# Speculative Disambiguation: A Compilation Technique for Dynamic Memory Disambiguation

Andrew S. Huang*, Gert Slavenburg†, and John Paul Shen*
ahuang@ece.cmu.edu, gert@prpa.philips.com, shen@ece.cmu.edu

*Carnegie Mellon University
Dept. of Electrical and Computer Engineering
5000 Forbes Avenue
Pittsburgh PA, 15213
(412) 268-3601

†Philips Research Palo Alto
North America Philips Corporation
4005 Miranda Avenue
Palo Alto CA, 94304
(415) 354-0310

## Abstract

*Ambiguous memory references have always been one of the main sources of performance bottlenecks. Many papers have addressed this problem using static disambiguation. These methods work extremely well when the memory access pattern is linear and predictable. However they are ineffective when the memory access pattern is nonlinear or when the access pattern cannot be determined statically. For these difficult problems, this paper presents speculative disambiguation, a compilation technique for architectures supporting instruction level parallelism and either speculative execution or conditional execution (or both). This technique produces specialized code at compile time to disambiguate memory references at run time. It is shown that on machines with sufficient resources, the technique will always result in lower execution time. Speculative disambiguation has been implemented for a VLIW architecture with guarded execution. Preliminary results indicate that it can help bridge a significant fraction of the performance gap between a good and a perfect static disambiguator. Occasionally it can outperform the perfect static disambiguator.*

## 1.0 Introduction

Recent microprocessor announcements point to two trends. Processors will have increasing number of deeply pipelined functional units, and the gap between processor speed and memory speed will continue to widen. To increase resource utilization and to reduce the effects of long memory latencies, it is necessary to execute multiple memory operations in parallel and to execute them out of order. In order to do this, it is necessary to perform memory disambiguation to expose those memory references that are independent and therefore can be executed in parallel. Static memory disambiguation techniques have been quite effectively applied to many numeric applications [9]. However, disambiguating non-numeric applications and other difficult numeric applications remains elusive. We introduce *speculative disambiguation* (SpD), which takes advantage of speculative execution [10] and multiple functional units to speed up the execution of code containing ambiguous references. SpD is designed for architectures that support conditional execution, for example, architectures with guarded execution [10]. This is not an unreasonable requirement. A single-chip very long instruction word (VLIW) machine with support for guarded execution has already been fabricated [11]. Two other architectures contain features that are indicative of the move towards conditional execution architectures. In the ARM processor [4], each instruction has a condition field that specifies the necessary condition for its execution. The DEC Alpha [7] has conditional register move instructions in which a register move occurs only if a third register has certain values.

Section 2 introduces the problems of ambiguous memory aliases and summarizes current methods for dealing with them. Section 3 discusses machine model features that are relevant to SpD. Section 4 develops the SpD concept for an architecture that supports speculative or conditional execution. Section 5 presents implementation issues for SpD. Section 6 presents experimental data from our initial experiments with SpD. We conclude the paper in Section 7 with discussion and future areas of research.

## 2.0 The Memory Alias Problem

One of the biggest problems with program paralleliza-



1063-6897/94 $03.00 

tion concerns memory accesses. Because the addresses of memory locations being accessed are not known until the program is executed, the compiler must take the conservative approach and execute memory accesses in the sequential order of the source code. In Example 2-1, this would correspond to writing to *a[i]* before reading from *a[j]*. It may be desirable to execute memory references *out of order* (read from *a[j]* before writing to *a[i]*) to reduce execution time. However this would result in the wrong value being read from *a[j]* if it turns out that $i = j$ at run-time.

```
Example 2-1
   a[i] = ...
   x    = f( ...,a[j],... )
```

To prevent such errors, compilers usually place dependence arcs between such memory references to ensure that they are executed in the sequential order. These dependence arcs are *ambiguous*; they are there not because of dependence, but because of *possibility* of dependence. The load from *a[j]* and the store to *a[i]* are *ambiguously aliased* if there is a possibility that $i = j$ at run-time. *Memory disambiguation* is the task of performing analysis to determine if dependence actually exists for a pair of ambiguously aliased memory references.

For each pair of memory references, we define the *alias probability* to be the number of times that the pair refer to a common location divided by the number of times that the pair is executed. Like branch probabilities, alias probabilities can be collected through profiling, and they can vary with the input data set of a program. Quite often, they can be computed by the compiler using static analysis techniques.

## 2.1 Static Disambiguation

Most programs spend the majority of their execution time inside loops. Within these loops, most of the memory operations involve accessing array elements with a regular pattern. For this class of memory accesses, it is often possible to resolve at compile-time whether or not two memory accesses will ever reference the same address. This is called *static disambiguation* since the ambiguity of the alias is resolved at compile time. [3][9][13][16] discuss various static techniques for memory disambiguation. Most of these techniques work by constructing linear diophantine equations [2] that describe the access patterns of the memory operations. Some of them require information such as loop bounds. While static techniques have been found to be quite effective for many applications, they do not work under the following circumstances:

- Access pattern is non-linear. Example: FFT code accesses data in exponential order.
- Address for memory access is read out of another memory location. Example: Pointer dereferencing and use of index array.
- Information needed by the static disambiguator is not available. Example: Loop bounds passed as parameters.

## 2.2 Dynamic Disambiguation

An alternative is to resolve the aliasing uncertainty at run-time. This is known as *dynamic disambiguation*. Since memory addresses are known then, it would be simple to compare them at run-time to see if aliasing exists. To see how well a dynamic disambiguator performs compared to a static disambiguator, consider what happens when a static disambiguator is asked if two memory accesses alias. One of three answers is returned:

- No. Never alias.
- Yes. Alias at least once.
- Unknown. Unable to determine statically whether aliasing ever occurs.

A dynamic disambiguator can correctly determine aliasing for the "Unknown" cases. It can also improve program performance for the "Yes" cases where aliasing occurs only some of the time; i.e. the alias probability is greater than zero and less than one.

```
Example 2-2
   for i = 1 to 100 do
        a[2i] = ...
        y     = f( ...,a[i+4],... )
   end
```

In Example 2-2, the two references to *a[]* have an alias probability of 0.01. The static disambiguator would return "Yes" because it knows that the two alias at least once. As a result, the two statements are forced to execute sequentially even though they are independent for all iterations except the fourth iteration. Using a dynamic disambiguator would allow the two statements to execute in parallel for 99 of the 100 iterations.

## 2.3 Hybrid Disambiguation

Usually dynamic disambiguation is implemented strictly in hardware. However, dynamic disambiguation can be performed with some compile-time aid to reduce or relieve the hardware requirement. We call this *hybrid disambiguation*. The Multiflow TRACE [6], for example, has a memory disambiguation system that combines elements of static and dynamic techniques. The TRACE has a memory system made up of multiple memory banks. When a memory reference is issued to a bank, that bank is busy for some length of time during which it cannot accept another reference. A *bank conflict* occurs if a reference is issued to a bank that is still servicing a previous reference. Like other VLIWs, the TRACE is a statically scheduled

machine. But it also has a hardware mechanism for stalling the entire machine when memory bank conflict is detected. The compiler's static disambiguator is able to disprove memory bank conflicts for simple cases. For cases where the static disambiguator returns "Unknown," the compiler aggressively places the two accesses in a potentially conflicting schedule in the hope that no bank conflict will occur. At run-time, the the machine is stalled if a bank conflict occurs. Therefore the inclusion of the dynamic hardware mechanism allows the TRACE to squeeze some additional performance from the "Unknown" cases.

However, because the TRACE is statically scheduled, it does not take full advantage of dynamic (memory bank) disambiguation. Two ambiguously aliased references are not reordered even though doing so may result in greater performance. A dynamically scheduled machine, e.g. a superscalar processor, would do better since it could reschedule based on whether or not an alias exists. The Motorola 88110 [5] is such a processor. It is able to reorder loads and stores within the load/store unit, effectively performing disambiguation and rescheduling on the fly. The only problem is that the scope visible to a dynamic scheduler is typically a small number of instructions. To be able to consider a larger scope would require significant amount of hardware.

SpD is a compilation technique for supporting dynamic disambiguation. Because it is performed at compile-time, it is able to make use of the large scope visible to a compiler. It performs transformations on the code to anticipate either outcome of an ambiguous alias. It exploits the conditional execution mechanism and does not require any additional hardware support.

A similar technique is Nicolau's [14] *run-time disambiguation* (RTD), an extension to *trace scheduling* [8]. For each pair of ambiguously aliased memory references, RTD produces two paths where there was one before. On one of the paths, it is assumed that the two memory operations alias. On the other path it is assumed that the memory operations do not alias. When the program is executed, the addresses are compared *in software* to determine which path to take. Since an explicit branch may be taken as a result of the address comparison and incur a branch penalty, it is important to predict correctly which of the two paths will be executed more often. The less frequently executed path is then scheduled off-trace. SpD is similar to RTD. Potential explicit branches are replaced by guarded instructions, and no branch penalty is incurred. However, it is designed to work with architectures supporting speculative or guarded execution. Furthermore, its effectiveness does not necessarily depend on accurate branch prediction.

# 3.0 Machine Model

SpD is designed for architectures supporting instruction level parallelism (ILP). When SpD is applied to a pair of ambiguously aliased memory references within a basic block, it can potentially produce the following effects:

1. A single control flow path through the basic block is transformed into two paths, anticipating the two possible outcomes of an ambiguous alias.
2. The critical path length of each of the two new control flow paths is less than or equal to that of the original path through the basic block.
3. The total number of operations is increased.

Effect 2 is the desired effect. Effect 3 can hinder a program's performance unless there is sufficient machine parallelism. This could be in the form of a deep pipeline or multiple functional units.

Effect 1 can potentially cause a program to execute slower, due to the high cost of branch penalties in pipelined processors. Speculative execution can support the concurrent execution of operations from both paths. Conditional execution provides a way to execute both paths without incurring a branch penalty.

## 3.1 Speculative Execution

Given a condition $c$, which determines whether not not an operation $p$ should be executed, $p$ is *speculatively executed* if it is completed (see Figure 3-1) prior to the availability of $c$. In Figure 4-1, the execution of the subtract in $BB_3$ is determined by the result of the (greater than) comparison in $BB_1$. The subtract is speculatively executed if it is moved from $BB_3$ to $BB_1$ and executed before the comparison produces its result. Speculative execution produces three types of problems:

1. The operation that is speculated may overwrite a register needed on the other path. For example, the subtract may write its result to a register that is an operand of the multiply in $BB_2$.
2. The speculated operation may cause an exception. For example, if the multiply is moved from $BB_2$ to $BB_1$, it may cause an overflow that would not have happened if control is transferred to $BB_3$. Another type of exception that may be generated is page faults due to speculated loads.
3. The speculated operation modifies the system memory. For example, if the store $S_1$ in $BB_2$ is moved into $BB_1$, and control is then transferred to $BB_3$, it would have incorrectly written a value to memory.

An architecture that supports general speculative execution must provide mechanisms to deal with these three types of problems. For example, Smith et al. [17] proposes

the *boosting* concept, which handles type 1 problems with a shadow register file, type 2 with a shift buffer, and type 3 with store buffers. These extra buffers hold the *speculated state* of the machine until the condition is available, at which point the machine state is updated with the speculated state and exceptions are examined.

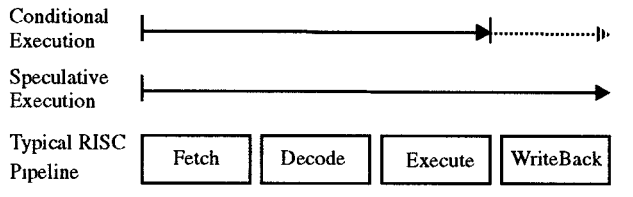



**Figure 3-1 Amount of speculation in two performance enhancing techniques.**

Speculative execution can greatly increase ILP. However, the hardware cost of supporting general speculative execution can be prohibitive.

### 3.2 Conditional Execution

Another form of speculation, known as *conditional execution*, provides some of the benefits of speculative execution at a lower cost. Each conditionally executed operation is predicated by a condition that determines its execution. As Figure 3-1 shows, a conditionally executed operation is speculatively fetched, decoded and executed. However it cannot proceed to the *write back* stage prior to the outcome of the condition that determines its execution. When the condition is available, either the instruction is allowed to complete (dashed arrow in Figure 3-1) or it is cancelled before the *write back* stage. Hence, there is no need to maintain a speculative state.

## 4.0 Speculative Disambiguation

In this paper the concept of SpD is developed for ILP architectures that support speculative or conditional execution. This section describes how speculative and conditional execution can be used to increase program performance and how they are used to implement SpD.

### 4.1 Program Model

In our program model, operations fall into two categories: those that have side effects and those that do not. An operation has *side effect* if it causes type 2 or type 3 problems as described in Subsection 3.1. To simplify the discussion of SpD, it is assumed that floating-point operations do not raise exceptions and loads do not cause page faults. The only operations that have side effects are memory stores.

Operations without side effects (type 1) can be speculatively executed; the register file holds the speculative state. Operations with side effects can be executed with *guarded execution*, which is a form of conditional execution. Each *guarded* operation receives a boolean value as an extra operand. Normally this is the result of a comparison (the condition) that determines if the operation should be executed. Thus, using the guard bit, the machine can simultaneously execute operations from multiple control flow paths and commit only the results from one of those paths. Instead of the basic block, the basic unit in the control flow graph is the *decision tree* [10], the largest group of basic blocks with a single entry point, multiple exit points and no backward edges. By executing one decision tree at a time, the machine is effectively executing all the paths through the tree.

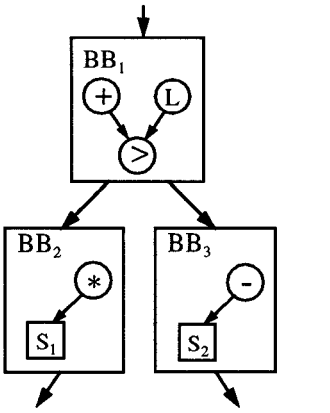



**Figure 4-1 Decision tree used in a guarded architecture.**

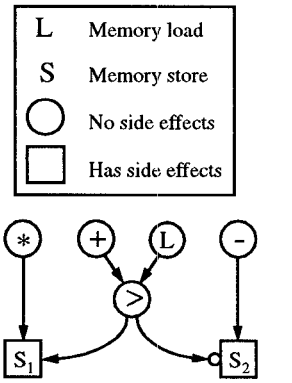



**Figure 4-2 Using guards to represent control dependences.**

Figure 4-1 shows a decision tree consisting of three basic blocks. In Figure 4-2, the control dependences have been converted to data dependences [1] through the use of guards. The vertical positions of the operations in the dataflow graph indicate the earliest possible issue times of the operations due to data dependence constraints. The operations in $BB_2$ and $BB_3$ that do not have side effects are now speculatively executed before the compare in $BB_1$. The two memory stores in $BB_2$ and $BB_3$ are now data dependent on the compare, which generates their guards. The bubble on the store $S_2$ indicates inversion of the guard value.

In essence, guarding eliminates the need to do explicit branches. The uncertainty of the branch direction is dealt with by speculating past the branch and appropriately guarding operations on both paths. Speculative execution shortens the critical path through the tree. Both features

contribute to higher overall performance.

### 4.2 Fundamental Concept

When SpD is applied to a pair of ambiguously aliased memory references, the disambiguator creates an additional copy of the code that is data dependent on the memory references. In one copy of the code, it is assumed that the addresses of the two references are the same. In the other copy of the code, it is assumed that they are different. In both copies, operations that do not have side effects are executed speculatively. Those that do are guarded by the result of comparing the two addresses. In the following subsections, the SpD concept is presented for the three types of memory dependences, i.e. RAW, WAR, and WAW dependences. Figure 4-3 contains the legend for Figure 4-4 through Figure 4-6.

**Figure 4-3 Legend.**

L Memory load
S Memory store
O No side effects
□ Has side effects
∘ Inversion of guard value
Data dependence
Ambiguous dependence
Added for spec disambig

### 4.3 Read After Write Dependence

By far the most critical type of memory dependence, in terms of performance, is the read after write (RAW) dependence. Figure 4-4 shows a fragment of code before and after speculative disambiguation.

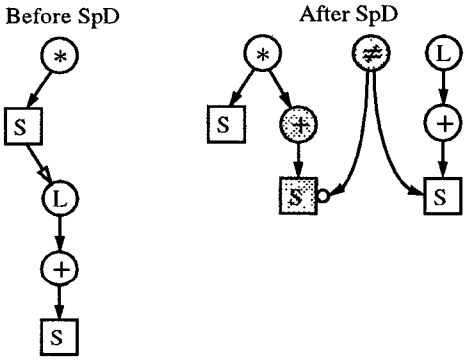



**Figure 4-4 Applying SpD to RAW dependence.**

Originally, the operations that are dependent on the load are prevented from executing early because of the potential dependence between the load and a previous store. Assuming that there is no alias allows the earlier execution of the load and its dependent operations. To guarantee correctness, it is necessary to guard any side effect operations with the result of the compare. If aliasing does occur, the execution time can still be shortened by forwarding the result of the multiply directly to the add, eliminating from the critical path the latencies of the store and the load.

Note that for both the case where the addresses alias and the case where they do not, the resulting code will always run faster provided that the machine has enough resources to accommodate the concurrent execution of the speculated operations.

The shortened critical paths come with the price of increased code size. In this case, SpD adds an address compare operation and replicates all operations that are data dependent on the load. The total cost is $1 + n_L$, where $n_L$ is the number of operations that are directly or indirectly data dependent on the load.

### 4.4 Write After Read Dependence

Figure 4-5 shows a situation in which there are three ambiguous dependences, two of which are RAW dependences and one is a write after read (WAR) dependence.

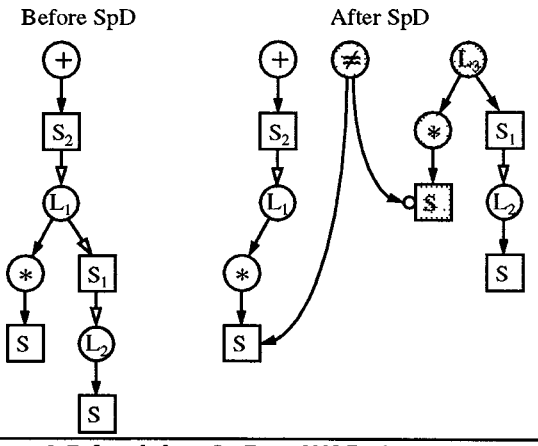



**Figure 4-5 Applying SpD to WAR dependence.**

Assume that the SpD guidance heuristic selects the WAR dependence for speculative disambiguation. The following would occur. The store $S_1$ and its dependent operations are allowed to move up past the load $L_1$. It is possible that the two memory operations alias and thus $S_1$ would destroy the original value at that location before $L_1$ has a chance to read it. To prevent this from happening, a new load $L_3$ is inserted before $S_1$, which reads from the same address as $S_1$. The value read is then used in the same computation as the value read by $L_1$. Thus there are two copies of the computation, guarded by opposite values of the address com-

parison.

Since $L_3$ accesses the same address as $S_1$, it is ambiguously aliased with every operation that $S_1$ is ambiguously aliased with. There would be an arc from $L_3$ to $L_2$ except that both operations are loads. There is no arc from $S_2$ to $L_3$ because there is no arc from $S_2$ to $S_1$

Generally, WAR dependences are not selected for SpD. This is because in order for SpD to be beneficial, the dependence chain starting from the store $S_1$ must be longer than the one starting from $L_1$. But $S_1$ is a store, so the only operations that can be dependent on it are other stores and loads, and they are ambiguously dependent on $S_1$. Any such memory operations are likely to be ambiguously aliased with other memory operations in the tree and may not be able to follow $S_1$ as it ascents past $L_1$.

WAR dependences also have the highest cost when SpD is applied. The additional operations created by SpD are the address comparison, the load to precede $S_1$, and all operations directly or indirectly data dependent on the load $L_1$. The total cost is $2 + n_L$.

### 4.5 Write After Write Dependence

In the case of a write after write (WAW) dependence, SpD allows the second store ($S_2$ in Figure 4-6) to be executed first. The result of the address comparison is used to guard the first store $S_1$, which is not executed if the addresses are the same. This is because it would have been overwritten by $S_2$ in the original code.

Similar to that of the WAR dependence, the benefit from applying SpD to a WAW dependence is small. However it is slightly greater than in WAR dependences. Let us assume the most common case where loads occur at the beginning of series of computation, and stores occur at the end. Here both memory operations are stores. Thus moving $S_2$ up past $S_1$ can potentially reduce the execution time by the latency of a store. In the WAR dependence (Figure 4-5), if $S_1$ were at the end of a series of computation and had no dependent operations, moving $S_1$ past $L_1$ would have no effect on the critical path.

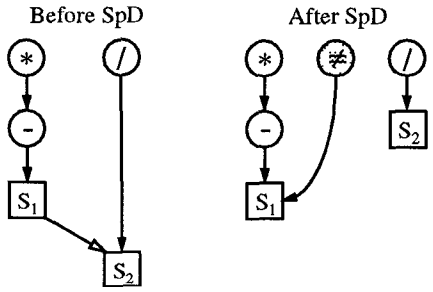



**Figure 4-6 Applying SpD to WAW dependence.**

There is another reason that SpD will tend to be applied to WAW dependences more than it is applied to WAR dependences. WAW dependences have the lowest cost associated with it. Only one address comparison operation is required.

### 4.6 Faulting Loads

Throughout this section, we made the simplifying assumption that loads do not cause page faults and can thus be speculatively executed. For many processors this is simply not the case. We now consider the effect of having loads that can cause page faults. For RAW dependences (see Figure 4-4,) the load that is executed speculatively in the transformed code is *always* executed in the original code. Thus any page fault that it causes would also occur in the original code, only that they now occur at an earlier time. For WAR dependences (Figure 4-5,) the new load $L_3$ access the same memory location as $S_1$, which is always executed in the original code. The introduction of $L_3$ causes page faults due to $S_1$ to occur earlier. It does not introduce any extraneous page faults. For WAW dependence, there are no loads involved; assuming a faulting load makes no difference. Thus our simplifying assumption is a reasonable one, since it has no impact on the effectiveness of SpD.

## 5.0 Compilation Issues

Speculative disambiguation is a code transformation technique and is meant to be incorporated into the compilation tool. This section presents some of the implementation issues relevant to SpD.

### 5.1 Profile Information

The idea of profiling programs is not new. Traditionally, it has been used to fine tune program performance. For SpD, the availability of profile information is highly desirable. Without profile information, it is still possible to determine which memory aliases are on the critical path. But it would be difficult to assess the relative benefits of applying SpD to each of those aliases. Profile information in the form of path probabilities, provides a way to compute the potential benefit from applying SpD to each memory alias on the critical path.

### 5.2 Application of Speculative Disambiguation

SpD can be applied to a program at two different points of the compilation process. It can be applied while the program is in its source form or after it has been converted to its intermediate representation. Each of these methods has certain requirements for supporting SpD.

To apply SpD at the level of the intermediate representation, it is necessary for the intermediate representation to have a basic unit that can represent multiple paths. The decision tree is an example of such a basic unit. Another is the hyperblock [12].

At the source level, the application of SpD creates two paths, on one of which, there has to be a way to inform the compiler to assume two references do not alias. This requirement can be met in the form of a compiler assertion or pragma.

```
Example 5-1
   *a    = ...
   ...   = *b
```

Example 5-1 shows an ambiguous RAW dependence, which after application of SpD, would look like Example 5-2.

```
Example 5-2
   if ( a != b )
   #pragma    depoff
         *a    = ...
         ...   = *b
   #pragma    depon
   else
         *a    = ...
         ...   = *b
   end
```

The pragmas *depoff* and *depon* inform the compiler that there are no memory dependences in the code between the two pragmas.

The drawback to performing SpD on source codes is that it is difficult to tell when it would be beneficial to apply SpD to an ambiguous alias, since it is impossible to tell if that alias will end up on the critical path after the source code has been compiled to the intermediate representation. In our first implementation of SpD, we have chosen to apply SpD at the level of decision trees.

## 5.3 Guidance Heuristic

Because speculative disambiguation has an undesirable side effect of increasing code size, it is important to be able to predict the usefulness of applying the method before actually applying it. To do this, it is important to be able to estimate the time it takes to execute a given tree. A basic block generally takes the same amount of time to execute. But a tree may take different amount of time to execute depending on the path that is taken through the tree. In our first implementation of SpD, we use a platform that provides path probabilities through profiling. Figure 5-1 shows the current heuristic.For a given tree, the heuristic iteratively applies SpD to the pair of ambiguous alias whose removal would result in the largest performance gain for the tree. Each application of SpD results in more code and possibly more ambigous aliases. Two paramters are used to prevent run-away code explosion. *MaxExpansion* places an upper bound on the amount of code expansion that is permissible. *MinGain* is the gain threshold for application of SpD. When there are no more aliases with predicted gain of at least *MinGain*, the heuristic stops.

The impact of doing SpD on a pair of references is predicted by the *Gain()* function. It is computed as the difference between the average execution time for the tree before and after removing the ambiguous dependence arc. The actual performance gain when SpD is applied may be slightly different due to the overhead of SpD. An example of this is the address comparison, which may end up on the critical path, reducing the benefit of SpD.

```
SpecDisambig( T, MaxExpansion, MinGain) {
  MaxSize ← TreeSize(T) * MaxExpansion
  S    ← CriticalAlias(T)
  while ( TreesSize(T) < MaxSize ∧ |S| > 0 )    {
    A ←
        x| (x ∈ S ∧ Gain(x) = max_{∀(y ∈ S)} Gain(y))
    if ( Gain(A) < MinGain)
      break
    T ← ApplySpD(T,A)
    S ← CriticalAlias(T)
  }
}
```

| | |
|---|---|
| *MaxExpansion* | *Maximum code expansion factor.* |
| *MinGain* | *Gain threshold for performing SpD.* |
| *TreeSize(T)* | *Returns the size of tree T in number of operations..* |
| *CriticalAlias(T)* | *Returns the set of ambiguous aliases that are on the critical paths of tree T.* |
| *ApplySpD(T,A)* | *Returns tree T after applying the SpD code transformation (Section 4.0) for alias A.* |
| *Gain(A)* | *Returns the predicted gain for removal of alias A.* |

**Figure 5-1 Guidance heuristic for SpD**

When SpD is applied to a tree, a control flow split is introduced due to the address compare operation. The branch probability for this split is exactly the alias probability of the memory references for which SpD is applied. As of this time, we have no means of obtaining the alias probability through profiling. Therefore we assume an alias probability of 0.1. For the first iteration of the heuristic, this assumption has no negative effect on the performance of SpD, since the critical paths for both outcomes of the alias have been shorten. However, for each successive iterations, the *Gain()* function is less and less accurate because path probabilities are less accurate. This may cause the heuristic to make suboptimal decisions.

## 6.0 Initial Experiments

### 6.1 Platform

Our initial experiments with SpD are performed with the compilation tools developed at Philips Research Palo Alto (in Palo Alto, California) for the LIFE VLIW architecture [11]. The LIFE architecture consists of multiple pipelined functional units connected to a global register file. The pipeline stages of all functional units operate synchronously and employ guarded execution. In each cycle, a very long instruction word consisting of multiple operations is issued to the functional units. The position of the operation within the instruction determines the functional unit that executes it. Figure 6-1 shows a generic LIFE processor with *n* functional units. Each functional unit reads two source operands and one guard value from the global register file and writes back a result in every cycle. Special functional units such as memory units may have additional ports to the memory.

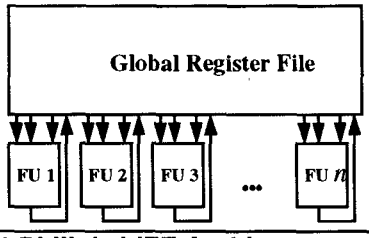



**Figure 6-1 Philip's LIFE Architecture with guarded execution.**

An 50 MHz chip with two ALUs, one branch unit and one memory unit has already been designed and fabricated [11]. In addition, there exists a suite of software tools for exploring the processor design space. These tools are configured with a machine description file, which can be used to specify an arbitrary implementation of the architecture. Using the machine description file, it is possible to compile, schedule and produce cycle-level simulation results for that implementation.

A subset of the LIFE tools are used in our experiments. These tools include:

- An optimizing C compiler which generates decision trees.
- A cycle-level infinite machine simulator that takes a program in the form of decision trees and produces program results as well as a program profile.
- A scheduler that schedules decision trees for constrained resource machines.

In addition, for this paper a postpass disambiguator has been developed which performs both static disambiguation and speculative disambiguation on decision trees. Static disambiguation is implemented with the GCD test and the Banerjee inequalities [3]. Although these are not the most sophisticated tests available, Goff et al. [9] have shown that even simple tests that are less general than the GCD/Banerjee combination are sufficient for disproving ambiguous aliases in most programs. Speculative disambiguation is implemented using the guidance heuristic outlined in Subsection 5.3.

The experiments are performed as follows. The C compiler generates decision trees from the benchmark source codes. The decision trees are then processed by the disambiguator before being fed into the simulator, which produces an execution cycle count. It also produces the program output, which is used to validate the correctness of the decision trees. The execution time computed by the simulator is for an infinite machine. However, we are able to schedule for any machine resource configuration and get either estimates of execution time or cycle-accurate simulation of actual execution.

### 6.2 Machine Model

The experiments use a machine description file that describes LIFE implementations with one to eight universal functional units. Each universal functional unit is capable of executing any type of operation. Table 6-1 lists the operation latencies used. To evaluate the effects of memory latency on SpD, we gathered two sets of data, one using a two-cycle memory latency and the other a six-cycle memory latency.

**Table 6-1: Operation latencies**

| Operation | Latency (cyc) |
|---|---|
| Integer multiplies | 3 |
| Integer and FP divides | 7 |
| FP compares | 1 |
| Other ALU operations | 1 |
| Other FPU operations | 3 |
| Memory loads and stores | 2 or 6 |
| Branches | 2 |

### 6.3 Initial Experimental Data

For our initial experiments, we used a set of benchmarks that are difficult to disambiguate statically. It includes six programs from *Numerical Recipes in C* [15] (NRC), seven programs from the Stanford Integer Benchmarks (StanfInt), and one program from SPECint 92. Table 6-2 lists these programs and their descriptions.

The programs from NRC are quite challenging for the static disambiguator because they contain a large number of pointer dereferences, most of which are due to arrays

passed into procedures. With StanfInt, three of the programs were not affected by SpD at all. We will merely state this rather than present the data for those programs.

**Table 6-2: Benchmark Descriptions**

| Benchmark | Lines | Description |
|---|---|---|
| adi, NRC | 143 | Alternating direction implicit method for partial differential equations. |
| bcuint, NRC | 63 | Bicubic interpolation. |
| fft, NRC | 73 | Fast fourier transform. |
| moment, NRC | 29 | Moments of distribution. |
| smooft, NRC | 219 | Smoothing of data. |
| solvde, NRC | 381 | Relaxation method for two point boundary value problems. |
| perm, StanfInt | 88 | Recursive permutation program. |
| queen, StanfInt | 97 | Eight queens problem. |
| quick, StanfInt | 94 | Quicksort. |
| tree, StanfInt | 117 | Treesort. |
| espresso, SPEC | 14,838 | Boolean function minimization. |

Table 6-3 shows a breakdown of the types of ambiguous aliases that SpD is applied to for each benchmark. For example, for the two-cycle memory latency model, SpD is applied to RAW dependences 16 times in the benchmark adi. The SpD heuristic does not discriminate against any particular type of dependence. It makes its selection solely on the basis of potential performance gain. The statistics confirm our intuition that SpD would benefit RAW dependences much more than it would WAR or WAW dependences. For this particular set of benchmarks, it does not benefit WAR dependences at all.

**Table 6-3: Frequency of SpD application by dependence type.**

| | 2 Cycle Memory Latency | | | 6 Cycle Memory Latency | | |
|---|---|---|---|---|---|---|
| Program | RAW | WAR | WAW | RAW | WAR | WAW |
| adi | 16 | 0 | 2 | 19 | 0 | 2 |
| bcuint | 6 | 0 | 0 | 8 | 0 | 1 |
| fft | 4 | 0 | 2 | 4 | 0 | 1 |
| moment | 8 | 0 | 3 | 7 | 0 | 3 |
| smooft | 1 | 0 | 2 | 1 | 0 | 2 |
| solvde | 13 | 0 | 1 | 11 | 0 | 2 |
| perm | 1 | 0 | 0 | 2 | 0 | 1 |
| queen | 1 | 0 | 1 | 1 | 0 | 5 |
| quick | 0 | 0 | 1 | 0 | 0 | 1 |
| tree | 1 | 0 | 1 | 1 | 0 | 1 |
| espresso | 36 | 0 | 8 | 40 | 0 | 10 |
| TOTAL | 87 | 0 | 22 | 94 | 0 | 30 |

To evaluate the effectiveness of SpD, we define the four disambiguators listed in Table 6-4. PERFECT is an omniscient static disambiguator. It represent the limit of static disambiguation. The performance numbers for PERFECT are obtained by running the simulator twice. During the first run, the simulator records the number of times that each pair of ambiguously aliased memory references actually access the same address. If this number is zero, the dependence arc between the two references is *superfluous*. All superfluous arcs are removed prior to the second run. Obviously the set of superfluous arcs for a given benchmark may be a function of the data set. However, the set of arcs that would be removed by a perfect disambiguator is always a subset of the superfluous arcs obtained by the simulator. Therefore, our PERFECT numbers are on the optimistic side. Its performance is always as good as or better than that of a true perfect static disambiguator.

**Table 6-4: Disambiguators used in experiments.**

| Disambiguator | Types of disambiguation performed. |
|---|---|
| NAIVE | None |
| STATIC | Static (GCD/Banerjee) |
| SPEC | Static followed by SpD. |
| PERFECT | Perfect static. |

**Figure 6-2 Speedup over the NAIVE disambiguator for a 5 FU machine.**

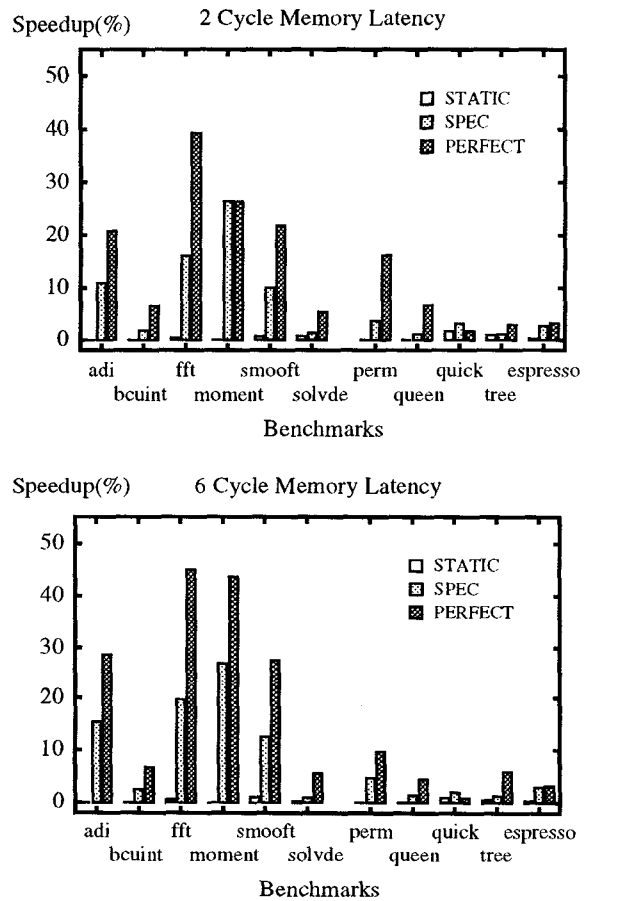
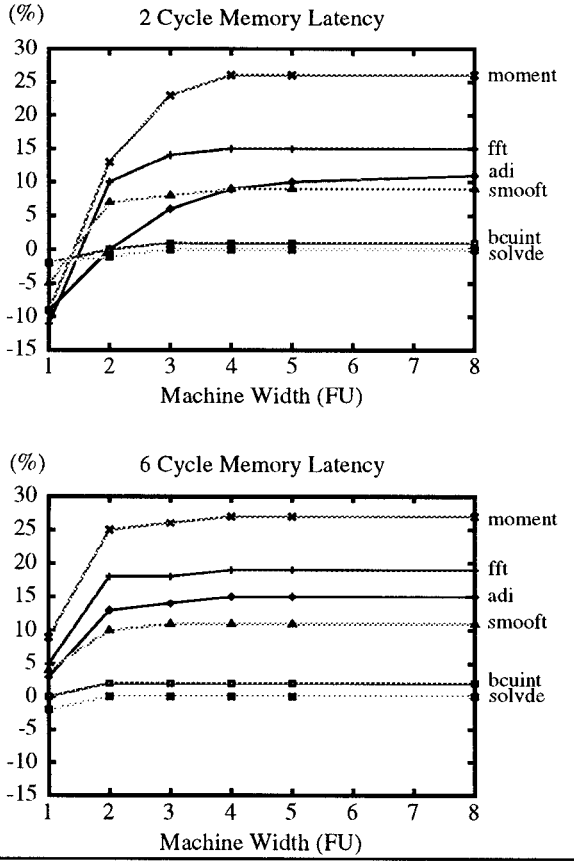


Figure 6-2 shows the relative performance of the disambiguators for 2 and 6 cycle memory latencies. For each group of three bars, the first bar (starting from the left) is the performance of STATIC relative to NAIVE. This is computed as the cycle count of the benchmark when processed by NAIVE over cycle count of the benchmark when processed by STATIC, minus one. The second bar shows the performance of SPEC relative to NAIVE. The third bar shows the performance of PERFECT relative to NAIVE. Note that for the benchmark *quick*, SPEC outperforms PERFECT, despite the code overhead incurred by SpD.

Another way to look at the data is to see how much additional perfermance can be expected from SpD if one is already using static disambiguation. Figure 6-3 shows the contribution of SpD to program speedup as functions of machine width and memory latency.

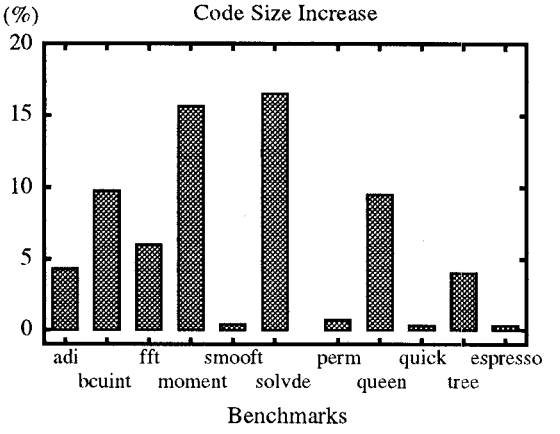
**Figure 6-3 Speedup of SPEC over STATIC**




Only the data for the NRC benchmarks are presented to avoid cluttering the graphs. Because SpD produces additional code, it will actually slow down machines with insufficient resource. Different program requires different amount of resource to benefit from SpD. With a two cycle memory latency, most programs need between two and three functional units to take advantage of SpD. When the memory latency is increased to six cycles, most programs will benefit from SpD with as few as one functional unit. It is not that fewer functional units are needed to benefit from SpD, but that ambiguous aliases hinder performance much more as memory latency increases.

Figure 6-4 shows the amount of code increase due to SpD for 2 cycle memory latency.

**Figure 6-4 Code size increase due to SpD.**




Code size is measured in term of number of operations, rather than number of VLIW instructions. We feel that this is more meaningful since it does not count no-ops. Note that this would correspond to superscalar code size. The increases in code sizes is quite reasonable for the mount of speedup achieved. *Smooft* obtained a speedup of close to 10% at a cost of 0.5% increase in code size. On the other hand, a 16% increase in the code size of *solvde* result in less than 1% speedup. This poor cost/benefit ratio can be improved by making better use of profile information in the guidance heuristic.

## 7.0 Conclusion

In this paper, we have presented a compilation technique for disambiguating difficult memory aliases. Speculative disambiguation performs code transformation at compile-time to resolve ambiguous aliases at run-time. Such code transformations can result in better performance on machines with adequate machine parallelism.

The key to this technique is architectural support for the concurrent execution of multiple paths. This can be provided in the form of speculative execution or conditional execution, or a combination of both. Although this paper describes SpD for the LIFE model of speculative execution and guarding, it should be a simple matter to

apply this compilation technique to other models of speculative execution.

Our initial experiments on the LIFE architecture shows that there is indeed a performance gap between what a realistic static disambiguator can achieve and what a perfect disambiguator can achieve. SpD is a compile-time transformation that is able to bridge part of this gap at the cost of increased code size, the effects of which can be mitigated through the use of profile information.

We see two major areas for further research in this topic. One is the effect of scope on SpD. Our experience with the Stanford Integer Benchmarks shows that the trees in integer programs are often too small to have pairs of ambiguous memory references. Enlarging trees through code replication techniques such as *grafting* [11] should expose more opportunities for applying SpD.

Another interesting area is the application of SpD to multiple pairs of references. The one-at-a-time approach currently used can result in up to $2^n$ copies of code for the $n$ pairs of references in a tree. A more intelligent way to handle this situation may be to use alias probabilities from profiling or correlations of alias probabilities to generate one version of code for the most likely outcome. Then generate another version of the code that will execute correctly, albeit more slowly, for the other $2^n$-1 outcomes.

There is great potential in applying SpD to superscalar processors. Most current superscalar processors still execute loads and stores sequentially. To achieve higher levels of performance in future superscalars, parallel and out of order execution of loads and stores is essential. SpD can expose more load/store parallelism and alleviate some of the unnecessary memory dataflow bottlenecks due to assumed but false memory dependences.

## 8.0 Acknowledgements

The authors thank the anonymous referees for their constructive comments. We also thank Yen Lee at Philips Research Palo Alto, for his help with the LIFE software tools. This work has been supported by Philips and ONR #0001491-J-1518.

## References

[1] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. "Conversion of Control Dependence to Data Dependence." *10th ACM Symp. on Prin. of Prog. Lang.* 1983.

[2] G. E. Andrews. *Number Theory.* Philadelphia PA: W B Saunders, 1971.

[3] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic, 1988.

[4] B. Case, "ARM Architecture Offers High Code Density." *Microprocessor Report.* Dec. 18, 1991.

[5] B. Case, "Superscalar Techniques: SuperSPARC vs. 88110." *Microprocessor Report.* Dec. 4, 1991.

[6] R.P. Colwell, R.P. Nix, J.J. O'Donnell, D.B. Papworth, P.K. Rodman. "A VLIW Architecture for a Trace Scheduling Compiler." *Proc. of ASPLOS II.* October, 1987.

[7] Digital Equipment Corporation. *Alpha Architecture Handbook.* 1992.

[8] J. A. Fisher. "Trace Scheduling: A Technique for Global Microcode Compaction." *IEEE Trans on Computers.* July 1981.

[9] G. Goff, K. Kennedy, C. Tseng. "Practical Dependence Testing." *Proc of the ACM SIGPLAN '91 Conf on PLDI.* June, 1991.

[10] P. Y. T. Hsu and E. S. Davidson. "Highly Concurrent Scalar Processing." *Proc of the 13th Symp on Comp. Arch.* 1986.

[11] J. Labrousse. G. A. Slavenburg. "A 50 MHz Microproecssor with a VLIW Architecture." *ISSCC '90.* 1990.

[12] S A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, R. A. Bringmann. "Effective Compiler Support for Predicated Execution Using the Hyperblock" *Proc of MICRO-25.* December 1992.

[13] D. E. Maydan, J. L. Hennessy, and M. S. Lam. "Efficient and Exact Data Dependence Analysis." *Proc of the SIGPLAN 1991 PLDI.* 1991.

[14] A. Nicolau. "Run-Time Disambiguation: Coping with Statically Unpredictable Dependencies." *IEEE Trans on Computers.* May 1989.

[15] W. H. Press, et al. *Numerical Recipes in C.* Cambridge University Press, New York, 1988.

[16] W. Pugh. "A Practical Algorithm for Exact Array Dependence Analysis." *CACM.* August 1992.

[17] M. D. Smith, M. Horowitz, M. S. Lam. "Efficient Superscalar Performance Through Boosting." *Proc of ASPLOS V.* September 1992.