Speculative Alias Analysis for Executable Code

Manel Fernández and Roger Espasa

Computer Architecture Department
Universitat Politècnica de Catalunya
C/ Jordi Girona 1–3, 08034 Barcelona, Spain
E-mail: {mfernand,roger}@ac.upc.es

Abstract

Optimizations performed at link time or directly applied to final program executables have received increased attention in recent years. Such low-level optimizations can benefit greatly from pointer alias information. However, as almost all existing alias analyses are formulated in terms of source language constructs, they turn out to be of limited utility at the machine code level.

This paper describes two different approaches to high-quality, low-cost, speculative may-alias analysis*, to be applied in the context of link-time or executable code optimizers. The key idea behind our proposals is the introduction of* unsafe speculations *at analysis-time, which increases alias precision on important portions of code, and keeps the analysis reasonably cost-efficient. Experimental results indicate that introducing speculation at analysis-time is clearly beneficial: precision increases up to 83% in average, against a baseline precision of 16%. Furthermore, the percentage of dynamic misspeculations is typically about 2%, which shows that our technique can be used even for scenarios where speculation recovery is expensive.*

1. Introduction

Code transformations on executable code can benefit greatly from pointer alias information, as already happens with the compilation of source-level programs. For instance, whole program optimizations may open up opportunities for moving invariant memory instructions out of loops. However, alias information is key to identifying such opportunities. While there is an extensive body of work on pointer alias analysis of various kinds [30, 27, 12, 8, 16], these are mostly high level analyses carried out in terms of source language constructs. Unfortunately, such analyses turn out to be of limited utility at the machine code level. In fact, the problem of memory disambiguation is one of the weak points of object code modification, because high level information available in a traditional compiler is lost. Furthermore, features such as pointer arithmetic and out-of-bounds array accesses must be handled at this level, where the contents of every register is potentially an address.

This paper presents two approaches to high-quality, low-cost, speculative alias analysis, to be applied in the context of link-time or executable code optimizers. The key idea behind our two proposals is to trade off analysis complexity against *safeness*. Our alias analysis incorporate in their dataflow equations the notion of "guessing" when two memory references are *most likely* independent. By being more liberal in the propagation of the dataflow information, we obtain two alias analyses that correctly disambiguate a lot more often, yet the analyses may sometimes be wrong. Our results will show that these techniques achieve high levels of accuracy: over 80% of all disambiguation queries are answered with a response different than "unknown". Furthermore, the dynamic number of instances where the disambiguator was wrong, and, consequently, recovery action would have been invoked, is sufficiently low (less than 2% over all queries) to render our proposed analysis useful even for optimizations with high-cost recovery schemes.

The rest of this paper is organized as follows. In Section 2 we introduce the state of the art on low-level alias analysis algorithms. Section 3 describes in detail our different contributions in order to speculatively increase alias analysis accuracy for binary code. Next, Section 4 describes the methodology we used, and compares the effectiveness of each approach. We discuss related work in Section 5, and finally Section 6 concludes the paper.

2. Background

The problem of alias analysis or *memory disambiguation* at the machine code level is to determine the relationship of every pair of memory references in a program. A *reference* typically identifies a memory address and an access size. Then, for two particular references, there are three possible answers that an alias disambiguator can return:

- *Identical*, or *intersecting*. This means that both references always point to the same location, or that memory accessed by both references partially overlaps.
- *Disjoint*, which means that they are never aliased, and therefore, independent.
- *Unknown*. That is, the disambiguator cannot determine statically the relationship between the two references.

Performing *no alias analysis* leads to the assumption that every load and store instruction are always dependent on every previous store instruction. In general, the aliasing problem can be formulated by a combination of *may-alias* analysis, which answers whether two memory references are independent, and *must-alias* analysis, which checks references for dependencies. This work is about *may*-alias analysis algorithms for executable code.

In the next sections, we assume for simplicity a canonical RISC instruction set. Memory is accessed only through explicit load and store instructions, which have the form `load` $k(r_b), r_a$ and `store` $r_a, k(r_b)$. Memory instructions have the effect of reading/writing from/to the location whose address is $k + contents(r_b)$, where k is a constant offset and r_b is the base register. Two special registers, denoted as *sp* or *stack pointer*, and *gp* or *global pointer*, point to the program stack and global data areas, respectively. For other instructions we assume the form *op* src_1, src_2, dst, where *op* denotes an operation, dst is a destination register, and src_1 and src_2 are source registers[1]. For instance, an `add` instruction computes the sum of src_1 and src_2 into dst. Many other operations can be expressed in terms of this form (e.g., register moves can be modeled in terms of addition). Other additional instructions such as conditional and unconditional jumps have the only effect of determining the control flow graph of the program, so they are not considered explicitly in the context of alias analysis. We also ignore operations on floating point registers, assuming that such operations are not used for address computation.

2.1. Alias analysis by instruction inspection

A common technique in compile-time instruction schedulers is *alias analysis by inspection* [21]. Here, two memory references are considered within an extended basic block to see if it is obvious that they point to different memory addresses. Independence between instructions I_1 and I_2 can be proved if either of the following conditions hold:

- Different memory regions are referenced. For example, one of the instructions uses a register known to point to the stack and the other uses a register known to point to the global data area.

[1]To simplify the discussion we abuse notation and allow either src_1 or src_2 to be an integer constant, denoting an immediate operand.

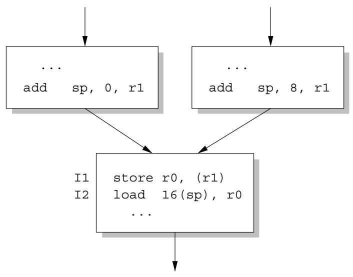


Figure 1. Sample code where different definitions are reaching a use.

- They access data at addresses $k_1(r_1)$ and $k_2(r_2)$, where base registers r_1 and r_2 are computed by two (possibly empty) sequences of instructions such that $r_1 = c_1 + contents(r_0)$ and $r_2 = c_2 + contents(r_0)$, for a register r_0. Accesses are non-aliased if chains use the same definition of r_0, and $c_1 + k_1$ and $c_2 + k_2$ do not overlap. To detect definition of register r_0 a simple backwards dataflow algorithm may be used.

All other memory instruction pairs are considered to be aliasing. Unfortunately, this simple approach does not work if information about register copies and address arithmetic needs to be propagated across extended basic block boundaries. To do so, *register use-def chains* is a well known analysis that provides, for each use of a register, a pointer to its definition [21]. In the general case, an instruction inspection algorithm tries to derive a symbolic description for each memory instruction and then compare these descriptions for checking independency.

The use-def chains are a directed graph whose nodes are instructions and whose edges are use-def pointers. When there are several definitions of a register reaching a use, it is common to introduce a pseudo instruction at an appropriate place which also defines that register, thereby shadowing the other definitions. This is analogous to ϕ functions used with the static single assignment (SSA) form [21]. Pseudo insertions result in “less accurate” use-def information. However, the analysis becomes space-feasible on executable code, because fully linked executables tend to be considerably larger than source languages modules [22].

2.2. Residue-based global alias analysis

Instruction inspection fails when several definitions are reaching a use. For example, in Figure 1, register r_1 is defined with two possible stack values. However, possible locations accessed at instruction I_1 are disjoint with respect to the location accessed by instruction I_2.

Debray *et al.* [11] propose a combination algorithm based on the analysis described in the previous section, and a novel low-level interprocedural approach, to reason about pointer aliases in executable code. The analysis, which is implemented in the context of a link time optimizer, can handle complex pointer arithmetic and features usually ignored by traditional alias analysis algorithms. The ideas described in this paper were motivated by their work.

An alias analysis will in general associate each register with a set of possible addresses at each program point. The idea of the algorithm is to reason about arithmetic computations modulo some pre-selected value k. A set of addresses is represented by an *address descriptor*, which is a pair $\langle I, S\rangle$, where I is the *defining instruction* for a machine register r, and S is a set of mod-k residues with respect to the value computed by I. The representation can then distinguish between addresses involving distinct "small" displacements (i.e., less than k) from a base register. Comparing descriptors is reduced to a comparison of mod-k sets.

Since k is fixed, S can be represented as a bit vector of length k. Their implementation corresponds to mod-k residues with $k = 64$, in part determined by the fact that the set of mod-k residues for this choice of k corresponds to a bit vector that fits exactly in one 64-bit machine word. This means that set operations such as union, intersection, checking containment, etc., are compactly representable and can be carried out in $O(1)$, which is cheap enough to be practical for the analysis of large binaries.

As far as the analysis is concerned, a data flow system is used to propagate values through the control flow graph. They use a widening operation to "merge" the information coming along the incoming edges at vertices in the interprocedural control flow graph. Thus, if the values for a register r being propagated along two incoming edges at a vertex in the flow graph are described by address descriptors $\langle I_1, S_1\rangle$ and $\langle I_2, S_2\rangle$ respectively, and $I_1 \neq I_2$, then the information about r is widened to $\perp$, denoting a lack of information. The essential idea behind this operation is to associate a single descriptor with a register at each program point, rather than a set of descriptors, keeping the memory requirements of the analysis reasonable: for each basic block one address descriptor per register is needed, corresponding to the *out* register set at the exit of the block. For a given choice of k, the analysis requires $RN(k + w)$ bits of memory for a program with N basic blocks on a machine with R registers, where w is the number of bits per machine word.

3. Speculative *may*-alias analysis

Residue-based alias analysis fails in several situations, leading to an undesirable loss in precision. First, by using mod-k residues, the algorithm is clearly oriented to "fine grain" disambiguation, but it is unable to effectively catch "coarse grain" alias relationships (e.g., whether two references point to different memory regions). Second, in order to keep the algorithm space- and time-feasible, the conservative widening operation causes information to be lost when joining definitions of the same register from different control flow paths. This is specially negative for pointer arguments at the entry node of functions with multiple call sites, since the context-insensitive nature of the algorithm leads to a massive application of the widening operation. Furthermore, the algorithm does not keep track of the contents of memory, which causes information to be lost when registers are saved/restored.

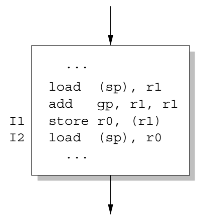
Typically, complexity of dataflow analyses has been a compromise between *cost* and *precision* (see, for example, [1, 21, 5, 13, 26]). For high level compilation, compiler writers have tended to use sophisticated analysis at the expense of increased resource usage. However, given that statically-linked executable programs tend to be significantly larger than the corresponding source level entities, traditional analyses applied to machine code level are of limited usefulness, because either the cost is too high or the precision is not accurate enough.

The key idea behind our work is to introduce a new variable in the game: *safeness*. Breaking the strong constraint of safeness, a dataflow analysis may reach a high level of precision at low cost, by paying the price of not always being correct. In other words, the dataflow analysis becomes *speculative*, or *unsafe*. As far as we know, this is the first attempt to systematically introduce unsafe speculations into dataflow analysis algorithms.

In the following sections we introduce two dataflow algorithms that increase alias analysis accuracy of binary code using speculation. The two algorithms are orthogonal and, thus, can be applied independently or coupled together. The first algorithm tries to disambiguate memory references by classifying them into separate memory regions (heap, stack and global) and assumes that whenever an arithmetic operation is performed on a pointer, the pointer will not change its pointed-to memory region (of course, this is an unsafe speculation). The second algorithm uses profile information and assumes that memory instructions on hot paths are not aliased to memory references on cold paths (again, another unsafe speculation). By making these speculations, we obtain more precise information in the common case, yet the analysis results are not always correct. This means that any optimization performed using these speculative analyses will be speculative as well, and some type of recovery mechanism must be provided. We will extend this discussion in Section 3.4.

3.1. Region-based speculative alias analysis

Figure 2 shows a situation where the analysis presented in Section 2.2 is not accurate enough. As it can be seen, the value of register r_1 is defined by a load instruction.



```
         ...
     load  (sp), r1
     add   gp, r1, r1
I1   store r0, (r1)
I2   load  (sp), r0
         ...
```

Figure 2. Sample code where pointer information is lost because a component of the *use-def* chain of r_1 is defined by a load operation.

Therefore, the analysis in unable to propagate information through the uses of r_1. Yet, it is very *unlikely* the register r_1 at instruction I_1 points to regions other than the global data area because the value loaded from memory was operated previously with the *gp* register. Thus, the safeness of the residue-based alias analysis is missing a very likely opportunity of disambiguating the two references. Note that this loss of precision would also happen if the descriptor of r_1 had been mapped to $\perp$ because of the widening operation.

To overcome this drawback, our first proposal is to propagate which *memory regions* a register may point to, instead of being worried about symbolic descriptors based on instructions which defined that register.

Definition 3.1 *A* region descriptor α_r^p *is a subset of the finite set of values* $\{\mathcal{G}, \mathcal{S}, \mathcal{H}\}$*, denoting all possible memory regions (i.e.,* global*,* stack *and* heap*, respectively) pointed by a given register r at program point p.* ■

For a particular region descriptor, a value of $\emptyset$ denotes that register is not used as a pointer to any memory region, and is written as $\top$; while a value of $\{\mathcal{G}, \mathcal{S}, \mathcal{H}\}$ denotes a total lack of information, and is written as $\perp$. This information will be then propagated using a general dataflow iterative algorithm. The input values, for every register r at every program point p, are initialized as follows:

$$\alpha_r^p = \begin{cases} \{\mathcal{G}\} & \text{if } r = gp \text{ (global pointer)} \\ \{\mathcal{S}\} & \text{if } r = sp \text{ (stack pointer)} \\ \top & \text{otherwise} \end{cases}$$

The value $\{\mathcal{H}\}$ denoting a pointer to the *heap* memory area is assigned to the destination register of the system call *break*, at such program point. System call *break* is used by Unix-based operating systems for heap management, through library functions `malloc` and `free`.

When propagating information through the control flow graph, the effect of instructions on its corresponding destination register may vary. For example, a load instruction sets the descriptor of its destination register to $\perp$, since no information about the contents of memory cells is kept in the algorithm. The general behavior is as follows:

Definition 3.2 *Let* $op^p\ r_i, r_j, r_k$ *be an instruction at program point p, with source registers* r_i, r_j *and destination register* r_k*. Let* α_i^p *and* β_j^p *be the region descriptors for registers* r_i *and* r_j*, respectively. Then, the region descriptor* γ_k^{p+1} *for register* r_k *at program point* $p+1$*, is set to* $\alpha_i^p \bullet \beta_j^p$*, defining the* $\bullet$ *operator as:*

$$\alpha_i^p \bullet \beta_j^p = \begin{cases} \alpha_i^p & \text{if } \alpha_i^p \neq \top \wedge \beta_j^p = \perp \\ \beta_j^p & \text{if } \beta_j^p \neq \top \wedge \alpha_i^p = \perp \\ \alpha_i^p \cup \beta_j^p & \text{otherwise} \end{cases}$$

■

In the case that a source operand is a constant instead of a register, the corresponding region descriptor is assumed to be $\top$. Note that a region descriptor with a value different than $\top$ being operated with a region value of $\perp$, will propagate the non-$\top$ descriptor to the instruction destination register. Strictly, it is *unsafe* to make such an assumption, although the opposite rarely occurs. For instance, a C code might produce from a pointer to the global data area, a pointer to the program stack. However, it is uncommon for many programs to generate these types of accesses[2].

If a node of the control flow graph has more than one predecessor, the information stemming from them must be integrated. In dataflow frameworks, joining paths in the flow graph is implemented by the union operator.

Definition 3.3 *Let* α_r^p *and* β_r^p *be region descriptors for register r at program point p, coming from two different predecessors. Then, the widening operation* $\bigtriangledown$ *is defined as:*

$$\alpha_r^p \bigtriangledown \beta_r^p = \begin{cases} \perp & \text{if } \alpha_r^p = \perp \vee \beta_r^p = \perp \\ \alpha_r^p \cup \beta_r^p & \text{otherwise} \end{cases}$$

■

Note that region-based widening operation differs from residue-based widening operation in that "widening" does not imply loosing all information, but performs simply a union between region descriptors.

A region descriptor may be represented as a 3-bit vector, where every bit denotes one of the considered memory regions. As a result, the resulting analysis requires only $3RN$ bits of memory for a program with N basic blocks on a machine with R registers. The *region-based alias analysis* presented here is complementary to any *points-to* alias scheme, and may be coupled to it, or may be computed separately.

3.2. Profile-guided speculative alias analysis

When trying to keep the residue-based alias analysis algorithm space- and time-feasible, the conservative widen-

[2]We do not found such scenarios in our benchmark suite, but operating system kernels and tools, as well as virtual machines are programs where this situation might happen. Certainly, our proposed analysis could be run on user-demand by some compiler command line option, like actual production compilers do on several unsafe optimizations.

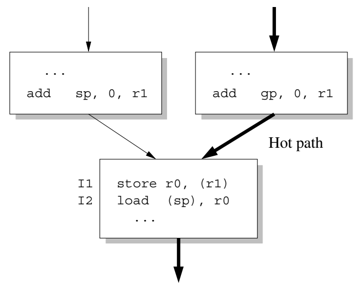


Figure 3. Sample code where different definitions are reaching a use, but there is a more likely executed path.

ing operation does not join definitions of the same register from different control flow paths in a set. Because of this, when computing the meet of the incoming information at the entry of the basic block, the information associated with the register is widened to $\perp$. That is, all the information is lost. An example of this can be seen in Figure 3, where register r_1 is defined from two different instructions. In the example, region-based alias analysis does not solve the problem either, since joining regions, although less conservative, also fails in this case.

The preceding situation also occurs on pointer arguments at the entry node of functions. A possible solution would be to use a context-sensitive interprocedural approach. However, the defining instructions for a register are generally different at different call sites to a function, which means that the callee will have to be analyzed separately for each such call site. Given that statically-linked executable programs tend to be significantly larger than the corresponding source level entities, this indicates that the cost of a traditional context-sensitive analysis is likely to be quite high.

We chose, instead, to use a profile-guided analysis. The basic idea is to propagate alias information only for important paths, ignoring those paths whose information will cause loss of precision in the most common cases. As a result, widening operations to $\perp$ will be drastically reduced. For instance, looking again at Figure 3, we can see that the most likely definition of register r_1 is the one from register *gp*. We could then easily determine by a single inspection that accesses at instructions I_1 and I_2 are likely to be disjoint. More formally:

Definition 3.4 *Let* $\alpha_r^{p,1}, \ldots, \alpha_r^{p,n}$ *and* $\beta_r^{p,1}, \ldots, \beta_r^{p,m}$ *be the set of symbolic descriptors for register* r *at program point* p*, coming from* hot *predecessors* $1, \ldots, n$ *and* cold *predecessors* $1, \ldots, m$ *respectively; and let* $\bigtriangledown$ *be a widening operator, the new widening operation* $\bigtriangledown_s$ *is defined as:*

$$\bigtriangledown_s \left(\alpha_r^{p,1}, \ldots, \alpha_r^{p,n}, \beta_r^{p,1}, \ldots, \beta_r^{p,m}\right) = \bigtriangledown \left(\alpha_r^{p,1}, \ldots, \alpha_r^{p,n}\right)$$

■

That is, the algorithm only takes into account the information coming from *hot* edges of the control flow graph. Note that the result of this new meet operation is *speculative* in nature, because we simply "ignore" some possible (although infrequent) paths in the analysis. This will give more precise information in the common case, but the result of the analysis will not always be correct.

This simple speculative dataflow scheme is general enough to be applied on top of any traditional dataflow analysis algorithm, not necessarily related to neither pointer aliasing nor machine code level. On the other hand, the cost of a speculative dataflow technique does not change with respect to the non-speculative safe version which it is based on. However, the intuition indicates that ignoring unimportant paths, significant cost reductions can be achieved.

3.3. Putting it all together

We have implemented the two alias analysis algorithms presented in the previous sections within the Alto link-time optimizer [23]. As a result, we have obtained a high-quality, low-cost, combined speculative alias analysis framework for executable code. Computing alias information, the algorithm uses the following scheme.

Phase 1 An interprocedural dataflow analysis computes the *use-def* chains (Section 2.1). This phase is the only one required for detecting memory dependencies.

Phase 2 The algorithm performs an interprocedural dataflow analysis computing residue-based information (Section 2.2). The resulting analysis is the *safe aliasing information*. At the same time, region-based information is computed (Section 3.1), which produce preliminary unsafe aliasing data.

Phase 3 Finally, Phase 2 is recomputed (i.e., residue-based and region-based analyses) as speculative profile-guided schemes (Section 3.2). Additionally, region-based analysis may now assume that contents of memory cells will not be used as memory pointers (i.e., corresponding descriptors are mapped to $\top$ [14]). The resulting data and region-based information from the previous phase is the *unsafe aliasing information*.

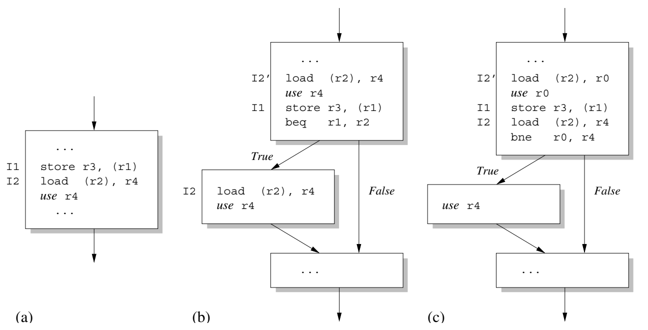
Memory disambiguation for a particular pair of memory references is then applied incrementally, as can be seen in Figure 4. Note that a new relationship is used for those pairs of references which are likely to be disjoint. This new status of *"likely independent"* gives the choice of conscious speculation to any following speculative optimization.

3.4. Recovery-based usage of speculative analysis

The proposals presented in this work increases, at low cost, the precision of the alias analysis by providing more

```
Input: Two memory instructions $I_1, I_2$.
Output: An alias relationship,
    {dependent, independent, likely independent, unknown}.
Method:
    if ud-chains($I_1, I_2$) $\neq$ unknown then
        return ud-chains($I_1, I_2$);
    elsif aliasing($I_1, I_2$,safe) $\neq$ unknown then
        return aliasing($I_1, I_2$,safe);
    elsif aliasing($I_1, I_2$,unsafe) $\neq$ unknown then
        return likely independent;
    else
        return unknown;
    endif
End Method
```

Figure 4. Memory disambiguation scheme, for a pair of memory instructions.



```
    ...
I1  store r3, (r1)
I2  load  (r2), r4
    use r4
    ...
```

```
    ...
I2' load  (r2), r4
    use r4
I1  store r3, (r1)
    beq   r1, r2

I2  load  (r2), r4
    use r4

    ...
```

```
    ...
I2' load  (r2), r0
    use r0
I1  store r3, (r1)
I2  load  (r2), r4
    bne   r0, r4

    use r4

    ...
```

Figure 5. Reordering memory operations [20]: (a) original sample sequence; recovery-based reordering by using (b) *interference test*, **(c)** *coherence test*.

reliable information in the common case. However, the speculative nature of our schemes causes the analysis results to be not always correct. That is, optimization performed using speculative analysis will be speculative as well.

Speculative optimizations [19, 15, 20, 18, 25] have been widely used in the compiler world for reducing the overall execution time of programs. The key idea behind speculation is breaking the original program sequence by executing a (possibly *unsafe*) "better" reordering of instructions, corresponding to the most likely execution paths. Since the new executed sequence may be unsafe, some type of *check-and-recovery* mechanism must be provided for validating/undoing such assumptions at run-time. In this mechanism, "checking" must be cheap enough and "recovery" should be invoked infrequently, in order to not incur into unnecessary penalties. Discussion of speculative optimizations as well as check-and-recovery mechanisms are, however, beyond of the scope of this paper.

In general, speculative alias analysis is particularly well suited to be used in combination with speculative optimizations based on reordering memory operations [19, 15, 20, 25], including those related to the IA-64 architecture [4]. An example of such optimizations can be seen in Figure 5. By using the new status of *"likely independent"*, our speculative disambiguator not only provides information about which instructions are *likely* to be moved, but also which ones are *not recommended* to be involved in code motion.

4. Evaluation

4.1. Experimental framework

We have implemented our proposed alias analysis algorithms within the Alto link-time optimizer [23]. Details about the implementation have been presented in Section 3.3. The information reported here was obtained after several optimization rounds carried out by Alto.

The benchmarks used were the eight programs in the SPEC95 integer benchmark suite. All programs were compiled with full optimizations, using the vendor-supplied C compiler on an AlphaServer 8400. For processing by Alto, the compiler was also invoked with linker options to retain information and to produce statically linked executables[3]. Later, they were instrumented using Pixie and executed on the SPEC training inputs to obtain an execution frequency profile. Finally, these binaries and their profiles were processed by Alto using different degrees of alias analysis, to obtain different measures of their precision.

To apply our profile-guided analysis, we need to determine the set of edges which are the the most frequently executed from the program control flow graph, or *hot edges*. There are some profiling tools that, using edge counting instrumentation, can obtain basic block and edge counts directly. However, it is often the case that only basic block counts are available. Our framework gets program profiling information using Pixie, which only provides block execution counting. Although it is widely known that block counts can be derived form edge counts and the converse does not hold, edges whose counts cannot be determined from block counts are usually fewer than 1%. We use then a variation of the algorithm from Tamches and Miller [29] for deriving edge counts from Pixie profile data.

To determine the set of hot edges, we first specify what it means for a basic block to be considered *hot*. Given a value ϕ in the interval $(0, 1]$, we determine the largest execution frequency threshold N such that, by considering only those basic blocks that have execution frequency at least N, we are able to account for at least a fraction ϕ of the total number of instructions executed by the program (as indicated by

[3]We used statically linked executables because Alto relies on the presence of relocation information for its control flow analysis. The Tru64 Unix linker refuses to retain information for non-statically linked binaries.

Method	Description
Inspect	Corresponds to the disambiguation mechanism by instruction inspection, using *use-def* chains (Section 2.1). This method is the only one required for later detecting memory dependencies.
Residue	Disambiguation based on a residue-based analysis (Section 2.2).
Region	Corresponds to the application of the speculative region-based alias analysis. This is the first unsafe analysis method (Section 3.1).
PG$_{Res}$	Profile-guided speculative residue-based alias analysis (Section 3.2).
PG$_{Reg}$	The profile-guided speculative technique is applied to the also speculative region-based method.
PG$_{Reg'}$	Corresponds to the previous analysis method, but contents of memory cells are assumed to not be used as memory pointers [14].

Table 1. Description of alias analysis methods for memory disambiguation.

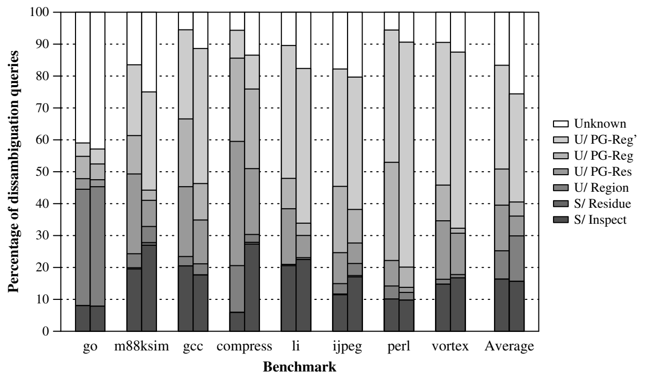


Figure 6. Breakdown of disambiguation queries, by alias analysis method. The left bar considers the full set of queries, while the right bar restricts the set to those queries where both components are instructions from *hot* paths. The *S/* and *U/* prefixes denote *safe* and *unsafe* analysis, respectively.

its basic block execution profile). Any basic block whose execution count is at least N is then said to be *hot* with respect to the threshold ϕ. For example, given $\phi = 0.95$, the hot basic blocks of a program consist of those that allow us to account for at least 95% of the instructions executed at runtime. A value of $\phi = 1.0$ will consider every basic block to be hot. Finally, using the same parameters, an edge is consider to be *hot* with respect to the threshold ϕ when its execution count is at least N. For all our experiments we have used a ϕ value of 0.6.

4.2. Measuring static precision

We start evaluating the effectiveness of each alias analysis described in Table 1 by comparing their static accuracy in terms of "disambiguation queries". A *disambiguation query* is a question made to the memory disambiguator about the relationship between two memory instructions. We are not really interested in discovering the exact type of relationship (i.e., dependency or independency), but rather in whether the analysis returned an answer different than the "*unknown*" relationship. We consider that a given alias analysis "is better" if it returns less "*unknown*" responses.

Since our different alias analyses are not being driven by any particular optimization, we have generated a representative set of queries by using the following algorithm. First, we consider every load/store instruction within the *hot basic blocks* of every function[4]. For every candidate, we start looking back over all paths for load/store instructions that reach the candidate. For each load/store instruction found and its candidate, a query is made to the disambiguator. We believe this scheme faithfully mimics the behavior of many compiler optimizations, which will only generate queries of instruction pairs connected by existing paths.

[4]We choose instructions only from the *hot* path because we are only interested in measuring the precision of critical instructions in a program.

For each benchmark, Figure 6 presents the percentage of queries *successfully resolved* (that is, the resulting answer is different than "unknown") by each alias analysis. Thus, the left bar presents the relative contribution of each analysis to the resolution of our given full set of queries. The right bar presents the same results restricting queries to those where both components are instructions from *hot* paths.

From the results shown in Figure 6, four main conclusions can be drawn. First, speculative alias analysis is quite beneficial: aliasing precision increases to 83% in average (74% considering only hot path references), from a baseline precision around 16% corresponding to the non-speculative schemes. Some cases such as `gcc` or `compress` achieve up to 95% of precision. Of course, this spectacular reduction in the number of "unknown" responses will only translate into positive opportunities for optimization if the number of misspeculations (errors made by our analysis) is sufficiently low. This point will be discussed in the next section.

Second, the profiling information proves very useful to increase the accuracy of both the non-speculative residue-based analysis and the speculative region-based analysis. Indeed, the profile-guided analyses almost double the total accuracy achieved by these same methods without using profile information (50% accuracy in front of 26%). An interesting exception is `go`, where the profile-guided schemes do not significantly increase precision. The reason is that, in go, almost every execution path is a hot path. Here, profile data does not reduce the number of paths to be considered.

Third, the $PG_{Reg'}$ heuristic based on the assumption that memory cells will not be used as memory pointers achieves a high level of precision. This is very surprising, since our benchmark programs make heavy use of pointers, which are naturally stored in memory. For example, perl jumps to a 94% of accuracy from a 53% achieved by the previous analyses. This program makes heavy use of dynamically linked lists. Consequently, there are many load instructions that indeed read from memory a value that is later used as a pointer. The other alias analyses correctly assume the worst scenario, and set the corresponding descriptors to $\perp$. The key insight is that pointers in a linked list hardly ever are aliased to each other. Thus, $PG_{Reg'}$, although being extremely aggressive in assuming independence, is right most of the time, as next section will show.

Finally, comparing results for the full set of queries (left bars) versus query results for the hot path only (right bars), it is clear that accuracy for the profile-guided schemes is slightly lower on the hot path. This was expected since profile-guided schemes simply return "likely independent" on those queries containing an instruction that belongs to a cold path, thus increasing accuracy on the full set of queries.

4.3. Measuring misspeculation rate

As mentioned in the previous section, speculating at analysis-time will open opportunities for speculative optimizations, which will only be profitable if our guesses are mostly correct. Otherwise, the cost of the particular recovery scheme implemented by the optimization will offset the benefits of our speculative alias analysis. This section presents data on the number of times that each speculative disambiguator produces a wrong answer.

Misspeculation rate must be measured at run time and depends on the program input data (we used variants of the official SPEC input sets to keep simulation time down to a manageable value [14]). The process we used is as follows. When running the speculative alias disambiguator in our link-time optimizer, the compiler generates a file including every query whose answer is "*likely independent*"[5]. Then, we modified the *safe* simulator of the SimpleScalar 3.0 toolset [6] to read this file at start time and build a hash table with all queries. Every time that a load/store instruction is reached, the hash table is checked to see if the instruction is a component of a query (or queries). If this is the case, we then check if the other member of the query pair has been executed in the past. If so, we compare their effective addresses, which were stored also in the hash table when each member of the pair was executed. If the effective addresses overlap, we have a misspeculation and we increase the misspeculation counter for that particular query. In any case, the total execution counter for the query is also incremented. At the end of the simulation run, we have for each query the number of times it was dynamically executed and the number of times it was misspeculated.

[5]We chose only queries where both components are *hot* instructions, since the rest of possible queries will be rarely executed at run time.

From this procedure we obtain two sets of results, presented in Table 2. First, Table 2a shows the number of queries that were misspeculated *at least once*, presented as a percentage of the total number of queries that were sometime executed. However, to get a complete picture of the cost of misspeculation, we need to know *how many times* a misspeculated pair of instructions was executed to know the number of times that the associated recovery scheme would have been invoked. Table 2b presents this second set of data, showing the *total* number of misspeculations, presented as a percentage over the total number of dynamically executed queries. The last column in both data sets corresponds to an "aggressive disambiguation" approach assuming that "unknown" query responses will also be handled as if they were "likely independent". This will give a measure of whether the proposed alias analysis methods are useful.

From the table, we can highlight several interesting points. First, the region-based analysis, despite being unsafe, is extremely accurate, to the point that we did not find a single misspeculation across all benchmarks. Second, the other speculative analysis are fairly accurate too, with a static misspeculation rate typically below 2%. Measured in terms of dynamic misspeculation rates, the situation is even more favorable. For example, for go, even though more than 8% of the static queries were misspeculated, their weight over the total number of executed queries is much lower, typically below 2%. Finally, when comparing this results against the "always speculative" approach, the misspeculation rate doubles the rate obtained using the $PG_{Reg'}$ method (a mean rate of 2.2% in front of 1.04%). This fact is encouraging, hinting that even for optimizations with high-cost recovery techniques, our speculative alias analysis might prove very useful given its high accuracy.

5. Related work

While a number of systems have been described for optimization of executable code [28, 17, 9, 23], to the best of our knowledge, alias analysis carried out by these systems is limited to fairly simple local analysis. On the other hand, there is an extensive work on pointer alias analysis of various kinds [30, 27, 12, 8, 16]. However, in almost all cases, these are high level analyses carried out in terms of source language constructs that ignore features typically encountered in executable programs. As a result, such analyses are of limited utility at the machine code level.

Amme *et al.* [2] present a general method to detect data

Benchmark	(a) Static misspeculation rate (%)					(b) Dynamic misspeculation rate (%)				
	Region	PG_{Res}	PG_{Reg}	$\text{PG}_{Reg'}$	Always	Region	PG_{Res}	PG_{Reg}	$\text{PG}_{Reg'}$	Always
`go`	0.00	13.54	13.09	8.97	10.20	0.00	1.60	0.98	0.70	1.14
`m88ksim`	0.00	0.97	1.49	0.80	2.10	0.00	0.02	0.92	0.32	8.42
`gcc`	0.00	1.84	1.96	1.58	2.16	0.00	2.32	1.80	1.02	1.93
`compress`	0.00	1.49	2.07	1.86	2.67	0.00	0.20	3.05	2.48	2.73
`li`	0.00	1.25	1.14	0.65	1.20	0.00	0.46	0.82	0.75	1.01
`ijpeg`	0.00	1.02	1.03	1.36	1.93	0.00	0.73	0.96	2.36	3.21
`perl`	0.00	1.18	2.53	0.91	1.13	0.00	0.81	1.66	0.70	1.49
`vortex`	0.00	0.23	0.64	2.17	4.44	0.00	0.04	0.15	1.99	2.21
G. Mean	0.00	1.37	1.87	1.57	2.48	0.00	0.34	1.00	1.04	2.20

Table 2. (a) Percentage of queries that were misspeculated at least once, relative to the total number of sometime-executed queries. (b) Percentage of dynamic queries misspeculated, relative to the total number of dynamic queries.

dependencies in assembly code by using symbolic value propagation. They are able not only to provide *may*-alias data, but also *must*-alias information, which allows to derive memory dependencies. However, the algorithm does not work beyond procedure boundaries, and symbolic values are not propagated through memory. Although it has been applied to assembly code, it is not obvious that using the algorithm for interprocedural whole-program analysis would scale up to problems of this size.

There is a considerably body of work on interprocedural dataflow analyses design to analyze only part, but not all, of a program (see, for example, [3, 5, 13, 26]), although only some of them use profile information to guide their decisions. This profile information is, however, widely used when performing optimizations [24, 7, 10, 18]. On the other hand, while speculation has been commonly used in the compiler world for optimizing programs [19, 15, 20, 18, 25], as far as we know, this is the first attempt to introduce unsafe speculations into a dataflow analysis algorithm.

6. Summary and future directions

Code transformations on executable code can benefit greatly from pointer alias information. However, the problem of memory disambiguation is one of the weak points of object code modification, because high level information available in a traditional compiler is lost. Besides, existing alias analyses turn out to be of limited utility at the machine code level, because either they do not consider typical issues of binary code, or their application is too expensive to be practical for analyzing large binaries.

This paper has presented two approaches to high-quality, low-cost, speculative alias analysis, to be applied in the context of link-time or executable code optimizers. The key idea behind our two proposals is to trade off analysis complexity against *safeness*. First, we presented a region-based alias analysis that disambiguates memory references by classifying them into separate memory regions and assumes that, whenever an operation is performed on a pointer, the pointer will not change its pointed-to memory region. This assumption yields increases in disambiguation accuracy, and, four our set of benchmarks and inputs, did not cause a single misspeculation. Our second proposal uses profile information and assumes that instructions on hot paths are not aliased to memory references in cold paths. This assumption yields substantial increases in disambiguation accuracy. When combining the two algorithms with a third unsafe assumption that memory cells do not contain pointers, accuracy increases to over 80%, while the runtime misspeculation rate is still below 2%.

Although it has been commonly used for optimizing programs, speculation has been only introduced so far in the compiler world at analysis-time by means of a set of rules and heuristics. As far as we know, this is the first attempt to systematically introduce unsafe speculations into dataflow analysis algorithms. Speculative alias analysis is particularly well suited to be used in combination with speculative optimizations based on reordering memory operations [19, 15, 20, 25], including those related to the IA-64 architecture [4]. The low dynamic misspeculation rate makes our analyses suitable even for optimizations with high-cost recovery schemes. Future investigation will further consider the application of our algorithms in combination with these type of optimizations at link-time.

Acknowledgments

The authors gratefully thank the anonymous reviewers for their detailed comments and suggestions, which have helped improve both the contents and presentation of this paper significantly. This work is being supported by the European Union and the Spanish Ministry of Science and Technology, under grants TIC2001-0995-C02-01 and PN98 46057403-1. The research described in this paper has been developed using the resources of the CEPBA.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers principles, techniques, and tools*. Addison-Wesley, Reading, MA, 1986.

[2] W. Amme, P. Braun, E. Zehendner, and F. Thomasset. Data dependence analysis of assembly code. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, pages 340–347, Paris, France, Oct. 12–18, 1998. IEEE Computer Society Press.

[3] G. Ammons and J. R. Larus. Improving data-flow analysis with path profiles. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 72–84, Montreal, Canada, June 1998.

[4] J. Bharadwaj, W. Y. Chen, W. Chuang, G. Hoflehner, K. Menezes, K. Muthukumar, and J. Pierce. The Intel IA-64 compiler code generator. *IEEE Micro*, 20(5):44–53, 2000.

[5] R. Bodík and S. Anik. Path-sensitive value-flow analysis. In *The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 237–251, Orlando, Florida, Jan. 19–21, 1998.

[6] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report CS-TR-97-1342, CS Department, University of Wisconsin-Madison, 1997.

[7] P. P. Chang, S. A. Mahlke, W. Y. Chen, and W. mei W. Hwu. Profile-guided automatic inline expansion for C programs. *Software Practice and Experience*, 22(5):349–369, 1992.

[8] B.-C. Cheng and W. mei W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 57–69, June 2000.

[9] R. Cohn, D. Goodwin, P. G. Lowney, and N. Rubin. Spike: An optimizer for Alpha/NT executables. In USENIX, editor, *The USENIX Windows NT Workshop 1997*, pages 17–23, Seattle, Washington, Aug. 11–13, 1997.

[10] R. Cohn and P. G. Lowney. Hot cold optimization of large Windows/NT applications. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 80–89, Paris, France, Dec. 2–4, 1996. ACM Press.

[11] S. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. In *The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 12–24, Orlando, Florida, Jan. 19–21, 1998.

[12] A. Diwan, K. S. McKinley, and J. E. B. Moss. Type-based alias analysis. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 106–117, Montreal, Canada, June 1998.

[13] E. Duesterwald, R. Gupta, and M. L. Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems*, 19(6):992–1030, Nov. 1997.

[14] M. Fernández and R. Espasa. Speculative alias analysis for executable code. Technical Report UPC-DAC-2002-27, Computer Architecture Department, Universitat Politècnica de Catalunya, Barcelona, 2002.

[15] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. mei W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–193. ACM SIGARCH, SIGOPS, SIGPLAN, and the IEEE Computer Society, Oct. 1994.

[16] R. Ghiya, D. Lavery, and D. Sehr. On the importance of points-to analysis and other memory disambiguation methods for C programs. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation*, pages 47–58, June 2001.

[17] D. W. Goodwin. Interprocedural dataflow analysis in an executable optimizer. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 122–133, Las Vegas, Nevada, June 1997.

[18] R. Gupta, D. A. Berson, and J. Z. Fang. Path profile guided partial redundancy elimination using speculation. In *Proceedings of the 1998 International Conference on Computer Languages*, pages 230–239, Chicago, May 14–16, 1998.

[19] A. S. Huang, G. Slavenburg, and J. P. Shen. Speculative disambiguation: A compilation technique for dynamic memory disambiguation. In *Proceedings of the 21st Symposium on Computer Architecture*, pages 200–210, Chicago, Illinois, Apr. 1994. ACM SIGARCH.

[20] M. Moudgill and J. H. Moreno. Run-time detection and recovery from incorrectly reordered memory operations. Technical Report RC-20857, IBM Research Report, 1997.

[21] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, 1997.

[22] R. Muth. *Alto: A Platform for Object Code Modification*. PhD thesis, Department of Computer Science, University of Arizona, 1999.

[23] R. Muth, S. Debray, S. Watterson, and K. de Bosschere. alto: A link-time optimizer for the Compaq Alpha. *Software Practice and Experience*, 31(6):67–101, Jan. 2001.

[24] K. Pettis and R. C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 16–27, June 1990.

[25] M. A. Postiff, D. A. Greene, and T. N. Mudge. The store-load address table and speculative register promotion. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 235–244, Los Alamitos, CA, Dec. 10–13, 2000. IEEE Comp. Society.

[26] A. Rountev, B. G. Ryder, and W. Landi. Data-flow analysis of program fragments. In *ESEC/FSE '99*, volume 1687 of *Lecture Notes in Computer Science*, pages 235–252. Springer-Verlag / ACM Press, 1999.

[27] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, Jan. 1997.

[28] A. Srivastava and D. W. Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, 1(1):1–18, Dec. 1992.

[29] A. Tamches and B. P. Miller. Dynamic kernel code optimization. In *Proceedings of the 3rd Workshop on Binary Translation*, Barcelona, Spain, Oct. 2001.

[30] 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*, pages 1–12, La Jolla, California, June 18–21, 1995.