SMARQ: Software-Managed Alias Register Queue for Dynamic Optimizations

Cheng Wang  Youfeng Wu   Hongbo Rong  Hyunchul Park
Programming Systems Lab
Microprocessor and Programming Research
Intel Labs
{cheng.c.wang, youfeng.wu, hongbo.rong, hyunchul.park}@intel.com

Abstract

Traditional alias analysis is expensive and ineffective for dynamic optimizations. In practice, dynamic optimization systems perform memory optimizations speculatively, and rely on hardware, such as alias registers, to detect memory aliases at runtime. Existing hardware alias detection schemes either cannot scale up to a large number of alias registers or may introduce false positives. Order-based alias detection overcomes the limitations. However, it brings considerable challenges as how software can efficiently manage the alias register queue and impose restrictions on optimizations. In this paper, we present SMARQ, a Software-Managed Alias Register Queue, which manages the alias register queue efficiently and supports more aggressive speculative optimizations. We conducted experiments with a dynamic optimization system on a VLIW processor that has 64 alias registers. The experiments on a suite of SPECFP2000 benchmarks show that SMARQ improves the overall performance by 39% as compared to the case without hardware alias detection. By scaling up to a large number (from 16 to 64) of alias registers, SMARQ improves performance by 10%. Compared to a technique with false positives (similar to Itanium), SMARQ improves performance by 13%. To reduce the chance of alias register overflow, the novel alias register allocation algorithm in SMARQ reduces the alias register working set by 74% as compared to a straightforward alias register allocation based on program order.

1. Introduction

Dynamic optimization systems perform optimizations at runtime. Thus it is crucial to keep under control the time spent on the optimizations. It is difficult and expensive to perform traditional memory alias analysis at runtime [13]. First, the dynamic optimizer may not have source level information, such as data type, array and subscript information. Second, for dynamic optimizations the alias analysis time is part of the execution time, and the expensive alias analysis can seriously impact the overall performance [13]. On the other hand, memory alias information is critical to the effectiveness of optimizations, especially for in-order processors [3]. Due to this dilemma, dynamic optimizations on in-order processors usually use simple or speculative alias analysis algorithms [1, 14] and rely on hardware (HW) to detect aliases at runtime. In this way, they can optimize memory accesses speculatively [6, 10].

Figure 1 shows the overall framework for a dynamic optimization system with HW alias detection. The application code runs on top of the system, which dynamically optimizes the application code to run on the CPU through an optimizer. The optimizer may assume no alias for memory operations that are unlikely to alias to each other. When the optimized code runs, the alias HW in CPU may detect the aliases for these operations and raise an alias exception. The optimized code is put into atomic regions for speculative execution [10] such that at the entry of each atomic region execution, the atomicity HW in CPU will create a checkpoint for rolling back the region in case of alias exception. The runtime module in the dynamic optimization system will catch the alias exceptions and trigger the optimizer to re-optimize the region conservatively; this time it assumes the two memory operations that just triggered the exception are always aliased. All the HW interrupts, exceptions and memory consistency violations are also caught by the runtime module to trigger region rollbacks [11].

10% Compared to a technique with false positives (similar to Itanium), SMARQ improves performance by 13%. To reduce the chance of alias register overflow (i.e. running out of alias registers), the novel alias register allocation algorithm in SMARQ reduces the alias register working set by 74% as compared to a straightforward alias register allocation based on program order.

Figure 1: A Dynamic Optimization System with HW Alias Detection

Existing HW alias detection schemes suffer from several problems. HW alias detection on Transmeta Efficeon [6] is not scalable and cannot support more than 15 alias registers. HW alias detection on Itanium [2] (which is called Advanced Load Address Table or ALAT) may introduce false positives and cannot detect aliases between stores. Order-based alias detection overcomes the limitations in Efficeon and Itanium. However, it brings considerable challenges to the efficient software management of the alias register queue and imposes restrictions on optimizations. (We will describe all of them in details in Section 2). In this paper, we present SMARQ, a Software-Managed Alias Register Queue for speculative dynamic optimization, which overcomes all the problems in the existing HW alias detection schemes. We conducted experiments with a dynamic binary translation/optimization system on a VLIW processor that has 64 alias registers. The experiments on a suite of SPECFP2000 benchmarks show that optimizations with SMARQ improve the overall performance by 39% as compared to those without hardware alias detection support. By scaling up the alias register number from 16 to 64, SMARQ improves performance by 10%. Compared to a technique with false positives (similar to Itanium), SMARQ improves performance by 13%. To reduce the chance of alias register overflow (i.e. running out of alias registers), the novel alias register allocation algorithm in SMARQ reduces the alias register working set by 74% as compared to a straightforward alias register allocation based on program order.

The rest of the paper is organized as follows. In Section 2, we present the motivation of the work. In Section 3, we use several examples to show the architectural features used in SMARQ. In Section 4, we develop the compiler analysis framework for identi-
fying the alias detection constraints. In Section 5, we describe our fast alias register allocation algorithm to satisfy all alias detection constraints. We present our experiments in Section 6 and discuss the related works in Section 7. At last, we conclude the paper and point out the future direction in Section 8.

2. Motivation

In this section, we illustrate the concept of HW alias detection. Then we show the HW alias detection in Efficeon and Itanium, and point out their weaknesses. We motivate our solution that addresses the weaknesses.

2.1 HW Alias Detection

Figure 2 shows an example for HW alias detection. The code in Figure 2 (a) is speculated into the code in Figure 2 (b). For simplicity, the only optimization applied here is reordering the memory accesses such that loads are performed as early as possible. Since it is unlikely that load M1 and store M2 alias to each other, M1 is speculated reordered above M2 after the optimization. Also, M0 is reordered below M2. For HW alias detection between M1 and M2, the optimizer needs to annotate M1 to set an alias register AR0 and annotate M2 to check AR0 for aliasing. Assume each memory access is 4 bytes. Then during the execution, M1 will set AR0 with its memory access range [r0, r0+3]. Later M2 will check this range in AR0 against its own memory access range [r0, r0+3]. If the two ranges overlap, an alias exception is raised.

(a) Original Program
M0: st [r0+4] = …
M1: … = ld [r1]
M2: st [r0] = …
M3: … = ld [r2]

(b) Optimized Program
M0: … = ld [r2]
M1: … = ld [r1]
M2: st [r0] = …
M3: st [r0+4] = …

Figure 2: A HW Alias Detection Example

2.2 Efficeon HW Alias Detection

A memory operation may need to check multiple alias registers. For example in Figure 2, M0 needs to check 2 alias registers, AR0 and AR1. Transmeta Efficeon [6] uses a bit-mask in the instruction to specify the individual alias registers that need to be checked. There is limited encoding space for the bit-mask in an instruction. For this reason, the alias register file cannot scale up to a large number of alias registers. In the current encoding scheme, Efficeon cannot support more than 15 alias registers. This is tolerable in Efficeon where an atomic region is typically small, with only about 30 x86 instructions on average. However, large scheduling/optimization regions are critical for achieving good performance on in-order processors [15, 19]. Since speculative optimization with large regions will inevitably use more alias registers, the scalability of alias register support will be a serious issue. In our experiment with a dynamic binary translation/optimization system on a VLIW processor, we see performance improvement for anmp benchmark in SPEC2000 by 30% by using 64 alias registers instead of 16 alias registers.

2.3 Itanium HW Alias Detection

Itanium memory alias detection [2] (which is called Advanced Load Address Table or ALAT) requires stores to automatically check all the alias registers set by previous advanced loads (i.e. ld.a instruction) without the need to specify the individual alias registers that need to be checked. Unfortunately, that may lead to false positives in the alias detection, which is a serious issue in dynamic optimization (Note that Itanium was not designed for dynamic optimization). Figure 3 shows the Itanium alias detection for the example shown in Figure 2. M2 does not need to check M1 (i.e. check AR1) for aliasing, instead, Itanium ALAT hardware checks M2 against all alias registers. This would result in a false positive if M1 aliases with M2. Moreover, Itanium memory alias detection cannot detect aliases between stores, and therefore, it does not allow speculative reordering between stores. However, our experiments show that speculative reordering between stores is a desirable optimization: it can improve performance by 13% for mesa benchmark in SPEC2000.

(a) Optimized Program
M0: … = ld [r2] // set AR0
M1: … = ld [r1] // set AR1
M2: st [r0] = … // check AR0, AR1
M3: st [r0+4] = … // check AR0, AR1

Figure 3: Itanium Alias Detection for Optimization in Figure 2

2.4 Order-Based HW Alias Detection

To overcome the scaling bottleneck, prevent the false positives and allow aggressive optimizations (like reordering stores), we can adopt the idea of order-based memory alias detection [4], which was used in Memory Order Buffer (MOB) in out-of-order processor to detect aliases between memory operations reordered by SW. In order-based memory alias detection, the alias registers are organized into an ordered circular queue (called alias register queue in this paper). Each memory operation is allocated an alias register, whose order matches the original program execution order. Figure 4 shows how the order-based alias detection works for the example shown in Figure 2. The alias register numbers 0-3 (see the second column of the code in Figure 4 (a)) corresponding to alias registers AR0-AR3. They are allocated to memory operations M0-M3 respectively in their original program execution order. When a memory operation executes, HW will set its memory access range against the alias register queue set by previously executed instructions for aliasing. Here by “alias register ARx is later than alias register ARy”, we mean x>y. So when M3 executes, the alias registers will have contents as shown in Figure 4 (b), and HW will only check AR3 but not AR1 because in the alias register queue, AR1 is earlier than the alias register allocated to M2 (i.e. AR2). HW also auto-
matically marks the alias registers that are set by loads so that later loads do not check against them. So in this example, \( M_1 \) will not check \( M_1 \) for aliasing (i.e. not check \( AR3 \)). In general, order-based alias detection can guarantee to detect all the aliases between reordered memory operations without any false positive. Table 1 shows the comparison between order-based alias detection and alias detection on Efficeon and Itanium.

### Table 1: Comparison between different HW Alias Detections

<table>
<thead>
<tr>
<th>Features</th>
<th>Efficeon</th>
<th>Itanium</th>
<th>Order-Based</th>
</tr>
</thead>
<tbody>
<tr>
<td>Features</td>
<td>bit-mask</td>
<td>ALAT</td>
<td>ordered queue</td>
</tr>
<tr>
<td>Scalability</td>
<td>Poor</td>
<td>Good</td>
<td>Good</td>
</tr>
<tr>
<td>False positive</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Detect alias between stores</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Even though the order-based alias detection can overcome all the problems with Efficeon and Itanium, it has three serious drawbacks. First, it is very inefficient in alias register usage by allocating each memory operation an alias register in program order. Second, it may perform many unnecessary alias detections, which costs energy. For example in Figure 4, the compiler analysis may find that \( M_0 \) and \( M_2 \) never alias to each other. So \( M_0 \) does not need to check \( M_2 \) (i.e. check \( AR2 \)) for aliasing, even though they are reordered in optimization. Moreover, allocating alias registers in program order can only detect aliases if speculative memory reordering is the only optimization applied, but not if there are other optimizations such as speculative load/store elimination, which may need alias detection between memory operations that are not re-ordered. For example in Figure 5 (a), \( M_1 \) and \( M_2 \) access the same memory location so we speculate eliminate the load at \( M_1 \) by forwarding data from \( M_1 \) to \( M_2 \) (see Figure 5 (b)), assuming there is no alias between \( M_1 \) and \( M_2 \). To ensure correctness, \( M_1 \) needs to check \( M_1 \) for aliasing, even though they are not reordered in the optimization.

**2.5 SMARQ: Software-Managed Alias Register Queue**

To reduce unnecessary alias register usage in order-based alias detection and keep the alias register allocation algorithm to be fast and suitable for dynamic optimizations, SMARQ uses a constraint order among the memory operations derived from compiler analysis to enforce the order among the registers assigned to the memory operations. Our constraints not only make sure that all required alias detections are performed, but also avoid any unnecessary alias check that may lead to false positive alias exception. Based on the constraint graph, SMARQ integrates alias register allocation and instruction scheduling in a single pass, and leverage a few architecture features to achieve the above objectives. SMARQ improves over the order-based alias detection approaches in following way:

- reducing unnecessary alias detection between reordered memory operations (between \( M_0 \) and \( M_2 \) in Figure 4)
- performing the necessary alias detection between even non-reordered memory operations for speculative load/store eliminations (between \( M_1 \) and \( M_2 \) in Figure 5)

Below we will first describe the architectural features used in SMARQ. Then we will show the two components in our solution in details: constraint analysis and alias register allocation algorithm.

### 3. SMARQ Architectural Features

In this section, we introduce the key architectural features in SMARQ. The compiler analysis can leverage to allocate alias registers efficiently. These features include 1) protection (\( P \)) and check (\( C \)) bits; 2) alias register rotation; 3) alias moving. These features are useful for avoiding unnecessary alias detection, reducing alias register overflow and preventing false positive in the alias detection.

#### 3.1 Avoid Unnecessary Alias Detection with P/C Bits

To avoid unnecessary alias detection, in addition to the alias register number, we allow the compiler to set a \( P \) bit to a memory operation that needs to set an alias register, and a \( C \) bit to a memory operation that needs to check alias registers. So we get:

\[
\text{[ORDERED-ALIAS-DETECTION-RULE]} \quad \text{Memory operation } X \text{ checks memory operation } Y \text{ for memory aliasing if and only if:}
\]

- \( Y \) precedes \( X \) in the optimized program execution and,
- \( X \) has \( C \) bit and,
- \( Y \) has \( P \) bit and,
- The alias register allocated to \( X \) is not later than the alias register allocated to \( Y \) in the alias register queue.

For a memory operation with both \( P \) bit and \( C \) bit, the memory operation will check alias registers before setting the alias register to prevent the alias detection on itself. As an example, Figure 6 shows the order-based alias detection with \( P/C \) bit for the optimization shown in Figure 2. \( M_1 \) and \( M_2 \) do not need to check any earlier executed memory operation for memory aliasing. So \( M_1 \) and \( M_2 \) only set alias register numbers 1 and 0 respectively with \( P \) bit and, \( C \) bit for memory aliasing if and only if:

- \( P \) bit and \( C \) bit, the memory
- \( Y \) precedes \( X \) in the optimized program
- \( X \) has \( C \) bit and
- \( Y \) has \( P \) bit and
- The alias register allocated to \( X \) is not later than the alias register allocated to \( Y \) in the alias register queue.

For a memory operation with both \( P \) bit and \( C \) bit, the memory operation will check alias registers before setting the alias register to prevent the alias detection on itself. As an example, Figure 6 shows the order-based alias detection with \( P/C \) bit for the optimization shown in Figure 2. \( M_1 \) and \( M_2 \) do not need to check any earlier executed memory operation for memory aliasing. So \( M_1 \) and \( M_2 \) only set alias register numbers 1 and 0 respectively with \( P \) bit and, \( C \) bit for memory aliasing if and only if:

- \( P \) bit and \( C \) bit, the memory
- \( Y \) precedes \( X \) in the optimized program
- \( X \) has \( C \) bit and
- \( Y \) has \( P \) bit and
- The alias register allocated to \( X \) is not later than the alias register allocated to \( Y \) in the alias register queue.

For a memory operation with both \( P \) bit and \( C \) bit, the memory operation will check alias registers before setting the alias register to prevent the alias detection on itself. As an example, Figure 6 shows the order-based alias detection with \( P/C \) bit for the optimization shown in Figure 2. \( M_1 \) and \( M_2 \) do not need to check any earlier executed memory operation for memory aliasing. So \( M_1 \) and \( M_2 \) only set alias register numbers 1 and 0 respectively with \( P \) bit and, \( C \) bit for memory aliasing if and only if:

- \( P \) bit and \( C \) bit, the memory
- \( Y \) precedes \( X \) in the optimized program
- \( X \) has \( C \) bit and
- \( Y \) has \( P \) bit and
- The alias register allocated to \( X \) is not later than the alias register allocated to \( Y \) in the alias register queue.

For a memory operation with both \( P \) bit and \( C \) bit, the memory operation will check alias registers before setting the alias register to prevent the alias detection on itself. As an example, Figure 6 shows the order-based alias detection with \( P/C \) bit for the optimization shown in Figure 2. \( M_1 \) and \( M_2 \) do not need to check any earlier executed memory operation for memory aliasing. So \( M_1 \) and \( M_2 \) only set alias register numbers 1 and 0 respectively with \( P \) bit and, \( C \) bit for memory aliasing if and only if:

- \( P \) bit and \( C \) bit, the memory
- \( Y \) precedes \( X \) in the optimized program
- \( X \) has \( C \) bit and
- \( Y \) has \( P \) bit and
- The alias register allocated to \( X \) is not later than the alias register allocated to \( Y \) in the alias register queue.

For a memory operation with both \( P \) bit and \( C \) bit, the memory operation will check alias registers before setting the alias register to prevent the alias detection on itself. As an example, Figure 6 shows the order-based alias detection with \( P/C \) bit for the optimization shown in Figure 2. \( M_1 \) and \( M_2 \) do not need to check any earlier executed memory operation for memory aliasing. So \( M_1 \) and \( M_2 \) only set alias register numbers 1 and 0 respectively with \( P \) bit and, \( C \) bit for memory aliasing if and only if:

- \( P \) bit and \( C \) bit, the memory
- \( Y \) precedes \( X \) in the optimized program
- \( X \) has \( C \) bit and
- \( Y \) has \( P \) bit and
- The alias register allocated to \( X \) is not later than the alias register allocated to \( Y \) in the alias register queue.

For a memory operation with both \( P \) bit and \( C \) bit, the memory operation will check alias registers before setting the alias register to prevent the alias detection on itself. As an example, Figure 6 shows the order-based alias detection with \( P/C \) bit for the optimization shown in Figure 2. \( M_1 \) and \( M_2 \) do not need to check any earlier executed memory operation for memory aliasing. So \( M_1 \) and \( M_2 \) only set alias register numbers 1 and 0 respectively with \( P \) bit and, \( C \) bit for memory aliasing if and only if:

- \( P \) bit and \( C \) bit, the memory
- \( Y \) precedes \( X \) in the optimized program
- \( X \) has \( C \) bit and
- \( Y \) has \( P \) bit and
- The alias register allocated to \( X \) is not later than the alias register allocated to \( Y \) in the alias register queue.
register allocation for a reordering optimization, where $M_0, M_1, \ldots, M_f$ represent their original program execution order. Since no memory operation after $M_0$ will set or check $AR_0$, we can rotate the alias register by 1 after $M_0$ as shown in Figure 7 (b). After the rotation, the base alias register pointer $BASE$ will point to $AR_1$, and alias register $AR_0$ is cleaned up and logically becomes a free alias register at the end of the alias register queue. Then $M_2$ needs to use offset 1 to reference alias register $AR_2$ instead of offset 2. Similarly, we can rotate the alias register by another 1 after $M_2$.

Rotation can reduce the register usage, and thus the chance of alias register overflow. For example, if HW supports only 2 alias registers, we can run the code in Figure 7 (b), as after the first rotation, $AR_0$ is reused as alias register $AR_2$ at offset 1. However, we cannot run the code in Figure 7 (a) as there is no alias register at offset 2. In general, the maximum offset + 1 gives the minimum number of HW alias registers required to run the code without alias register overflow.

It seems that in Figure 7 (a), we may directly reuse and set/check the alias register $AR_0$ in $M_4$ and $M_2$ without the rotation. However, the check of $AR_0$ in $M_2$ in this example may lead to a false positive alias detection between reordered memory operations. Rotation makes it hard for the alias register allocation algorithm to manage the ordered alias registers to prevent false positive (see more details in section 5). Therefore, SMARQ is designed to reuse alias registers only through rotation.

<table>
<thead>
<tr>
<th>(a) Without Rotation</th>
<th>(b) With Rotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_0$: $r1 = [r1+8]$</td>
<td>$M_0$: $r1 = [r1+8]$</td>
</tr>
<tr>
<td>$M_1$: $r2 = [r1]$</td>
<td>$M_1$: $r2 = [r1]$</td>
</tr>
<tr>
<td>$M_2$: $r0 = [r0]$</td>
<td>$M_2$: $r0 = [r0]$</td>
</tr>
<tr>
<td>$M_4$: $r1 = [r1]$</td>
<td>$M_4$: $r1 = [r1]$</td>
</tr>
<tr>
<td>$M_5$: $r1 = [r1+4]$</td>
<td>$M_5$: $r1 = [r1+4]$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>set AR0</td>
<td>set AR0</td>
</tr>
<tr>
<td>set AR1</td>
<td>set AR1</td>
</tr>
<tr>
<td>check AR0, AR1</td>
<td>check AR0, AR1</td>
</tr>
<tr>
<td>check AR1, AR2</td>
<td>check AR1, AR2</td>
</tr>
<tr>
<td>check AR2</td>
<td>check AR2</td>
</tr>
</tbody>
</table>

Rotation: 1

- rotate $BASE$ to $AR_1$
- rotate $BASE$ to $AR_2$

(c) Dependences
(d) Constraints

Figure 7: An Alias Register Rotation Example

Due to the rotation of $BASE$, the "offset" relative to $BASE$ does not reflect the actual order of the alias registers in the alias register queue. To avoid confusion, in the rest of the paper, we will use "order" to refer to the alias register number relative to $BASE$ 0, and "offset" to refer to the alias register number relative to the $BASE$ at the memory operation execution. So in Figure 7 (b), the offset of the register allocated to $M_4$ is 1 (denoted offset($M_4$) = 1), but the order of the register allocated to $M_4$ is 2 (denoted order($M_4$) = 2) because the $BASE$ at $M_2$ execution is 1 (denoted base($M_2$) = 1). In general, base($X$) keeps increasing in the optimized program execution with the rotation instructions and order($X$) reflects the actual order of alias registers in the circular queue with the invariance: order($X$) = base($X$) + offset($X$). Note that base($X$) and order($X$) are independent of HW register count and can be from 0 to infinity, but offset($X$) must be smaller than the HW alias register count. Since alias register allocation in SMARQ assigns an order($X$) to each memory operation $X$ that needs an alias register, we may refer to the allocation of alias registers as the allocation of alias register orders.

3.3 Prevent False Positive with Alias Moving

Although order-based alias detection can guarantee to detect all the aliases between reordered memory operations without any false positive, the alias detection between non-reordered memory operations for speculative load/store eliminations (see an example in Figure 5) may introduce false positives (see detailed discussions in Section 5.3). To prevent false positives, SMARQ introduces an alias moving instruction denoted "AMOV offset1, offset2", which moves the memory access range stored in the alias register at offset1 to the alias register at offset2. After the moving, the alias register at offset1 is cleaned up. We also allows offset2 to be same as offset1, which only cleans up memory access range in the alias register at offset1 without moving it to another alias register. The alias moving instructions can prevent all false positives in the alias detection.

4. Constraint Analysis

In this section, we develop the compiler analysis to identifying the necessary alias detection constraints in three general speculative optimizations: memory operation reordering, load elimination and store elimination. These general optimizations subsume other specific memory optimizations, such as speculative register promotion [7], etc. As the first study on this topic, we focus on optimization in superblock regions [15], and we believe the solution can be generalized to arbitrary code regions.

4.1 Check-constraints

In our compiler analysis, we first derive a set of check-constraints, denoted $X \rightarrow$check, $Y$, such that $X$ needs to check $Y$ for aliasing to ensure optimization correctness. We first consider only instruction scheduling without speculative load/store elimination. In this case, check-constraints can be computed in two steps. Before the instruction scheduling, we compute a set of dependences, denoted $X \rightarrow$dep, such that memory operation $Y$ depends on memory operation $X$ with the following conditions:

[DEPENDENCE] $X \rightarrow$dep $Y$ if:
- $X$ precedes $Y$ in the original program execution order and,
- $X$ and $Y$ may (including must) access the same memory location and,
- At least one of $X$ and $Y$ is a store.

Then after the instruction scheduling, we derive the check-constraints $X \rightarrow$check $Y$ from dependences $X \rightarrow$dep $Y$ with the following condition:
[CHECK-CONSTRAINT] \( X \rightarrow \text{check} \ Y \) if:

- \( X \rightarrow \text{dep} \ Y \) and,
- \( Y \) precedes \( X \) after scheduling.

As an example, for the memory reordering in Figure 7, Figure 7 (c) shows all the dependences before scheduling. The solid edges in Figure 7 (d), on the other hand, show all the check-constraints derived from dependences after scheduling. There is no dependence \( M_1 \rightarrow \text{dep} \ M_2 \) or \( M_3 \rightarrow \text{dep} \ M_2 \) since the compiler can easily disambiguate them. Due to that, there is no check-constraint \( M_1 \rightarrow \text{check} \ M_2 \) or \( M_3 \rightarrow \text{check} \ M_2 \), even though they are reordered in the scheduling. For each check-constraint \( X \rightarrow \text{check} \ Y \), we set the \( C \) bit to \( X \) and the \( P \) bit to \( Y \), as shown aside with the memory operations in Figure 7 (d).

To handle speculative load elimination, we only need to extend dependences with the following condition:

[EXTENDED-DEPENDENCE 1] If a load \( Z \) is eliminated speculatively by forwarding data from an earlier memory operation \( X \) (a store or a load) to \( Z \), we add extended dependences \( Y \rightarrow \text{dep} \ X \) for all loads \( Y \) satisfying:

- \( Y \) is between \( X \) and \( Z \) in the original program execution order and,
- \( Y \) and \( X \) may access the same memory.

As an example, in the optimization with speculative load elimination in Figure 5, \( M_1 \) and \( M_3 \) access the same memory location so we speculatively eliminate the load at \( M_3 \) by forwarding data from \( M_1 \) to \( M_3 \), and according to EXTENDED-DEPENDENCE 1, derive the extended dependence \( M_1 \rightarrow \text{dep} \ M_3 \), as shown in Figure 8 (a). With the extended dependence, we derive a check-constraint \( M_1 \rightarrow \text{check} \ M_3 \) after the scheduling according to CHECK-CONSTRAINT, as shown by the bold edge in Figure 8 (b). This is necessary because we need to enforce the alias detection between \( M_1 \) and \( M_3 \) in order for the load elimination to be correct, even though \( M_1 \) and \( M_3 \) are not reordered. Note that there are both dependence \( M_1 \rightarrow \text{dep} \ M_3 \) and extended dependence \( M_1 \rightarrow \text{dep} \ M_3 \) in Figure 8 (a). So depending on the orders between \( M_1 \) and \( M_3 \) after the scheduling, we will have either check-constraint \( M_1 \rightarrow \text{check} \ M_3 \) or \( M_3 \rightarrow \text{check} \ M_1 \) to enforce the alias detection between \( M_1 \) and \( M_3 \).

![Figure 8: Dependences and Constraints for Optimization in Figure 5](image)

Similarly, to handle speculative store elimination, we only need to extend dependences with the following condition:

[EXTENDED-DEPENDENCE 2] If a store \( X \) is eliminated due to the overwriting of the same memory location by a later store \( Z \), we add extended dependence \( Z \rightarrow \text{dep} \ Y \) for all loads \( Y \) satisfying:

- \( Y \) is between \( X \) and \( Z \) in the original program execution order and,
- \( Z \) and \( Y \) may access the same memory.

Figure 9 shows an example for speculative store elimination. \( M_3 \) overwrites the same memory location as accessed by \( M_0 \). So we speculative eliminate \( M_0 \), and according to EXTENDED-DEPENDENCE 2, derive the extended dependence \( M_3 \rightarrow \text{dep} \ M_4 \), as shown in Figure 9 (c). With the extended dependence, we derive the check-constraint \( M_3 \rightarrow \text{check} \ M_4 \) as shown by the bold edge in Figure 9 (d). This is necessary because we need to enforce the alias detection between \( M_1 \) and \( M_5 \) in order for the store elimination to be correct, even though \( M_1 \) and \( M_5 \) are not reordered after optimization. It is interesting that in EXTENDED-DEPENDENCE 2, we add extended dependence \( Z \rightarrow \text{dep} \ Y \) only for load \( Y \) between \( X \) and \( Z \), but not for store \( Y \) between \( X \) and \( Z \). Due to that, we do not enforce the alias detection between \( M_2 \) and \( M_5 \), or between \( M_2 \) and \( M_4 \), as the aliases between them do not affect the correctness of the store elimination (i.e. there is no need for check-constraint \( M_3 \rightarrow \text{check} \ M_2 \) or \( M_3 \rightarrow \text{check} \ M_4 \)).

![Figure 9: A Speculative Store Elimination Example](image)

4.2 Anti-Constraints

In the optimization example shown in Figure 5, we get the check-constraints \( M_0 \rightarrow \text{check} \ M_2 \) and \( M_4 \rightarrow \text{check} \ M_3 \), as shown in Figure 8 (b). Due to ORDERED-ALIAS-DETECTION-RULE (described in Section 3), the alias detection between \( M_0 \) and \( M_2 \) requires \( \text{order}(M_0) \leq \text{order}(M_2) \), and the alias detection between \( M_4 \) and \( M_2 \) requires \( \text{order}(M_4) \leq \text{order}(M_2) \). If we allocate \( M_2 \) before \( M_0 \), transitively, we will have \( \text{order}(M_2) \leq \text{order}(M_0) \) and \( \text{order}(M_4) \leq \text{order}(M_0) \), which means that \( M_2 \) will additionally check \( M_0 \) for aliasing. The resulting alias detection is shown in Figure 10 (a) (note the additional check of \( ARI \) by \( M_0 \)). Similarly, if we allocate \( M_0 \) before \( M_2 \), transitively, we will have \( \text{order}(M_2) \leq \text{order}(M_0) \) and \( \text{order}(M_4) \leq \text{order}(M_0) \), which means that \( M_2 \) will additionally check \( M_4 \) for aliasing. The resulting alias detection is shown in Figure 10 (b) (note the additional check of \( ARI \) by \( M_4 \)). In either case, we have to perform an additional alias detection not specified in the check-constraints in Figure 8 (b).

Although the additional detection between \( M_0 \) and \( M_2 \) in Figure 10 (a) is benign (i.e. will never generate alias exception), the additional alias detection between \( M_2 \) and \( M_4 \) in Figure 10 (b) will generate an alias exception, even though the alias does not affect the optimization correctness. This false positive will result in expensive rollback of execution.

To prevent false positive, our compiler analysis also derives anti-constraint, denoted \( X \rightarrow \text{anti} \ Y \), such that \( X \) should not be checked by \( Y \). So for the optimization in Figure 5, we need to derive the anti-constraint \( M_2 \rightarrow \text{anti} \ M_1 \) to prevent \( M_1 \) from being checked by \( M_2 \). Figure 11 shows how to prevent false positive
with anti-constraints. There are multiple alias register allocations (i.e. allocation 1 and allocation 2) that can perform all the alias detections enforced by check-constraints. However, allocation 2 also performs certain alias detections prohibited by anti-constraints which may lead to false positive. Our alias register allocation algorithm aims to find allocation 1, which performs all the alias detections enforced by the check-constraints without any alias detections prohibited by the anti-constraints.

(a) Alias Register Allocation 1

<table>
<thead>
<tr>
<th>M1: … = ld [r1]</th>
<th>0</th>
<th>P</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1: r2 = ld [r0+4]</td>
<td>1</td>
<td>P</td>
<td>-</td>
</tr>
<tr>
<td>M1: st [r0] = ...</td>
<td>0</td>
<td>C</td>
<td>-</td>
</tr>
<tr>
<td>M1: st [r1] = ...</td>
<td>1</td>
<td>C</td>
<td>-</td>
</tr>
<tr>
<td>M1: r4 = r2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

(b) Alias Register Allocation 2

<table>
<thead>
<tr>
<th>M2: … = ld [r1]</th>
<th>1</th>
<th>P</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>M2: r2 = ld [r0+4]</td>
<td>0</td>
<td>P</td>
<td>-</td>
</tr>
<tr>
<td>M2: st [r0] = ...</td>
<td>1</td>
<td>C</td>
<td>-</td>
</tr>
<tr>
<td>M2: st [r1] = ...</td>
<td>0</td>
<td>C</td>
<td>-</td>
</tr>
<tr>
<td>M2: r4 = r2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 10: False Positive for the Example in Figure 5

Alias detections enforced by check-constraints

Alias detections prohibited by anti-constraints

Alias detections performed with alias register allocation 1

Alias detections performed with alias register allocation 2

Figure 11: Preventing False Positive

An anti-constraint $X \Rightarrow anti \ Y$ is needed only if $X$ and $Y$ may alias to each other. So we compute the anti-constraints $X \Rightarrow anti \ Y$ from the subset of dependences $X \Rightarrow dep \ Y$ such that $X$ precedes $Y$ after scheduling. Moreover, if $Y \Rightarrow check \ X$, then $Y$ does need to check $X$ for aliasing to ensure optimization correctness. We cannot prohibit $Y$ from checking $X$ in that case (Remember that $X \Rightarrow anti \ Y$ means $Y$ cannot check $X$, which is opposite to what $Y \Rightarrow check \ X$ means). So an anti-constraint $X \Rightarrow anti \ Y$ can be enforced only when there is no $Y \Rightarrow check \ X$. At last, since only a memory operation $X$ with $P$ bit may be checked by a memory operation $Y$ with $C$ bit, an anti-constraint $X \Rightarrow anti \ Y$ is needed only if $X$ has $P$ bit and $Y$ has $C$ bit. So in summary, we compute the anti-constraints $X \Rightarrow anti \ Y$ with the following condition:

[ANTI-CONSTRAINT] $X \Rightarrow anti \ Y$ if:

- $X \Rightarrow dep \ Y$ and,
- $X$ precedes $Y$ after scheduling and,
- there is no $Y \Rightarrow check \ X$ and,
- $X$ has $P$ bit and,
- $Y$ has $C$ bit.

As an example in Figure 8 (b), we will derive the anti-constraint $M_2 \Rightarrow anti \ M_1$, with ANTI-CONSTRAINT. There is no anti-constraint $M_1 \Rightarrow anti \ M_1$ due to $M_1 \Rightarrow check \ M_1$. There is also no anti-constraint $M_0 \Rightarrow anti \ M_1$ because $M_0$ does not have $P$ bit.

5. Alias Register Allocation Algorithm

In this section, we show how to allocate alias registers to memory operations to satisfy all the check-constraints and anti-constraints. These constraints together form a constraint graph. First, we show how to perform a fast allocation if there is no cycle in the constraint graph. Then based on that, we show how to handle cycles. We then discuss how to avoid register overflow. Finally, we put together all of these parts into a complete algorithm.

5.1 Fast Alias Register Allocation in Constraint Order

According to ORDERED-ALIAS-DETECTION-RULE, we have:

[REGISTER-ALLOCATION-RULE]

- Given a check-constraint $X \Rightarrow check \ Y$, the alias register allocated to $X$ must be no later than the alias register allocated to $Y$, i.e. $order(X) \leq order(Y)$.
- Given an anti-constraint $X \Rightarrow anti \ Y$, the alias register allocated to $X$ must be earlier than the alias register allocated to $Y$, i.e. $order(X) < order(Y)$.

So the constraints naturally provide the orders for alias registers. If there is no cycle in the constraint graph, we can simply allocate alias registers to memory operations with a topological traversal of the constraint graph:

[FAST ALGORITHM] We use next_order to keep track of the next available alias register (i.e. its order) to be allocated, which is initialized to 0. We traverse memory operations in a topological order of the constraint graph to allocate alias registers. For a memory operation $X$, if $P(X)$ is set, we allocate a new alias register order to $X$ with $order(X) = next_order$ and increase next_order by 1. If only $C(X)$ is set, we just set $order(X) = next_order$ without increasing next_order.

Without alias register rotation, we can directly use $order(X)$ computed in FAST ALGORITHM as the alias register offset to satisfy all the constraints. As an example, for the optimization shown in Figure 7, we can allocate the alias registers in the topological order $M_0, M_1, M_2, M_3, M_4$ of the constraint order in Figure 7 (d) and get the alias register allocation as shown in Figure 7 (a).

Alias register allocation happens when an alias register offset is equal to or larger than the physical alias register count. Given $order(X)$, due to the invariance $order(X) = base(X) + offset(X)$, we can minimize offset(X) by maximizing base(X) through alias register rotation. Assume $base(X) = i$. Then all alias registers with orders in the range $[i, i+1]$ are released at the execution of $X$ and can no longer be set/checked by $X$ or any memory operations executed after $X$. So mathematically, we can compute maximum base(X) with the formula:

[MAX-BASE] $base(X) = MIN \{order(Y) | \forall Y \text{ that is } X \text{ succeeds } X \text{ after scheduling}\}$.

Then we need to insert rotating instruction “rotate base(X2) – base(X1)” between two consecutive memory operations $X_1, X_2$ if $base(X_1) < base(X_2)$. For the alias register allocation shown in Figure 7 (a), by maximizing base(X), we will get $base(M_0) = base(M_2) = base(M_4) = 0, base(M_3) = base(M_1) = 1, base(M_0) = 2$, and the register allocation shown in Figure 7 (b).
5.2 Handle Cycles in Constraint Graph

For an optimization that only speculatively reorders memory operations in superblocks, the constraint order cannot contain cycles because all the dependences and hence all the constraints (which are subset of dependences) follow the original program execution order. This is the reason that the alias register allocation in program order can always detect all the aliases without any false positive. However, with the speculative load/store elimination, we introduce extended dependences in the backward execution order of the original program. Hence we may encounter cycles in the constraint graph as shown in Figure 9 (d).

According to REGISTER-ALLOCATION-RULE, no alias register order can satisfy all the constraints in Figure 9 (d). So we introduce extended dependences in the backward execution order. However, with the speculative load/store elimination, we can always detect all the aliases without any false positive. This is the reason that the alias register allocation in program order cannot contain cycles. For an optimization that only speculatively reorders memory operations in superblocks, the constraints in Figure 9 (d) are subset of dependences) follow the original program execution order (line 2). Since there is no constraint added yet, the invariance holds at this point. As we add each constraint, we check if the invariance is met. If not (i.e. \( T(X) \geq T(Y) \)), we take necessary actions so that the invariance is met throughout all the constraints.

5.3 Prevent Alias Register Overflow

Since there is no hardware support to spill alias register, we embed our alias register allocation within a list scheduling framework so that we can allocate alias register during the instruction scheduling. In this way, we can dynamically adjust the schedule to prevent alias register overflow. Our list scheduler schedules the instructions in two different modes. When there are enough alias registers and thus no danger of overflow, we schedule instructions in the speculation mode, which speculatively reorders memory operations with alias registers. When the alias register may overflow, we switch to the non-speculation mode, which only rotates alias registers to release unused alias registers without allocating new alias registers. After releasing enough alias registers, we will switch back to the speculation mode.

5.4 Overall Alias Register Allocation Algorithm

The overall algorithm for the alias register allocation is shown as pseudo code in Figure 13, with details discussed in following sections. Note that alias register allocation is integrated with list scheduling, and we omit any necessary steps required by scheduler here.

5.4.1 Algorithm Overview

Initialization (lines 1-3): The algorithm begins with initializing required information. All the dependences are calculated for memory operations. For detecting cycles in the constraint graph, a partial order \( T(X) \) for each memory operation \( X \) is initialized to the original program execution order (line 2). The details on cycle detection with \( T(X) \) are explained later in this section.

Incrementally Building Constraints (lines 8-17): After scheduling a memory operation \( Y \) in the list scheduler, we incrementally build the constraints by adding all the constraints \( X \rightarrow check \ Y \) and \( X \rightarrow anti \ Y \) (line 11 and 14).

Once a memory operation \( Y \) is scheduled, all the dependences \( X \rightarrow dep \ Y \) are examined for adding corresponding constraints (line 8). In our list scheduler, the schedule is constructed by filling time slots in an increasing manner. That is, instructions scheduled later than \( X \) are always placed in the same or later time slots. Since check-constraints are always in a backward direction in the schedule and anti-constraints are always in a forward direction in the schedule, we only need to consider not-yet-scheduled instructions for check-constraints, while only considering already-scheduled instructions for anti-constraints. Lines 9 and 13 show the conditions for adding check-constraints and anti-constraints, respectively.

Cycle Detection (lines 33-54): An incremental cycle detection algorithm [12] is employed for cycle detection in the constraint graph. The algorithm maintains the following partial order \( T \) for all the instructions to ensure that there is no cycle in the constraint graph.

\[ Invariance: \text{if there exists a constraint } X \rightarrow check \ Y \text{ or } X \rightarrow anti \ Y, \text{then } T(X) < T(Y) \]

We first initialize \( T(X) \) for all the instructions to the original program execution order (line 2). Since there is no constraint added yet, the invariance holds at this point. As we add each constraint, we check if the invariance is met. If not (i.e. \( T(X) \geq T(Y) \)), we take necessary actions so that the invariance is met throughout all the constraints.

\[ M_1: ... = ld [r1] \]
\[ M_2: st [r4] = ... \]
\[ M_3: st [r2] = ... \]
\[ M'_4: AMOV 2, 0 \]
\[ M_4: st [r4] = ... \]
\[ M_4: st [r4] = ... \]
\[ M_5: ... = ld [r0+4] \]

Figure 12: AMOV for the Example in Figure 9

According to REGISTER-ALLOCATION-RULE, no alias register order can satisfy all the constraints in Figure 9 (d). So we need to insert an alias moving instruction AMOV to break the cycles. Figure 12 (b) shows the same constraints as Figure 9 (d). Suppose an AMOV instruction \( M'_4 \) is inserted between \( M_2 \) and \( M_4 \) to move the memory access range from the alias register allocated to \( M_4 \) to the new alias register allocated to \( M'_4 \) (how to choose an alias register to move is explained later in Section 5.4). Then constraints \( M_1 \rightarrow check \ M_4 \) and \( M_4 \rightarrow anti \ M_1 \) need to be changed to \( M_1 \rightarrow check \ M'_4 \) and \( M'_4 \rightarrow anti \ M_1 \) respectively as shown in Figure 12 (c), as at the execution of \( M_1 \) and \( M_4 \), the alias access range of \( M_4 \) will stay in the alias register allocated to \( M'_4 \) instead of the alias register allocated to \( M_4 \). So the alias moving instruction \( M'_4 \) breaks the cycles in Figure 12 (b) to get the constraints without cycle as shown in Figure 12 (c). Allocating the alias registers in constraint order as shown in Figure 12 (a). Instruction \( M'_4 \) moves the memory access range from \( AR2 \) (i.e. the alias register allocated to \( M_4 \)) to \( AR0 \) (i.e. the alias register allocated to \( M'_4 \)). Then \( M_2 \) will not check \( M_4 \) for aliasing. However, \( M_2 \) still can check the memory access range of \( M_4 \) in \( AR0 \) for aliasing.

In many cases, the AMOV instruction does not really need an additional alias register to save the memory access range, and the AMOV often needs merely to clean up the memory access range in the source alias register to avoid false positive. We will quantify this effect in the experiment section.

![Figure 12: AMOV for the Example in Figure 9](image-url)
compute all dependences (including extended dependences);
initialize all \( T(X) \) to the original program execution order;
next_order = 0;

while (there is not-scheduled instruction) {
    \( Y = \text{next_instruction}(); \) // next instruction for scheduling
    if (\( Y \) is not memory instruction) continue;
    for (all dependences \( X \rightarrow \text{dep} \ Y \)) {
        if (\( X \) is not scheduled) {
            set \( C(X), P(Y) \);
            add \( X \rightarrow \text{check} \ Y \);
            if (\( T(X) > T(Y) \) \( \text{or} \) \( T(X) = T(Y) - 1 \)) { \}
                else if (offset(\( Y \)) not set, \( P(X), C(Y) \), no \( Y \rightarrow \text{check} X \)) {
                    add \( X \rightarrow \text{anti} \ Y \);
                    if (\( T(X) > T(Y) \)) detect_cycle(X, Y);
                } \}
        }
    }
    if (\( P(Y) \) or \( C(Y) \)) allocate_reg(Y);
}

detect_cycle(X, Y) { // for \( X \rightarrow \text{anti} \ Y \)
    \( \delta = T(X) - (T(Y) - 1) \);
    set \( H \) = instructions reachable from \( X \) in constraint order;
    if (\( X \) not in set \( H \)) { // no cycle detected
        for (all instruction \( Z \) in \( H \)) {
            \( T(Z) = T(Z) + \delta \);
            insert AMOV(X) before \( Y \);
            if (\( Z \rightarrow \text{check} \ X \) not scheduled) {
                replace \( X \rightarrow \text{check} \ X \) with \( Z \rightarrow \text{check} \ X \);
                set \( P(X) \);
                \( T(X) = T(Y) - 1 \);
                add \( X \rightarrow \text{delay_queue} \);
                base(X) = next_order;
            } \}
        }
        remove \( X \rightarrow \text{anti} \ Y \);
    } else {
        for (\( X \rightarrow \text{check} \ Z \) or \( X \rightarrow \text{anti} \ Z \) \( \text{not} \) scheduled) {
            delete \( X \rightarrow \text{check} \ Z \) or \( X \rightarrow \text{anti} \ Z \);
            if (\( X \) not is \( \ast \)check \( Z \) or \( \ast \)anti \( Z \))
                add \( Z \) into \( \text{ready_queue} \);
            \}
        }
    }
}

allocate_reg(Y) {
    base(Y) = next_order;
    if there is no \( \ast \)check \( Y \) or \( \ast \)anti \( Y \)
        add \( Y \) into \( \text{ready_queue} \);
    else {
        while (\( \text{ready_queue} \) not empty) {
            \( X = \text{ready_queue}.\text{dequeue}() \);
            \( \text{offset}(X) = \text{next_order} - \text{base}(X) \);
            if (\( P(X) \) \( \text{next_order} \)++;
                for (\( X \rightarrow \text{check} \ Z \) or \( X \rightarrow \text{anti} \ Z \))
                    delete \( X \rightarrow \text{check} \ Z \) or \( X \rightarrow \text{anti} \ Z \);
                if there is no \( \ast \)check \( Z \) or \( \ast \)anti \( Z \)
                    add \( Z \) into \( \text{ready_queue} \);
            } \}
        if (next_order > base(Y))
            insert rotate instruction after \( Y \) by next_order – base(Y);
        dequeue all \( X \) at the head of \( \text{delay_queue} \) if offset(\( X \)) is set
    }
}

Figure 13: Alias Register Allocation Algorithm

For a check-constraint \( X \rightarrow \text{check} \ Y \), we simply decrease \( T(X) \)
to \( T(Y) - 1 \) (line 12) to ensure the invariance. Since \( X \) is not scheduled yet, there is no constraint \( \ast \)check \( X \) or \( \ast \)anti \( X \) yet. Therefore, this action will neither break any existing invariance conditions nor create any cycle in the constraints.

However, adding an anti-constraint \( X \rightarrow \text{anti} \ Y \) requires more complicated step (detect_cycle() in line 33), since it can create a cycle. We first build a set \( H \) of instructions that are reachable from \( Y \) in the constraint order (including \( Y \) itself). If \( X \) is not in \( H \), there is no cycle created and we simply increase the order numbers of \( Y \) and its connected component (set \( H \)) as in line 37-38. When a cycle is detected (\( X \) is in set \( H \)), we break the cycle by inserting an AMOV instruction (\( X' \)) just before \( Y \) (line 40). After the execution of \( X' \), the memory access range for \( X \) is transferred to the alias register allocated to \( X' \). So, all the instructions not yet scheduled (\( Z \)) should check \( X' \) instead of \( X \) (line 42). We also need to remove the original anti-constraint \( X \rightarrow \text{anti} \ Y \) (line 46) and add a new one \( X' \rightarrow \text{anti} \ Y \) if \( P(X') \) is set (line 48). Finally, \( X' \) is put into the delay_queue since it is not ready for allocation yet.

Allocation (lines 56-75): First, we remember the base pointer at \( Y \) execution in base(\( Y \)). This is used to calculate the offset of the allocated register (offset(\( Y \))). We maintain two FIFO queues for allocation: ready_queue and delay_queue. If \( Y \) has no constraints \( \ast \)check \( Y \) or \( \ast \)anti \( Y \), \( Y \) is ready for allocation and put into ready_queue. Otherwise, allocation is delayed by putting it into delay_queue. For instructions \( X \) in ready_queue, a simple allocation is performed as in line 64-65. When the alias register for an instruction \( X \) is allocated, we delete all the constraints \( X \rightarrow \text{check} \ Z \) or \( X \rightarrow \text{anti} \ Z \). This could make other instructions (\( Z \)) in delay_queue ready for allocation and we can continue with them. Due to the delayed alias register allocation, an alias register is allocated only when the last instruction that uses it is scheduled (i.e. the alias register can be released after the scheduled instruction).

So we rotate the base pointer after the scheduled instruction with the amount of newly allocated registers (line 72-73) and dequeue allocated memory operations from the head of delay_queue (line 74).

Preventing Overflow (lines 21-31): To avoid overflow, we need to ensure that offset(\( X \)) should not equal to or exceed the number of physical alias registers. We bound the maximum possible offset with the following three numbers.

First, we estimate the minimum possible base pointer (line 22). Then the maximum register order (max_order) is estimated. In addition to the allocated registers (next_order), each instruction \( Z \) with \( P(Z) \) set in delay_queue requires a new alias register so we estimate that they each will use a separate alias register (line 23). Thirdly, we need to consider the future register usage due to not-yet-scheduled instructions. Even if we follow the data dependencies without speculation, we might still need more alias registers for extended dependences introduced by load/store eliminations (line 24). The maximum possible offset is conservatively calculated (line 25) and it is compared against the number of physical alias registers to determine the memory reordering policy in the list scheduler.

5.4.2 Allocation Example

As an example of alias register allocation, let’s take a look at the optimization performed in Figure 7. After scheduling \( M_8 \), we insert a check-constraint \( M_8 \rightarrow \text{check} M_9 \) and delay the allocation for \( M_9 \). Similarly, after scheduling \( M_9 \), we insert check-constraints \( M_9 \rightarrow \text{check} M_{10}, M_{11} \rightarrow \text{check} M_7 \) and delay the allocation for \( M_7 \) as well. Then when scheduling \( M_{10} \), we allocate next_order 0 (i.e.
After the register allocation for \( M_0 \), we delete the check-constraints \( M_0 \rightarrow \text{check} M_3 \) and \( M_0 \rightarrow \text{check} M_5 \). Since there is no constraint \( \rightarrow \text{check} M_3 \) or \( \rightarrow \text{anti} M_5 \), \( M_5 \) becomes ready for allocation. Then we allocate \( \text{next_order} = 0 \) (i.e. \( \text{offset}(M_0) = 0 \)) to \( M_0 \), and increase \( \text{next_order} \) by 1. We then insert rotation by 1 after \( M_0 \), to rotate the allocated alias register at order 1. After scheduling \( M_0 \), the alias register allocation for \( M_0 \) is delayed due to the check-constraint \( \rightarrow \text{check} M_4 \) and \( \rightarrow \text{check} M_6 \). Since now \( \text{next_order} = 1 \), \( \text{base}(M_0) \) will get \( 1 \), which indicates that \( \text{BASE} \) is 1 at the execution of \( M_0 \). When scheduling \( M_0 \), we will allocate \( \text{next_order} = 1 \) to \( M_0 \), with P bit and \( \text{next_order} = 1 \) to 2. Since \( \text{base}(M_0) = 0 \), we will get \( \text{offset}(M_0) = 1 \). After \( M_0 \), we insert rotation by 1 to rotate the allocated alias register at order 1. At last after scheduling \( M_0 \), we allocate \( \text{next_order} \) 2 to \( M_1 \) and \( M_4 \). Since \( \text{base}(M_1) = 2 \) and \( \text{base}(M_4) = 1 \), we get \( \text{offset}(M_1) = 0 \) and \( \text{offset}(M_4) = 1 \). The final register allocation will be as shown in Figure 7 (b).

### 5.4.3 Cycle Detection Example

We take the constraints in Figure 12 (b) (based on the optimization shown in Figure 9) and illustrate how the cycle detection algorithm [12] works in our allocation scheme. We initialize \( T(M_j) = 1 \), \( T(M_2) = 2 \), \( T(M_3) = 3 \), \( T(M_4) = 4 \) and \( T(M_5) = 5 \). After scheduling of \( M_0 \), we add check-constraint \( M_0 \rightarrow \text{check} M_4 \) and \( M_0 \rightarrow \text{check} M_6 \) and delay the alias register allocation for \( M_0 \). Since \( T(M_0) \geq T(M_j) \), we update \( T(M_j) = T(M_0) - 1 = 0 \). After the scheduling of \( M_0 \), we add check-constraint \( M_2 \rightarrow \text{check} M_4 \) and \( M_2 \rightarrow \text{check} M_6 \) and delay the alias register allocation for \( M_0 \). After the scheduling of \( M_0 \), we add anti-constraint \( M_0 \rightarrow \text{anti M}_1 \) and delay the alias register allocation for \( M_2 \). When scheduling \( M_0 \), we need to add anti-constraint \( M_0 \rightarrow \text{anti M}_3 \). However, \( T(M_0) \geq T(M_3) \). Then we find the set \( H = \{ M_6, M_1, M_5, M_7 \} \) of instructions that are reachable from \( M_7 \). Since \( M_5 \) is in set \( H \), a cycle will be created with the adding of anti-constraint \( M_5 \rightarrow \text{anti M}_3 \). So we insert an alias moving instruction \( M_i \) before \( M_7 \) to move the memory access range of \( M_i \). We then replace the check-constraint \( M_3 \rightarrow \text{check M}_4 \), with \( M_3 \rightarrow \text{check M}_4' \), add anti-constraint \( M_3 \rightarrow \text{anti M}_4 \) and delay the alias register allocation for \( M_3 \) and \( M_5 \). We also set \( T(M_i) = -1 \) and update \( T(M_i) \) to -2. The resulting constraints are as shown in Figure 12 (c). At last we schedule \( M_3 \) and allocate alias registers to all the instructions in \( \text{delay_queue} \). The resulting code and alias register allocation is shown in Figure 12 (a).

### 6. Experiments

We evaluate SMARQ in a dynamic optimization framework as shown in Figure 1, which translates and optimizes x86 binary codes into code running on an internal VLIW CPU modeled by a cycle-accurate simulator. The VLIW simulator supports atomic region execution [10, 11] with 64 alias registers. Table 2 lists the important architecture parameters for the VLIW simulator.

Our dynamic optimizer forms superblock regions for optimization similar to [6]. X86 binary code is first executed through interpretation. The system profiles the execution for hot basic blocks. When a hot block is identified (i.e. the execution count of a basic block reaches a threshold for hotness), the dynamic optimizer forms a region along the hot execution paths starting from the basic block until it reaches a cold block (i.e. execution count lower than a threshold for coldness). After region formation, the x86 binary code is translated into an Internal Representation (IR) for optimizations. Table 2 lists the important optimizations. After all the optimizations, the optimizer performs register allocation and then the instruction scheduling algorithm as shown in Figure 13.

### 6.1 Performance Improvement

Figure 15 shows the speedups of SMARQ and two other alias detection schemes. The first one (SMARQ16) models an alias detection approach that uses only 16 ordered alias registers (we may refer to it as Efficeon-like since Efficeon scheme can only supports...
The left bar shows the speedup with SMARQ. Overall, optimizations with SMARQ improve the performance by 39% as compared to that without hardware alias detection support. The middle bar in Figure 15 shows the speedup of SMARQ16 with only 16 alias registers. On average, we get 29% speedup, a 10% smaller speedup than SMARQ. For ammp, the impact is as high as 30% due to the large number of memory operations in the superblocks, as shown in Figure 14. So the scalable alias register scheme in SMARQ is important for speculative optimization on programs with large superblocks. We believe SMARQ is even more promising for larger region and loop level optimizations. The right bar in Figure 15 shows the speedup with the Itanium-like model. With the non-ordered alias detection, we can only get 26% speedup, a 13% smaller speedup than SMARQ. For ammp, the impact is as high as 47%.

![Figure 15: Speedup with Different Alias Detection](image)

Note that, Itanium alias detection may not even achieve the performance in the right bar of Figure 15 because Itanium cannot detect alias between reordered stores. Figure 16 shows the performance impact of disabling store reordering. Without the store reorder in the optimization, on average, we observe 2.6% performance impact. For mesa, the impact is as high as 13%. Note that the ammp benchmark shows slight performance loss with store reorder. This can happen when the reordered stores alias to each other at runtime and cause execution rollback.

![Figure 16: Impact of Store Reordering](image)

### 6.2 Working Set Reduction

In SMARQ, all the live alias registers stay in a slide window referenced through alias register offsets. The window is from the current BASE to the maximum offset. So the maximum offset + 1 determines the size of the alias register working set. To prevent alias register overflow, the size of the alias register working set cannot exceed the physical alias register count.

The straightforward alias register allocation in program order cannot handle speculative load/store elimination, so we cannot directly compare performance with it. Since supporting aggressive reordering without alias register overflow is the key to performance, we collect statistics for the working set in the alias register allocation, as shown in Figure 17. All the data are normalize to the number of memory operations averaged over all the superblocks, which represents the size of the alias register working set by allocating each memory operation an alias register in program order. The first bar shows the number of memory operations that have P bits, which represents the size of the working set by allocating alias registers in program order to the memory operations that set alias registers. The second bar shows the size of the alias register working set in SMARQ. We can see that SMARQ can reduce the alias register working set by 74% as compared to the straightforward alias register allocation for each memory operation in program order. Even compare to the alias register allocation in program order for only the memory operations that set alias registers (i.e. the first bar), SMARQ can reduce the working set by 25% through rotation.

Given a check-constraint \( X \nrightarrow \text{check } Y \), the alias register set by \( Y \) needs to be kept alive between the execution of instruction \( Y \) and instruction \( X \). So like traditional register allocation, the maximum number of live-ranges that cross any program point provides a lower-bound on the size of the alias register working set in all possible alias register allocations. The last bar of Figure 17 shows that lower-bound. We can see that the size of the alias register working set in our fast alias register allocation with constraint order is close to the lower bound.

![Figure 17: Alias Register Working Set](image)

### 6.3 Optimization Overhead

To measure the optimization overhead, we put special markers (symbols) around our algorithm implementation. During simulation, the simulator will detect the markers and record the execution time to measure the optimization overheads. Figure 18 shows the translation overheads. The left bar shows the percentage of execution time spent in the overall optimization and the right bar shows the percentage of execution time spent in the scheduling. Overall, only 0.05% of execution time is spent in the optimization. Within it, around half of time is spent in the scheduling, which includes our alias register allocation.
6.4 Alias Register Allocation Statistics

To further show the efficiency of SMARQ, Figure 19 shows the number of constraints normalized to the number of memory operations. On average, for each scheduled memory operation, we insert 1.3 check-constraints and 0.1 anti-constraints. So the constraint graph is very sparse and the number of edges in the constraint graph is close to the number of memory operations.

7. Related Works

Dynamic binary optimization morphs existing binary programs to improve performance [6,17], save power [16], enhance security, reliability, and parallelism [20]. Architectural supports, such as atomic region [10,11] and alias registers [3,4] are crucially important for aggressive instruction scheduling and speculative optimizations, due to many constraints at binary level [18].

Compiler managed data speculation has been considered an important technique for instruction scheduling [1,3,5,7,8]. DAISY [17] used a load-verify instruction to reload the value of a previously speculated load in program order. Itanium [2] uses ALAT (Advanced Load Address Table) to enable scheduling of load above stores. The load moved above a store is encoded as an advanced load, and in the original load location there is a check-load to verify that no stores have invalidated the advanced load. This scheme requires each store to check all the advanced loads executed before the store and may result in false positives. Also it cannot be used for general instruction scheduling (e.g. move store above aliased store) or optimizations (e.g. store load forwarding, etc).

The Transmeta Effecion [6] uses a mask to selectively check multiple registers without false positive. It can also support scheduling of stores. However, the scheme is not scalable and limits the number of alias registers to the width of the bit-mask.

There is a proposal to use order-based alias detection for instruction scheduling [4]. However, it targets primarily at reordering memory operations. It has serious limitation when apply to other speculative optimizations, such as load/store eliminations. We identified the potential false positive issue and extended hardware support with the novel register move technique to handle potential false positive issues. We developed a novel fast algorithm to allocate the alias registers in constraint order and demonstrated its effectiveness via extensive experiment.

The study in [9] shows that compiler managed data speculation is less useful in a source compiler due to recent advance in static alias analysis techniques, and is significantly more useful in a dynamic binary optimization environment where sophisticated alias analysis is impractical. A simple and fast binary level alias analysis technique was proposed in [13]. However, it can only discover a small subset of aliases, mostly those between direct memory accesses and those indexed by stack-frame registers [14]. The technique proposed in [14] can discover reasonable amount of alias between memory operations indexed by non-stack-frame registers. Although it is useful for static analysis of executable files, it is impractical for dynamic optimizations, as it may take hours to analyze one binary. Speculative alias analysis [1] uses memory region analyses and profile information to increase alias analysis coverage, and use checking and recovery code for correctness. The increased alias analysis coverage can help our speculative optimizations to filter out the likely aliased cases and reduce recovery overhead.

8. Conclusion and Future Work

In this paper, we first studied in details the order-based alias detection mechanism and identified two serious limitations. First,
it can potentially generate false positives when it is used to support general speculative optimizations. Second, it imposes serious challenge to the allocation algorithm and a straightforward allocation in program order unnecessarily uses too many alias registers. We extended the mechanism with a simple alias-move instruction to avoid potential false positive, and develop a novel fast allocation algorithm to use the alias registers not only in instruction scheduling, but also in speculative optimizations. The new algorithm can avoid all the potential false positives so all the aliases detected will affect the optimization correctness. Our experiments show that dynamic binary optimization on a VLIW processor with 64 alias register can improve the overall performance by 39% as compared to the optimization without alias register. Our software-managed order-based alias detection can reduce the alias register working set by 74% over the straightforward allocation in original program execution order.

This work can be extended in several directions. First, we have informally argued that register “rotation” is the preferred method for reusing order-based alias registers. It would be interesting to prove that the traditional lifetime analysis may not be able to achieve better register usage efficiency than our SMARQ algorithm for order-based alias registers. Second, our alias register allocation algorithm is integrated with a list scheduler. We may try to integrate the allocation algorithm with other instruction scheduling techniques, such as software pipelining, etc.

References