Memory Disambiguation for General-Purpose Applications

Chi-Keung Luk
luk@cs.toronto.edu

Department of Computer Science
University of Toronto
Toronto, Ontario, M5S 1A4

Abstract

Memory disambiguation is a technique for removing spurious data dependences that severely limit the compiler's freedom of code-scheduling. Though dependence tests for linear array references work well in many scientific applications, they are incapable of handling *pointer dereferences*, which are very common in *general-purpose* applications written in C-like languages.

This paper[1] is a survey of several recently proposed memory disambiguation schemes that can handle pointer dereferences. They are classified as either *static* or *dynamic*. Static disambiguation uses data-flow analysis to compute an approximation of the set of memory aliases at each program point. Four static disambiguators are compared. Some evidences show that they may have already achieved high accuracy in practice.

Dynamic disambiguation resolves at run-time any ambiguous data dependences that cannot be determined during compilation. A pure software-based and two hardware-assisted dynamic disambiguators are examined. While they can bridge part of the gap between a raw static disambiguator and a perfect one, there is still room for improvement.

Based on our studies, we propose a number of future directions to develop memory disambiguators of better performance/cost.

1. The IBM contact for this paper is Graham Warren, manager, TOBEY Compiler Development, Compiler Back-End Technology, IBM Canada Laboratory.

1 Introduction

Determining whether two memory references access the same location (often known as the *memory aliasing* problem) is a critical step in both exploiting parallelism and improving data locality. Two instructions can be executed in parallel if they access different locations. On the other hand, two memory operations enjoy locality if they access the same or overlapped locations. In scientific applications, many references are made through arrays with linear subscript expressions. Although the aliasing problem over linear subscript expressions is NP-complete in theory, it is manageable in practice by a number of dependence tests [18, 29].

Nevertheless, the aliasing problem is worse in non-scientific applications where many references are made through *pointers*. This problem has been intensified as languages that support pointers (e.g., C, C++) have become popular. Consider the two C program fragments in Figure 1, one using array subscripts (Figure 1(a)) and one using pointers (Figure 1(b)), to exchange every pair of mirror elements of the array `a`.

While the pointer version might produce more efficient address computation code, the array version is more appropriate for parallelizing compilers to generate parallel- and memory-efficient code. A parallelizing compiler would be able to detect that for the array version, `a[i]` and `a[N-1-i]` of different iterations will never refer to the same location and hence the loop can be parallelized (the data dependence on `temp` can be easily removed by scalar expansion [4]). However, detecting this possibility of parallelization in the pointer version is still not possible in most existing compilers.

```
for (i=0; i<N/2; i++)
{
   temp = a[i];
   a[i] = a[N-1-i];
   a[N-1-i] = temp;
}
```
(a) array implementation

```
for (p=&a[0],q=p+N-1; p<q; p++,q--)
{
   temp = *p;
   *p = *q;
   *q = temp;
}
```
(b) pointer implementation

FIGURE 1. Exchange every pair of mirror elements of `a`

The techniques for coping with memory latency can also be quite different in the two versions. To illustrate this point clearly, let us assume that a cache block is `N` bytes in size and each element of `a` takes one byte. For the array version, the compiler could know that all elements needed in succeeding iterations are bought into the cache in the first iteration due to spatial locality, and hence nothing needs to be done to improve locality. In contrast, for the pointer version, since many compilers are not good in performing locality analysis on pointer dereferences, the compiler may do more than necessary to tolerate latency, such as emitting prefetches for `*q` (and `*p`) in a software pipelining manner for all iterations [31].

A number of experimental studies have also confirmed the importance of memory disambiguation [16, 24, 32, 34]. Wall [34] classifies memory disambiguation (referred to as "alias analysis" in his paper) into four levels of sophistication: (i) none, (ii) by inspection, (iii) by compiler, and (iv) perfect. He shows that sophisticated memory disambiguation (i.e., "by compiler" or "perfect") can increase the amount of exploitable ILP by more than 100% over no disambiguation for 18 benchmarks including both non-scientific and scientific applications. Experimental results of the three dynamic schemes [16, 24, 32] that we will discuss later also show significant speedup contributed by memory disambiguation over a wide range of applications.

Memory disambiguation can be done statically, dynamically, or through a combination of both. *Static memory disambiguation* is performed by the compiler, whereas *dynamic memory disambiguation* is accomplished at run-time through some additional code and/or hardware. Section 2 compares four static disambiguation schemes along three dimensions of approximation. Section 3 examines three dynamic disambiguation schemes with various degrees of hardware support. Finally, Section 4 suggests a number of future directions inspired by some experimental results of the work reported in this paper.

2 Static Disambiguation

The scope of static memory disambiguation includes the aliasing induced by pointer assignment and the passing of procedure parameters. A significant amount of research has been done in this area [2, 3, 6, 7, 9, 10, 11, 12, 23]. In this section, we compare four recently proposed algorithms for general pointer-induced aliasing [5, 13, 27, 28, 35].

Static memory disambiguation determines a static *approximation* of the actual (dynamic) aliases at every program point[2] *P*. There are two standard approximations: the *may-alias* approximation returns an estimate of the set of aliases that will hold at *some* instance of *P*; and the *must-alias* approximation returns an estimate of the set of aliases which hold at *every* instance of *P*.

The four algorithms can be characterized by their degrees of approximation along the following three dimensions [30]:

- *representation of aliases*
- *context-sensitivity*
- *depth of indirection*

---

2. A program point is a place between two statements.

Refinements in any of the three dimensions can lead to more precise approximations at the expense of time and/or space.

2.1 Representation of Aliases

The representation of aliases in an algorithm determines the space requirement, and also affects the preciseness of the algorithm. A representation of aliases can be *exhaustive* or *concise*. An exhaustive representation stores every alias pair explicitly. A concise representation only stores some basic aliases, and can deduce new aliases from existing ones by means of dereferencing and transitive closure. Therefore, an exhaustive representation requires more space than a concise one, but the latter approach may take more time to deduce aliases.

2.1.1 Alias Relations and Points-to Relations

There are two common forms of representation of aliases: *alias relations* and *points-to relations*. An alias relation `(a,b)` is an unordered pair that states that `a` and `b` are referring to the same memory location. A points-to relation `<p,q>` is an ordered pair that states that `*p` and `q` are referring to the same location. Points-to relations are concise. Emami et al. [13] and Wilson and Lam [35] use points-to relations in their algorithm. Alias relations can be exhaustive or concise. Landi et al.'s algorithm [27, 28] remembers all alias relations at each program point and hence is exhaustive. Choi et al. [5] employ a compact representation of alias relations that is in fact equivalent to the points-to representation. To ease our presentation, we use "`()`" to enclose an alias relation and "<>" to enclose a points-to relation.

While points-to relations require less space than alias relations, neither representation is always more precise than the other in the presence of multi-level pointers. Consider the C program shown in Figure 2(a). The corresponding alias relations and points-to relations at the four program points *P0*, *P1*, *P2* and *P3* are listed in Figure 2(b). In this example, the alias relation `(**a,c)` induced by `(*a,b)` and `(*b,c)` at *P2* becomes spurious at *P3*. In contrast, for the example in Figure 3, the points-to relation at *P3* can infer a spurious alias between `**x` and `z` by computing their transitive closure.

Note that both of the above problems can be handled by increasing the complexity of the algorithm. The problem in Figure 2 can be avoided by knowing that `(*a,b)` is a "must alias" at *P3* and so is `(*b,d)`, and hence `(**a,c)` is impossible. The problem in Figure 3 can be avoided by keeping track of the path on which a points-to relation is propagated along, and restricting that transitive closure is only applicable to points-to relations propagated along the same path. Also note that the points-to relations are more compact than the alias relations in both examples.

2.2 Context-sensitivity

Recent work on pointer-induced aliasing has emphasized *interprocedural* and *context-sensitive* analysis. Interprocedural pointer analysis is essential because the set of aliases is subjected to change across procedures due to pointer assignment and dynamic memory allocation. A *context* of a statement *S* is the chain of procedure invocations starting at `main()`[3] and terminating at the procedure that encloses *S*. Context-sensitive interprocedural pointer analysis is used to distinguish different invocations of the same procedure.

Various frameworks for context-sensitive interprocedural pointer analysis have been used. These include *Procedure Call Graph*s (PCG) (used by Choi et al. [5]), *Interprocedural Control Flow Graphs* (ICFG) (used by Landi et al. [27, 28]), and *Invocation Graphs* (IG) (used by Emami et al. [13] and Wilson and Lam [35]). For example, the PCG, ICFG, and IG for the program in Figure 4(a) are shown in Figure 4(b), (c), and (d) respectively.

In a PCG, each node denotes a different procedure in the program, and every procedure is denoted by exactly one node. Each edge denotes a distinct call site. Thus, the two edges from node

---

3. `main()` is the first procedure to be called in every C program.

```
main()
{
   int **a, *b, c, d;
               /* P0*/
   a = &b;
               /* P1 */
   b = &c;
               /* P2 */
   b = &d;
               /* P3 */
}
```

Program points	**Alias relations**	**Points-to relations**
P0	{}	{}
P1	{(*a,b)}	{<a,b>}
P2	{(*a,b), (**a,*b), (*b,c), (**a, c)}}	{<a,b>, <b,c>}
P3	{(*a,b), (**a,*b), (*b,d), **(**a,c)**, (**a,*d)}	{<a,b>, <b,d>}

FIGURE 2. An example that the points-to relations are more precise

```
main()
{
   int **x, *y, z;
   ...
            /* P0 */
   if (z)
      {
         x = &y;
            /* P1 */
      }
   else
      {
         y = &z;
            /* P2 */
      }
            /* P3 */
   ...
}
```

Program points	**Alias relations**	**Points-to relations**
P0	{}	{}
P1	{(*x,y), (**x,*y)}	{<x,y>}
P2	{(*y,z)}	{<y,z>}
P3	{(*x,y), (**x,*y), (*y,z)}	**{<x,y>, <y,z>}**

FIGURE 3. An example that the alias relations are more precise

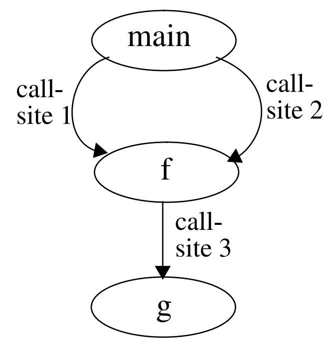
"main" to node "f" represent the two calls of `f()` in `main()`. An ICFG is the union of control flow graphs for each procedure and a set of call, return, entry, and exit nodes. Call nodes are connected to the entry nodes of procedures they invoke; exit nodes are connected to return nodes corresponding to the calls. In an IG, each node denotes an invocation of a procedure within a particular context. Each procedure invocation chain (i.e., a context) is represented by a unique path in the IG, and vice versa. For example, there are two distinct paths from node "main" to node "g" in the IG in Figure 4(d) because `f()` is invoked twice in `main()`.

The algorithms of Emami et al. [13] and Wilson and Lam [35] are *fully* context-sensitive since they propagate aliasing information along edges in the invocation graph so that every context is distinguishable. In contrast, the algorithms of Landi et al. [27, 28] and Choi et al. [5] are *partially* context-sensitive in the sense that they merge aliasing information from different contexts at the entries of procedures and need some ways to recover contexts at the exits of procedures. Consider again the example in Figure 4. An algorithm using the IG can easily find out that *S* = `{(*i,j), (**i,*j), (**i,k)}` holds after call site 2 but not after call site 1 because `(*i,j)` is propagated to `f()` and `g()` along a unique path (i.e., call site 2 and then call site 3) and the result *S* is then propagated backward along the same path. However, an algorithm using the ICFG or PCG will not know which call site *S* should be propagated back to by looking at the graph only. Propagating it back to both call sites is imprecise. To avoid this kind of impreciseness, partially context-sensitive algorithms propagate some auxiliary information in addition to the aliasing information. In particular, Landi et al.'s algorithm uses *assumed alias*

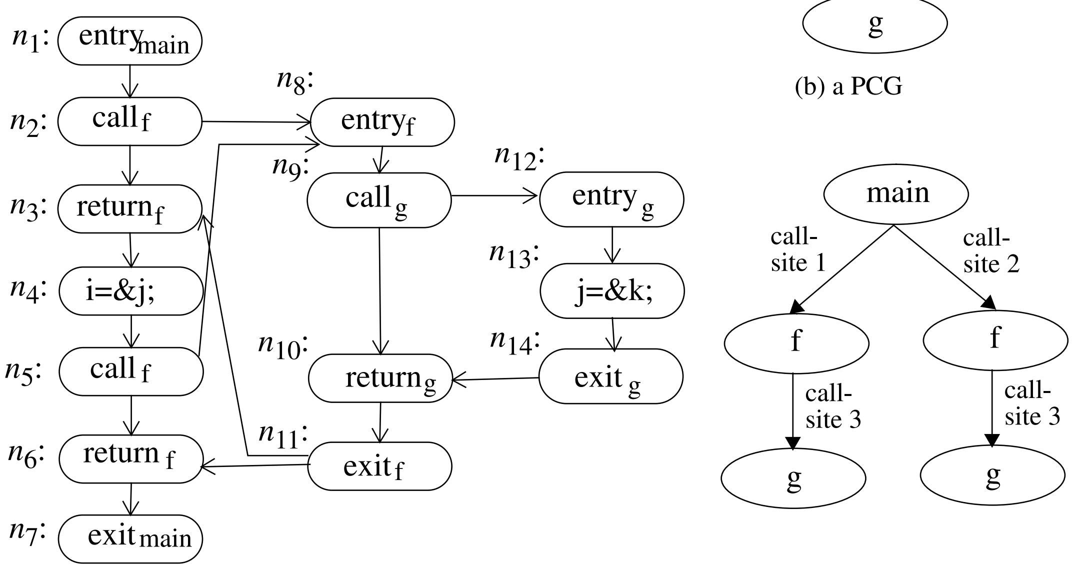
```
int **i, *j, k;
main()
{
   f();/*call site 1*/
   i=&j;
   f();/*call site 2*/
}
```

```
f()
{
   g();/*call site 3*/
}
g()
{
   j=&k;
}
```

(a) a C program



(b) a PCG



(c) an ICFG

(d) an IG

FIGURE 4. Three program representations for context-sensitive interprocedural pointer analysis

relations and Choi et al.'s algorithm uses *last call sites* to improve their precision.

2.2.1 Assumed Alias Relations and Last Call Sites

Landi et al. associate a set of assumed alias relations *AA* with every alias relation *AR* at a program point *P*. *AR* will hold at *P* if *AA* holds at the entry node *N* of the procedure containing *P*. To avoid considering an exponential number of assumed alias relations, they restrict *AA* to be a singleton. This may cause imprecision when they recover the calling context by determining the call sites that can propagate *AA* to *N* in the presence of multi-level pointers. For example, applying their algorithm to the program in Figure 5 will generate a spurious alias `(**b,e)` at either program point *P1* or *P2* (but not both). The problem here is that at *P3*, `(**b,e)` is conditional on two alias relations `(*a,*b)` and `(*c,e)` which cannot be true at the same time. However, since only one of them is able to be recorded as the assumed alias relation for `(**b,e)` at `P3`, the spurious alias will hold at *P1* if `(*c,e)` is recorded or at *P2* if `(*a,*b)` is recorded.

Choi et al. remember the last call site for each alias relation as a means of recovering the context. The problem with this scheme is that it cannot distinguish contexts composed of different invocations except the last ones. For example, for the program in Figure 6, their algorithm will report the alias relations `{(*a,b),(**a,*b),(*b,c),(**a,c)}` at `P1`, which is incorrect since

```
int **a, **b, *c, *d, e;
F()
{
   a=&d;
   c=&e;
   H();
/* P1 */
   ...
}

G()
{
   a=b;
   H();
/* P2 */
...
}

H()
{
   *a=c;
/* P3 */
}
```

FIGURE 5. Imprecision in recovering context by assumed alias relations.

```
int **a, *b, c;
w()
{
   a=&b;
   f(); /* call-site 1 */
}
v()
{
   b=&c;
   f(); /* call-site 2 */
}

f()
{
   g(); /* call-site 3 */
}
g()
{
/* P1 */
}
```

FIGURE 6. Imprecision in recovering context by the last call site.

(*a,b) and (*b,c) cannot be true in the same context.

2.2.2 Efficiency of Fully Context-sensitive Algorithms

Fully context-sensitive algorithms are desirable for their preciseness but may need to analyze an exponential number of cases in theory. Emami et al.'s algorithm will reanalyze a procedure when the input set of points-to relations is different from those that are analyzed before. That is, it will reanalyze a procedure for each of its calling contexts[4].

To achieve full context-sensitivity yet maintain efficiency, Wilson and Lam propose a *partial transfer function* (PTF) scheme for computing the effects of a procedure on the points-to relations. The idea is that for each procedure, the input values (i.e., sets of points-to relations) are partitioned into a number of disjoint domains. Each domain is given a PTF to compute the procedure's effects. A PTF is applicable to an input value if it falls in the PTF's domain. A PTF should be easy to compute yet cover a wide range of input values. To achieve this goal, domains are partitioned according to the initial set of points-to relations and the values of procedure pointers. The intuition behind this partitioning scheme is that some input values propagated from different contexts are likely to have the same alias relationships between parameters and hence can share a common PTF. All the inputs in a PTF's domain must possess the same values for procedure pointer parameters because these determine the procedures that will be invoked in the procedure under analysis.

To show the differences between the two algorithms, consider the program in Figure 7. Emami et al.'s algorithm will analyze procedure h three times since the input sets of points-to relations are different in the three invocations of h. Wilson and Lam's algorithm will only analyze h twice. The input values to the first two invocations fall into the same domain: {<x,1_x>,<y,1_x>, <e,f>} because parameters x and y point to the same object 1_x in both invocations. The actual value of 1_x is not important in this example as it is not dereferenced in h. Thus, at the first invocation of h, <a,1_x> is not included in the domain. For the same reason, <b,1_x> is also not included in the domain of the second invocation of h. Therefore, the first two invocations share

4. Emami mentioned in her thesis that not all contexts needed to be reanalyzed, but some could be memorized for later reuse [15].

```
main()
{
  int **a, **b, **c;
  ...
  h(a, a, f);
  h(b, b, f);
  h(a, a, g);
  ...
}
f()
{
... /* no pointer assignment */
}
```

```
g()
{
... /* no pointer assignment */
}
h(int **x, int**y, int (*e)())
{
  *x = *c;
  *c = *y;
  (*e)();
}
```

FIGURE 7. An example showing the differences between two fully context-sensitive algorithms.

```
int *i, *j;
main()
{
  i = f(); /* call-site c1 */
  j = f(); /* call-site c2 */
}
```

```
f()
{
  return((int*)malloc(sizeof(int));
/* the name of this object is O */
}
```

FIGURE 8. A context-sensitive *k*-limited approach

the same PTF. A new PTF is required for the third invocation of `h` because a new value is passed as the procedure pointer parameter `e`.

2.3 Depth of Indirection

Naming anonymous objects is a common problem for all aliasing analysis algorithms that consider heap objects. The number of names of heap objects is potentially infinite. A well-known finite approximating scheme is called the *k-limited* approach [25] in which any pointer dereference can have at most *k* levels of indirection where *k* is a predefined constant. Pointer dereferences with the same first *k* levels of indirection are indistinguishable. This scheme is used in Landi et al.'s algorithm [27].

Choi et al. enhance the *k*-limiting approach by associating a heap object with the place in the program where it is created. Consider the program in Figure 8. In the original *k*-limited approach, the two objects created at call sites *c1* and *c2* have the same name, say *O*, and so a spurious alias `(*i,*j)` is generated. In their context-sensitive *k*-limited approach, the first object created has the name *c1O* and the second object created has the name *c2O*. Hence, under this naming scheme, two objects are indistinguishable only if they are created at the same place with the same context and have the same first *k* levels of indirection.

Wilson and Lam use Choi et al.'s scheme for naming heap objects. However, they do not use field names to access fields within structures. Instead, they access a field in a structure by specifying its offset from the beginning of the structure. This solves the aliasing problems caused by type casting and union types.

Emami et al. advocate that the analysis of stack objects and heap objects should be treated as two different problems. They name all dynamically created objects as *heap* to obtain a safe but imprecise approximation of all points-to relations from stack objects to heap objects and vice versa. Their preliminary results show that this approximation is not a problem. More research is needed to examine the accuracy of this approximation.

Hendren et al. have proposed an approach that is based on the *path-matrix model* [21] for analyzing pointer dereferences between heap objects [19, 20, 21]. A principal difference between their approach and those we mentioned above is that it focuses on the aliasing between individual ele-

TABLE 1. A comparison of four recent pointer-analysis algorithms

	Landi et al.[a]	**Choi et al.**	**Emami et al.**	**Wilson and Lam**
Alias representation	complete alias relations	compact alias relations	points-to relations	points-to relations
Context-sensitivity	recover context by assumed alias sets; partially context-sensitive	recover context by last call sites; partially context sensitive	propagate information in invocation graphs; fully context sensitive	the same as Emami et al.'s algorithm
Depth of indirection	*k*-limiting	context-sensitive *k*-limiting	name all heap objects by a common name *heap*. Pointer dereferences between heap objects are decoupled	context-sensitive *k*-limiting; fields are accessed in terms of their offsets in structures
Target language	no type casting, no union structures, no procedure pointers	the same as Landi et al.'s algorithm	can handle procedure pointers and common type casting; no union in the implementation reported in Emami's thesis [15]	can handle all features in C

a. A detailed comparison between Landi et al.'s algorithm and Emami et al.'s algorithm can be found in Emami's thesis [15].

ments of dynamic data structures, while other approaches aim to distinguish separate data structures from one another. By knowing the aliasing relationships between individual elements of a dynamic data structure at each program point, the compiler can perform some valid transformations (mainly for parallelization) at appropriate places in the program. To be able to perform an accurate analysis, it requires the programmer to give hints to the compiler through *annotated data declarations* wherever the compiler cannot discover the underlying properties of the data structures automatically. Recently, Ghiya has extended the path-matrix based approach to analyze both the dependences between disjointed data structures and those between individual elements of each data structure [17].

2.4 Summary

We have seen a number of recent algorithms for analyzing pointer-induced aliasing. Beside their differences in the three dimensions of approximation, they also vary in the target language (all consider C-like languages) that they can handle. We summarize the important differences between them in TABLE 1.

3 Dynamic Disambiguation

A static memory disambiguator can have four[5] possible answers to an aliasing problem:

- No: never alias.
- Must: alias in every instance.
- May: alias in at least one instance.
- Unknown: cannot determine statically.

Without any run-time support, the compiler must treat "Unknown" as "May" to preserve correctness. However, this may be too conservative since aliasing may rarely occur or even never occur in the "Unknown" case. Dynamic memory

5. Some disambiguators return "Yes" for both "Must" and "May" and so they only give three answers.

disambiguation enables the compiler to optimistically treat “May” and “Unknown” as “No”. The validity of such assumptions is checked at run-time. If an assumption is found to be wrong, the corresponding recovery action will be invoked to preserve the correct semantics.

A number of hardware- and software-based schemes for dynamic memory disambiguation have been proposed. Out-of-order execution architectures [22, 26] reschedule memory operations based on the known memory addresses at run-time. However, the potential gain by this hardware-only scheme is small due to the limited size of the hardware instruction window.

3.1 A Software Scheme

Nicolau implements a pure software disambiguator called RTD [32] in the Bulldog VLIW compiler [14]. RTD inserts conditional branches to compare ambiguous memory addresses at run-time. Programs with RTD are scheduled in the same way as those without RTD by the trace scheduler. RTD can be conservative or aggressive. Conservative RTD has the restriction that ambiguous memory operations cannot be executed out of order *before* the address comparison. Aggressive RTD relieves this restriction by providing recovery code that will be invoked if the out-of-order execution violates the semantics. For example, consider the pseudo-assembly program shown in Figure 9(a). The compiler cannot determine whether `A` and `B` are aliases, and therefore `I2` must be executed before `I3` if there is no dynamic disambiguation. Conservative RTD allows `I3` to be executed before `I2` once it checks that `A` and `B` are not aliases (Figure 9(b)). Aggressive RTD allows out-of-order execution of `I2` and `I3` before the aliasing check (Figure 9(c)), where `I3′` is the recovery code.[6]

3.2 Hardware-assisted Schemes

Two of the most recent schemes for dynamic disambiguation involve hardware assistance. Gallagher et al. [16] use a special hardware structure called a *memory conflict buffer* (MCB), which is essentially a small set-associative cache for recording any incorrect memory operations detected at run-time. In particular, they focus only on store/load pairs. For example, in Figure 9(d), the load in `I3` is replaced by a new *preload* instruction that will enter the address and width of `B` into an MCB entry associated with `r3`. When the store in `I2` is executed, the hardware will detect whether `A` is an alias of a location entered in any valid MCB entry. If it is, the *conflict bit* of the corresponding register (`r3` in our example) is set. The new instruction “`check`” examines the conflict bit of `r3`. If it is set, the recovery code will be invoked.

SpD [24] is another hardware-assisted scheme that is based on guarded execution. SpD executes the recovery code before it is proven necessary. Operations that can cause side-effects are guarded by the corresponding aliasing check. In Figure 9(e), the instruction sequence from `I1` to `I5` is for the case where `alias(A,B)` is true, while the instruction sequence from `I3′` to `I5′` is for the false case. Both streams are executed simultaneously (i.e., it is possible that `I3′` is executed before `I2`). Shadow registers (i.e., `r3.T`, `r3.F`, `r4.T`, `r4.F`) are used to hold uncommitted values. Since SpD does not support shadow memory, the two stores in `I5` and `I5′` are guarded.

3.3 A Comparison

The two hardware-assisted schemes differ in the following ways. SpD is a software extension to guarded execution while MCB represents additional hardware needed by speculative execution [33]. Therefore, MCB is more expensive in terms of hardware. It is difficult to compare their relative performance since their experiments were done on two different sets of benchmarks.[7] Each scheme has advantages and disadvantages. MCB associates one piece of recovery code for every preloaded register no matter how many stores the preload has bypassed. In contrast, a copy of code

---

6. If `A` and `B` are exactly at the same location and of the same size, `I3′` can be replaced by `r3=r2`.

7. The only common program in their experiments is “espresso” which has only little speedup in both schemes.

```
I1: r2=r1*9
I2: store A=r2
I3: r3=load B
I4: r4=r4/r3
I5: store C=r4
```

(a) no dynamic disambiguation

```
I1: r2=r1*9
    if not alias(A,B) then
I3:     r3=load B
I2:     store A=r2
    else
I2':    store A=r2
I3':    r3=load B
I4: r4=r4/r3
I5: store C=r4
```

(b) Conservative RTD

```
I1: r2=r1*9
I3: r3=load B
I2: store A=r2
    if alias(A,B) then
I3':   r3=load B
I4: r4=r4/r3
I5: store C=r4
```

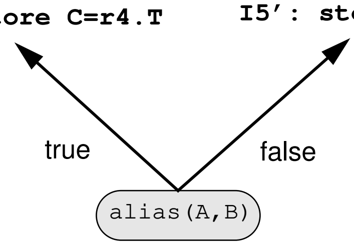
(c) Aggressive RTD

```
    I1: r2=r1*9
    I3: r3=preload B
    I4: r4=r4/r3
    I5: store C=r4
    I2: store A=r2
        check r3, recovery
  cont: ...

recovery:
    I3':r3=load B
    I4':r4=r4/r3
    I5':store C=r4
        Jump cont
```

(d) MCB

```
I1: r2=r1*9
I2: store A=r2
I3: r3.T=load B                 I3': r3.F=load B
I4: r4.T=r4/r3.T                I4': r4.F=r4/r3.F
I5: store C=r4.T                I5': store C=r4.F
```



(e) SpD

FIGURE 9. Dynamic memory disambiguation schemes

is required for every pair of disambiguated references in SpD. Thus, SpD is worse in terms of code expansion. However, the additional work needed by MCB to check overlapping addresses for every preload and store may potentially affect the cycle time. SpD incurs no branching overhead for recovery action but requires more machine resources than MCB since the total number of operations executed is *always* increased no matter whether recovery action is required or not.

Compared with RTD, both MCB and SpD provide the compiler with more freedom for scheduling. A limitation of RTD (or any pure software scheme) is that exception-taking instructions that depend on any aliasing condition are not allowed to move above the aliasing check. For instance, in Figure 9(c), `I4` and hence `I5` cannot be lifted above the check since `I4` may cause an exception if `r3` equals zero. MCB and SpD do not have this limitation because they are based on architectures that can handle speculative exceptions.

4 Future Directions

Is it necessary to support both kinds of memory disambiguation *in practice?* Dynamic disambiguation alone is not enough for two reasons. First, this may introduce too much recovery overhead at runtime if out-of-order code scheduling is performed for all potential aliases. Second, besides dependence analysis, static disambiguation is also very helpful to other analyses done by compilers such as constant propagation, reaching definitions, dead-code elimination, etc. [1]. In fact, all three of the dynamic disambiguation schemes use *raw* static disambiguators to eliminate as many ambiguous references as possible before applying dynamic disambiguation. Our first question is: *Can a sophisticated static disambiguator suffice in practice?*

Gallagher et al. [16] and Huang et al. [24] have compared their dynamic disambiguators with raw and perfect (not implementable) static disambiguators. They report that their dynamic disambiguators can bridge part of the gap between their static disambiguators and the perfect static disambiguator. Nevertheless, we observe that the performance gaps between their dynamic disambiguators and the perfect static disambiguator are still quite large in most of the benchmarks they presented.

The answer to our first question depends on how closely a sophisticated static disambiguator can approximate the perfect one. Landi et al. [28] and Emami et al. [13] report that the average number of locations pointed to by a dereferencing pointer is about 1.2, where 1.0 should be reported by a perfect disambiguator. Their results imply that these state-of-art static disambiguators may eliminate the need of dynamic disambiguation if the remaining ambiguous data dependences after static disambiguation have little effect on performance. Wilson and Lam [35] have a similar claim according to their results on two scientific applications. More research is needed to verify this claim in non-scientific applications.

Of course, one should get the most by combining a sophisticated static disambiguator with a dynamic disambiguator. The problem is that a desirable dynamic disambiguator requires special hardware, which is expensive. Nevertheless, this would not be a problem if this special hardware is already provided. This could be the case in systems that already support branch speculation. Thus, our second question is: *How can we achieve better performance improvement by combining static and dynamic disambiguation given that it is sufficiently inexpensive to do so?*

We believe that the key to this question is the precision of information conveyed from the static disambiguator to the dynamic disambiguator. In particular, the effectiveness of a dynamic disambiguator highly depends on how well it can estimate the *alias probability* of the two references in question. The constant alias probability approach used by SpD is obviously not satisfactory. Memory profiling [8] is also an expensive process for large programs.

Acknowledgments

I would like to thank Professor Todd C. Mowry, my advisor, and the anonymous referees for their helpful comments which improve the paper.

This work was financially supported by an IBM Canada Research Fellowship and a Canadian Commonwealth Fellowship.

About the Author

Chi-Keung Luk is a Ph.D. candidate at the University of Toronto and is a member of the IBM Centre for Advanced Studies. His research interests are Compiler Optimization, Computer Architecture, and Programming Languages. His advisor is Professor Todd C. Mowry.

References

[1] A. V. Aho, R. Sethi and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts, 1986.

[2] J. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Proceedings of the 6th Annual ACM Symposium on Principles of Programming Languages*, pp. 29-41, January 1979.

[3] J. M. Barth. A practical interprocedural data flow analysis algorithm. *Commun. ACM*, *21*, 9 (Sept.), 724-736.

[4] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, *26*, 4 (Dec.), 345-420.

[5] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages*, pp. 232-245, January 1993.

[6] R. Cytron and R. Gershbein. Efficient accommodation of may-alias information in SSA form. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, pp. 36-45, June 1993.

[7] K. Cooper and K. Kennedy. Fast interprocedural alias analysis. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages*, pp. 49-59, January 1989.

[8] W. Y. Chen, S. A. Mahlke, N. J. Warter, S. Anik, and W. W. Hwu. Profile-assisted instruction scheduling. *International Journal of Parallel Programming*, *22*, 2, 151-181.

[9] K. Cooper. Analyzing aliases of reference formal parameters. In *Proceedings of the 12th Annual ACM Symposium on Principles of Programming Languages*, pp. 281-290, January 1985.

[10] D. R. Chase, M. Wegman, and F. K. Zadek. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pp. 296-310, 1990.

[11] 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.

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

[13] 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.

[14] J. Ellis. *Bulldog: a compiler for VLIW architectures*. MIT Press, 1990.

[15] M. Emami. *A practical interprocedural alias analysis for an optimizing/parallelizing C compiler*. Master thesis, School of Computer Science, McGill University, August 1993.

[16] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In *Proceedings of the 6th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 183-193, October, 1994.

[17] R. Ghiya. *Practical techniques for interprocedural heap analysis*. Master thesis, School of Computer Science, McGill University, March 1995.

[18] G. Goff, K. Kennedy, and C.-W. Tseng. Practical dependence testing. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pp. 15-29, June 1991.

[19] L. J. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, pp. 249-260, June 1992.

[20] L. J. Hendren, J. Hummel, and A. Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pp. 218-229, June 1994.

[21] L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Trans. Parallel Distrib. Syst.*, *1*, 1 (Jan.), 35-47.

[22] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, San Mateo, California, 1990.

[23] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation*, pp. 28-40, June 1989.

[24] A. S. Huang, G. Slavenburg, and J. P. Shen. Speculative disambiguation: A compilation technique for dynamic memory disambiguation. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pp. 200-210, April 1994.

[25] N. D. Jones and S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structure. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, pp. 66-74, 1982.

[26] M. Johnson. *Superscalar Microprocessor Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[27] 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.

[28] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, pp. 56-57, June 1993.

[29] D. E. Maydan, J. L. Hennessy, and M. S. Lam. Efficient and Exact Data Dependence Analysis. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pp. 1-14, June 1991.

[30] T. J. Marlowe, W. A. Landi, B. G. Ryder, J. D. Choi, M. G. Burke, and P. Carini. Pointer-induced aliasing: a clarification. *ACM SIGPLAN Notices*, *28*(9), pp. 67-70, September 1993.

[31] T. C. Mowry. *Tolerating latency through software-controlled data prefetching*. PhD thesis, Stanford University, March 1994.

[32] A. Nicolau. Run-time disambiguation: coping with statically unpredictable dependen-

cies. *IEEE Trans. Comput.*, *38*, 5 (May), 663-678.

[33] M. D. Smith. *Support for speculative execution in high-performance processors*. PhD thesis, Stanford University, November 1992.

[34] D. W. Wall. Limits of instruction-level parallelism. Research Report 93/6, Western Research Laboratory, November 1993.

[35] 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.