instructions `s1` and `s2`. Although sophisticated (and quite expensive) numeric analysis can occasionally determine the bounds of an array access, the problem is generally undecidable. In general, a low-level memory analysis algorithm will be more conservative than a high-level one due to array accesses.

Though our experimental results show that the impact of this accuracy loss is negligible in practice, we propose an inexpensive way to remedy it. Specifically, during the translation from a high-level to a low-level IR, load and store instructions that access array variables can be annotated with the array's bounds. For example, this would allow a low-level points-to analysis to properly determine the independence of accesses to `buf` in Figure 8 from accesses to other stack variables.

5 High Level vs. Low Level

The VLLPA algorithm from Section 3 is implemented in the VELOCITY compiler, which is written in Java and compiled and run using Java version 1.5 from Sun MicroSystems. The VELOCITY compiler uses an IR similar to IMPACT's low-level IR, known as Lcode. This section evaluates the compile-time cost and accuracy of VLLPA with respect to the algorithm implemented in the IMPACT [17] compiler. The evaluation was performed on several benchmarks from the SPEC95, SPEC2000 CINT, and Mediabench benchmark suites. Each benchmark was first compiled to the Lcode IR by the IMPACT compiler's front end and then imported into the VELOCITY compiler.

5.1 Performance Evaluation

Benchmark	# Procs	# Opers	# Indirect Calls	Time (s) VLLPA	Time (s) IMPACT
epicdec	34	3998	0	0.770	0.116
g721dec	26	2396	1	0.035	0.150
g721enc	26	2395	1	0.036	0.091
gsmdec	94	11869	6	0.129	0.645
gsmenc	94	11869	6	0.146	0.472
mpeg2dec	114	10223	0	2.150	0.537
adpcmenc	3	288	0	0.071	0.061
adpcmdec	3	284	0	0.055	0.030
rasta	436	42500	7	3.880	2.428
099.go	372	55879	0	2.087	1.765
124.m88ksim	239	26663	3	4.584	1.357
129.compress	18	1211	0	0.268	0.0759
130.li	357	11953	4	14.843	73.340
132.ijpeg	473	33780	644	2.484	13.899
164.gzip	62	7346	2	0.764	0.339
175.vpr	255	25111	0	1.328	1.743
176.gcc	2220	463462	197	1495.318	1706.950
181.mcf	24	2157	0	0.285	0.1383
186.crafty	110	41370	0	1.543	0.694
197.parser	324	22686	0	2.835	3.388
254.gap	854	145017	1281	643.734	950.64
255.vortex	923	91864	15	12.107	42.330
256.bzip2	63	6725	0	0.485	0.2746
300.twolf	167	53950	0	1.567	1.136

Table 2. Benchmark Statistics

Table 2 shows the time taken by VLLPA on a 3GHz P4 with 512KB cache and 2GB of memory running RedHat 9.0. Note that this time is only the time needed to perform the analysis and does not include the time to read or write the IR. Analysis time, ranging from $0.035sec$ to $24min$, is largely a function of the number of indirect call sites, the size of SCCs, and the number of operations. For comparison, the time taken by IMPACT's source-level pointer analysis, written in C and compiled using GCC on the same machine, is also shown. VLLPA generally takes less time than IMPACT's analysis, in some cases significantly. This appears to be due to implementation deficiencies in IMPACT's analysis. In fact, due to these deficiencies, the current implementation of IMPACT's analysis cannot handle

253.perlbmk and produces incorrect dependence arcs for *176.gcc*. As the analysis times for VLLPA show, a low-level optimizer may apply our analysis multiple times at chosen points without suffering prohibitive compile-time costs.

5.2 Accuracy Evaluation

Benchmark	# Opers	VLLPA Arcs		Arcs w/ Array Info	
	w/ Arcs	More	Fewer	More	Fewer
099.go	13232	0	2393	0	2393
124.m88ksim	7161	4	2722	4	2722
129.compress	329	0	45	0	51
130.li	3762	33	720	33	724
164.gzip	1953	3	847	3	848
175.vpr	8166	97	1397	97	1402
181.mcf	705	7	199	7	199
186.crafty	12026	17	4751	17	4759
256.bzip2	1535	3	255	3	256

Table 3. Analysis precision measured on dependence arcs vs. IMPACT (fewer arcs imply better precision).

The accuracy comparison is made in terms of how effectively each analysis can disambiguate memory operations. Both the VELOCITY compiler and IMPACT use the results of memory analysis to compute dependence arcs between load/store and store/store pairs. In general, a *more* accurate analysis algorithm will result in *fewer* dependence arcs.

Rather than show the results for all 20+ benchmarks from Table 2, we now focus on a representative set of benchmarks from SPEC95 and SPEC2000 that avoid the implementation deficiencies of IMPACT's analysis. The "VLLPA Arcs" columns in Table 3 show the number of memory operations for which our analysis results in more and fewer dependence arcs (i.e. less and more precision respectively) than [2]. On average, our analysis is less precise than [2] on 0.3% of memory operations, and more precise on 26.8% of memory operations. This indicates that VLLPA does just as well or better than a sophisticated high-level analysis.

The "Arcs with Array Info" column in Table 3 shows the precision for each benchmark with VLLPA modified to utilize the annotated array bound information, as discussed in Section 4. As the results show, knowledge of array boundaries is not significantly important in practice, resulting in no decrease in the number of memory operations for which VLLPA produces less precise information than IMPACT on any of the benchmarks tried.

6 The Conservative Dependence Propagation Problem

This section illustrates the problem that arises from preforming alias analysis at the high level. Specifically, dependence arcs created by the high-level analysis must be propagated by subsequent code transformations. Without knowledge of how to perform additional memory disambiguation at the low level, this propagation has to be performed conservatively, resulting in many spurious dependences. These unnecessary dependence arcs may in turn limit the aggressiveness of later optimizations. Instruction scheduling can be particularly hurt by this, since the extra dependence arcs limit its ability to reorder memory operations. The effects of spurious dependence arcs on performance will be studied in Section 7.

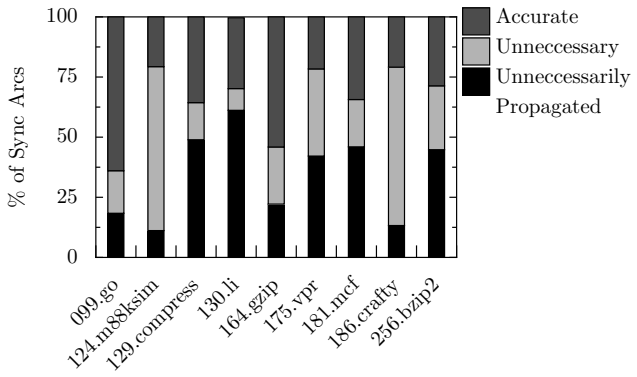

Figure 9. Dependence Arc Breakdown

In order to determine exactly how many spurious dependence arcs are created due to the conservative propagation of memory analysis information, the following experiment was performed. We ran VLLPA on the IR produced by IMPACT at the final stage of compilation, after both high- and low-level optimization has been performed. We then compared the dependence arcs produced by our analysis with those propagated by IMPACT from the high level. We generally expect our algorithm to produce fewer dependence arcs, both because it does not suffer from the effects of conservative dependence propagation, and because it is generally more accurate than IMPACT's high-level algorithm, as seen in Section 5. To separate the two effects, we also ran our algorithm on IMPACT's IR right after lowering and marked the dependence arcs that were generated by IMPACT but not by our algorithm. Subsequent code transformations in IMPACT were modified to propagate these marks along with the dependence arcs. The dependence arcs present in IMPACT's final stage were then classified in three categories, as seen in Figure 9.

Correct: The propagated arcs that coincide with arcs identified by our analysis.

Unnecessary: Spurious arcs that result from the propagation of spurious high-level arcs. These arcs arise due to the differences between the two memory analysis algorithms, rather than due to conservative propagation.

Thus they are not relevant to this experiment.

Unnecessarily Propagated The arcs that are spurious at the final stage, but are propagated from accurate high-level arcs. These are the arcs introduced by inaccuracies due to conservative dependence propagation.

As shown in Figure 9, this final category of arcs accounts for up to 50% of the arcs present at the final compilation stage. This shows that repeating memory analysis at the low level is significantly more accurate than propagating memory analysis information for the higher level. Conversely, rerunning memory analysis at the low level would result in 37% to 79% fewer dependence arcs than IMPACT's current dependence propagation scheme. In other words, this experiment shows that a low-level memory analysis algorithm is a very useful tool in an optimizing compiler.

7 Impact of Different Memory Disambiguation Schemes on Performance

Since the ultimate goal of pointer analysis is to facilitate optimizations, this section evaluates the effect of pointer analysis on performance. Specifically, we compare, for Itanium 2, the cycle counts computed from profile-weighted static instruction schedules of executables compiled under three different configurations. The baseline configuration uses IMPACT with simplistic pointer analysis, similar to that run by GCC. Only intraprocedural analysis is performed with simple instruction inspection, which includes differentiating loads and stores with the same base register but different offsets, or references to global vs. local structures. The second configuration applies IMPACT's memory analysis algorithm at the high level and then propagates memory dependence information through subsequent compiler phases. The third configuration recalculates memory dependences by running VLLPA after most optimizations are performed, but right before instruction scheduling and machine-specific optimizations. The resulting performance is shown in Figure 10.

As we can see, performing high-level pointer analysis significantly improves performance, with an average speedup of 8.3%. The more accurate pointer analysis provided by VLLPA increases the average speedup to 10.9%. This is because our algorithm removes many spurious memory dependences, as seen in Section 5, thus giving the scheduler more freedom to rearrange instructions.

Since IMPACT's scheduler moves instructions only within extended basic blocks (EBBs), spurious memory dependences between EBBs (or the lack thereof) will not affect performance. Table 4 classifies dependence arcs according to whether they link instructions within the same EBB (intra-EBB columns) or not (inter-EBB columns). The

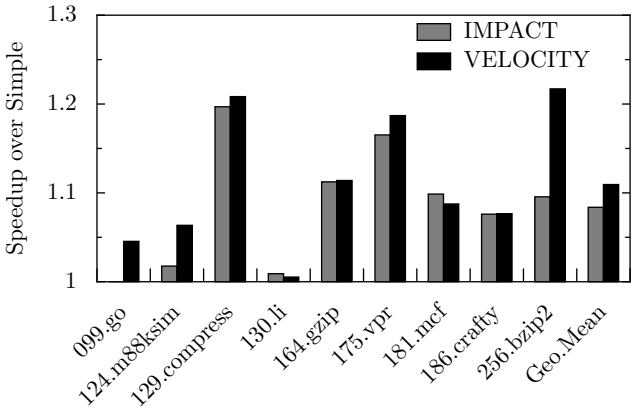

Figure 10. Speedup vs. Simplistic

columns labeled "Extra Arcs" refer to dependence arcs produced by VLLPA but not IMPACT, whereas the columns labeled "Removed Arcs" refer to dependence arcs produced by IMPACT but not by VLLPA. As we can see from this table, the vast majority of spurious dependences removed by VLLPA are inter-EBB, and thus of no use to the scheduler. This is why the big differences in the number of memory dependences shown in Section 5 translate to relatively small performance differences for certain benchmarks. However, inter-EBB dependence arcs can affect other code motion optimizations, as well as global schedulers, which are used by other aggressively optimizing compilers.

8 Conclusion and Future Work

Pointer analysis provides memory dependence information crucial for compiler optimizations. Currently, the dominant approach to pointer analysis is to apply it on a high-level IR that retains source language information. This approach is believed to be more accurate than a low-level pointer analysis, which is thought to be hampered by the loss of high-level semantics.

This paper is the first to provide a context-sensitive and partially flow-sensitive pointer analysis that operates on a low-level IR, and to evaluate the relative effectiveness of high-level versus low-level memory analysis in an aggressively optimizing compiler. Our results show that, for a C compiler, a low-level memory analysis algorithm can be as good as or better than a high-level one. The inaccuracy caused by the loss of information when translating from source code to a low-level IR is characterized, and found to have a negligible impact in practice. Moreover, the availability of low-level pointer analysis eliminates the inaccuracies caused by the conservative propagation of memory dependences through subsequent compiler phases, leading to more efficient back-end optimizations. Thus this work es-

Benchmark	Extra Arcs	% Intra-EBB	% Inter-EBB	Removed Arcs	% Intra-EBB	% Inter-EBB
099.go	0	0.00	0.00	10799	0.74	99.26
124.m88ksim	7	0.00	100.00	42685	2.69	97.31
129.compress	0	0.00	0.00	110	0.00	100.00
130.li	65	73.85	26.15	13827	29.00	71.00
164.gzip	4	0.00	100.00	3535	0.85	99.15
175.vpr	155	7.09	92.81	9764	8.37	91.63
181.mcf	8	25.0	75.00	388	0.00	100.00
186.crafty	121	0.00	100.00	125684	11.0	89.00
256.bzip2	4	0.00	100.00	2256	1.42	98.56

Table 4. Properties of mismatching sync arcs

tablishes low-level pointer analysis as a feasible and worthwhile memory disambiguation scheme.

9 Acknowledgments

We first thank Peng Wu, Michael Hind, and others at IBM for discussions that helped refine the goal of this work in its early stages. We also thank the other members of the Liberty Research Group for their support and ideas. This work has been supported by the National Science Foundation (CAREER CCF-0133712). Opinions, findings, conclusions, and recommendations expressed throughout this work are not necessarily the views of the National Science Foundation.

References

[1] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The superblock: An effective technique for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, pp. 229–248, January 1993.

[2] B.-C. Cheng and W. W. Hwu, "Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 57–69, 2000.

[3] M. Hind, "Pointer analysis: Haven't we solved this problem yet?," in *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, 2001.

[4] R. P. Wilson and M. S. Lam, "Efficient context-sensitive pointer analysis for C programs," in *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pp. 1–12, June 1995.

[5] B. Steensgaard, "Points-to analysis by type inference in programs with structures and unions," in *Lecture Notes in Computer Science, 1060* (T. Gyimothy, ed.), pp. 136–150, Springer-Verlag, 1996. Proceedings from the International Conference on Compiler Construction.

[6] R. Chatterjee, B. G. Ryder, and W. A. Landi, "Relevant context inference," in *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 133–146, January 1999.

[7] S. Debray, R. Muth, and M. Weippert, "Alias analysis of executable code," in *Proceedings of the ACM Symposium on Principles of Programming Languages*, January 1998.

[8] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in *Proceedings of the 13th International Conference on Compiler Construction*, pp. 5–23, 2004.

[9] B. Steensgaard, "Points-to analysis in almost linear time," in *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 32–41, January 1996.

[10] W. Landi and B. G. Ryder, "A safe approximate algorithm for interprocedural pointer aliasing," in *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 235–248, June 1992.

[11] L. O. Andersen, "Program analysis and specialization for the C programming language," May 1994.

[12] M. Emami, R. Ghiya, and L. J. Hendren, "Context-sensitive interprocedural points-to analysis in the presence of function pointers," in *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pp. 242–256, June 1994.

[13] A. Deutsch, "A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations," in *Proceedings of the 1992 International Conference on Computer Languages*, pp. 2–13, April 1992.

[14] M. Sagiv, T. Reps, and R.Wilhelm, "Solving shape-analysis problems in languages with destructive updating," in *Proceedings of the ACM 23rd SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo '00)*, January 1996.

[15] M. Hind and A. Pioli, "Which pointer analysis should I use?," in *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis*, August 2000.

[16] A. Deutsch, "Interprocedural may-alias analysis for pointers: Beyond k-limiting," in *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pp. 230–241, June 1994.

[17] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors," in *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 266–275, May 1991.