DeAliaser: Alias Speculation Using Atomic Region Support

Wonsun Ahn Yuelu Duan Josep Torrellas

University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu

Abstract

Alias analysis is a critical component in many compiler optimizations. A promising approach to reduce the complexity of alias analysis is to use speculation. The approach consists of performing optimizations assuming the alias relationships that are true most of the time, and repairing the code when such relationships are found not to hold through runtime checks.

This paper proposes a general alias speculation scheme that leverages upcoming hardware support for transactions with the help of some ISA extensions. The ability of transactions to checkpoint and roll back frees the compiler to pursue aggressive optimizations without having to worry about recovery code. Also, exposing the memory conflict detection hardware in transactions to software allows runtime checking of aliases with little or no overhead. We test the potential of the novel alias speculation approach with Loop Invariant Code Motion (LICM), Global Value Numbering (GVN), and Partial Redundancy Elimination (PRE) optimization passes. On average, they are shown to reduce program execution time by 9% in SPEC FP2006 applications and 3% in SPEC INT2006 applications over the alias analysis of a state-of-the-art compiler.

Categories and Subject Descriptors C.1.2 [*Processor Architectures*]: Multiple Data Stream Architectures — MIMD processors; D.3.4 [*Programming Languages*]: Processors — Compilers, Optimization

General Terms Algorithms, Design, Performance.

Keywords Alias Analysis, Atomic Region, Transactional Memory, Compiler Optimization.

1. Introduction

Recently, there has been a flurry of activity in the industry to integrate support for transactional memory into processors, in the hopes of making parallel programming more tractable. Several mechanisms or machines now provide architectural support for transactions, such as Intel's Transactional Synchronization Extensions [2], IBM's Bluegene/Q [1], IBM's POWER architecture [3], AMD's Advanced Synchronization Facility [10], and Azul's Vega [11].

Interestingly, transactions can be leveraged for a completely different purpose — to aid the compiler generate better code. In particular, they can enable more aggressive code motion transformations. Code motion forms the basis of most redundancy elimination optimizations, such as Loop Invariant Code Motion (LICM), Global Value Numbering (GVN), Dead Store Elimination (DSE), and Partial Redundancy Elimination (PRE). In these optimizations, the compiler attempts to move a computation to a location where it is less frequently executed, or to a location where it can be eliminated by proving redundancy. However, code motion is prevented when there is an intervening memory access in its span that potentially aliases with the computation.

Unfortunately, proving that such intervening memory accesses do not alias is a notoriously difficult problem, especially in the presence of pointers. The general problem of proving the aliasing properties of a program, or alias analysis, has been of much interest. Many researchers have worked on it over the years [6, 16, 17, 24, 37, 42], making important progress. However, the difficulty has been that there is a trade-off between precision and efficiency, and the most accurate algorithms have too high space and time complexity to gain general use [18]. Indeed, alias analysis is hard even from a theoretical point of view [9, 19, 22, 34, 36].

Luckily, the use of transactions, or atomic regions, can be a game-changer in the field of alias analysis. There are two reasons for this. First, transactions provide the ability to checkpoint and buffer modified memory state — since they have to guarantee commit atomicity. Second, they provide the ability to perform alias checks of memory state accessed by the transaction against other memory accesses — since they have to guarantee isolation. The former allows the compiler to perform code motion speculatively, based on assumptions about aliasing and without having to worry about recovery. The latter allows the compiler to check these assumptions at runtime with little or no overhead. For this to be feasible, however, we need to extend the hardware-software interface for transactions, making them more *permeable* to software.

In this work, we apply these insights to build a form of alias speculation that can be used by the compiler to perform aggressive optimizations that were not previously possible. Unlike conventional alias analyses, which need to prove aliasing properties, we only need to determine that the aliasing properties are true "most of the time". We can detect the cases when they are incorrect at runtime, and just roll back the execution in those rare cases.

Admittedly, this is not the first time that speculation has been adopted for alias analysis. Speculation has been used in the past in a purely software setting [31], or with the help of hardware in the Itanium Advanced Load Address Table (ALAT) [14, 25, 26] and the Transmeta data speculation support [15, 21]. However, past work has not fully taken advantage of what transactions can potentially offer, including the ability to (i) move store accesses (in addition to load accesses) and (ii) perform multiple optimizations with differing code motion spans in the same atomic region.

Specifically, the ALAT does not use atomic regions, and is only used to speculatively hoist loads across aliasing stores. Moreover, the compiler must place a check instruction at each location in the code where the original load was. Equally important, the compiler

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASPLOS'13, March 16–20, 2013, Houston, Texas, USA.

Copyright © 2013 ACM 978-1-4503-1870-9/13/03. . . $15.00

is in charge of generating the recovery code, which is complex and hampers subsequent optimization and code generation passes.

Transmeta is closest to our proposal in that it supports checkpointing hardware. It provides alias detection hardware to hoist loads to hide load latencies. However, its Code Morphing Software (CMS) x86 binary translator is also limited in that it does not support the movement of store accesses. Moreover, the existing literature on CMS does not explain how multiple optimizations with differing code motion spans could be performed within a single atomic region. In fact, there is only a cursory introduction of the alias-checking instructions in the literature. The specifics of what optimizations are performed and how they are realized is not discussed.

In this paper, we propose a more powerful and flexible environment for alias speculation. The main contributions of this work are as follows:

• It proposes, for the first time, to expose the alias checking capabilities present in upcoming transactional memory systems for the purpose of alias speculation. It also proposes a set of novel ISA extensions to enable this.

• It shows how to do speculative code motion in two new ways. The first one involves moving stores. The second one performs multiple optimizations with differing code motion spans inside the same atomic region.

• It is the first paper to evaluate the benefits of alias speculation using atomic regions in the context of a source-level compiler and not a binary translator. It evaluates LICM, GVN, and PRE optimizations enhanced with alias speculation on the LLVM compiler [23]. They result in an average speedup of 9% in SPEC FP2006 programs and 3% in SPEC INT2006 programs over the baseline LLVM alias analysis.

This paper is organized as follows: Section 2 gives a background; Section 3 presents our alias speculation approach; Section 4 describes our compiler implementation; Section 5 presents examples of optimizations and experimental results; Section 6 discusses applying the approach to other environments; and Section 7 lists related work.

2. Background: Code Motion & Atomic Regions

2.1 Code Motion

Code motion involves moving a piece of code from its original location to a more desirable one, often to minimize the frequency of execution at runtime. Code motion is crucial for many compiler optimizations. A popular example of code motion optimization is Loop Invariant Code Motion (LICM). Other optimizations such as Partial Redundancy Elimination (PRE), Global Value Numbering (GVN), Common Subexpression Elimination (CSE), and Dead Store Elimination (DSE) implicitly involve code motion. For example, in CSE, the compiler moves the redundant expression to the site of the original expression to perform the elimination. If there are any aliasing accesses in the code motion span, the optimization is denied.

The aliasing properties that need to be followed by code motion can be summarized in a few correctness rules. In the following, we describe these rules and apply them to a few optimizations, namely LICM, GVN, and PRE.

2.1.1 Correctness Rules

If code motion M is to be performed on expression E, the following rules need to be followed between the set of addresses read and written by E (R_E and W_E), and the set of address read and written by loads and stores in the code motion span (R_M and W_M).

- **Read Motion Rule ($R_E \cap W_M = \emptyset$).** The values read by E must remain invariant. If a location read by E is updated in the code motion span, E may generate a wrong value in its moved location.
- **Write Motion Rule ($W_E \cap R_M = \emptyset \wedge W_E \cap W_M = \emptyset$).** The value written by E must not be used by any load in the code motion span. Otherwise, the load may produce a wrong value. Also, the code motion must not cause E or any store in its span to write a stale value. If E is moved before an aliasing store, that will cause that store to write a stale value. If E is moved after an aliasing store, that will cause E to write a stale value.

2.1.2 LICM

LICM involves either *hoisting* an expression in the body of a loop to the preheader of the loop, or *sinking* an expression in the body of a loop to the exit block of the loop [41]. If there is a load that repeatedly loads from the same location, the load is hoisted. If there is a load that loads from different locations but the loaded value is not used inside the loop, the load is sunk and only the last load is executed. In both cases, the loaded location should not be modified by any store in the loop (**Read Motion Rule**).

If there is a store instruction in the loop that repeatedly stores to the same location, LICM attempts to promote that location to a register. In effect, this *sinks* the store to the exit block of the loop. For this to be legal, the stored value must not be accessed by any load in the loop and must not be overwritten by any store in the loop (**Write Motion Rule**).

2.1.3 GVN/PRE

GVN attempts to eliminate redundant expressions by assigning a value number to each expression, which is stored in a map, and replacing an expression with a value if it can find one with the same number [5]. PRE also attempts to eliminate redundant expressions but it can eliminate them even when they are partially redundant, meaning that they are redundant only on certain paths to the expression. In this case, PRE makes those expressions fully redundant by performing code motion across the control flow graph [20]. LLVM utilizes a PRE algorithm based on static single assignment (SSA) form that performs GVN simultaneously [40].

Both GVN and PRE are subject to the **Read Motion Rule** in that the value of the expression that is being moved must not change in the course of code motion.

2.2 Hardware Support for Atomic Regions

There are several recent examples of mechanisms or machines that support atomic regions, namely Intel's Transactional Synchronization Extensions [2], IBM's Bluegene/Q [1], IBM's POWER architecture [3], AMD's Advanced Synchronization Facility [10], and Azul's Vega [11]. Atomic regions allow programmers to demarcate a region of code that is guaranteed to execute in isolation with respect to other threads and commit its outcome atomically. Its goal is to help parallel programming. We wish to use the same hardware support for the purpose of alias speculation.

All atomic region implementations incorporate two primitives that are necessary to guarantee atomicity and isolation: checkpointing with speculative buffering and conflict detection. The first primitive takes a register checkpoint at the beginning of an atomic region and buffers all writes to memory. If isolation is compromised before the end of the atomic region, the buffered writes are discarded and the register checkpoint restored. If isolation is successful, the buffered writes are made visible to other threads all at once when the atomic region commits.

The conflict detection primitive guarantees isolation by keeping track of addresses read in the atomic region (the read-set) and written in the atomic region (the write-set). All remote memory accesses check the read-set and the write-set for memory conflicts that compromise the isolation of the atomic region. The read-set and write-set are typically recorded by setting speculatively read (SR) bits and speculatively written (SW) bits in a cache. There is one SR bit and one SW bit per each cache line to record whether that line belongs to the read-set and/or write-set of an atomic region. Besides conflict detection, SW bits are also used to mark cache lines that need to be buffered for rollback purposes. This type of cache organization is called a speculative cache.

In this paper, we assume an atomic region implementation using a speculative cache. However, our proposal can be applied to any hardware-based atomic region implementation.

3. Proposed Alias Speculation Approach

3.1 Overview

We propose to make hardware support for atomic regions more permeable to software through ISA extensions for the purposes of alias speculation. Specifically, we want to expose to software the *memory conflict detection* facility of atomic region hardware that is hidden under the covers in current transactional memory systems. Currently, the conflict detection hardware is only used to guarantee the isolation of a transaction with respect to remote accesses. We wish to repurpose this hardware to detect memory conflicts, or aliases, between accesses of the *same thread*, and expose this hardware to the software to perform alias checks.

To leverage this new capability, pairs of references that do not alias "most of the time" are speculated upon during compile time as not aliasing. The compiler communicates its assumptions to the hardware using our new ISA extensions. After the assumptions have been communicated, the alias detection hardware is in charge of automatically verifying them at runtime. Verification failure results in the atomic region being rolled back and a version of the code without speculation being executed in place of the atomic region. This approach allows the compiler to perform code movements and optimizations that were previously impossible.

In designing the new ISA extensions and compiler algorithms, we want to achieve four goals:

- The movement of both loads and stores should be supported.
- All optimizations that can be improved through alias speculation should be enabled to their fullest extent possible.
- The optimizations should be achieved with as little instrumentation overhead as possible.
- The ISA extensions should only expose pre-existing hardware structures and functionality that is required for transactions, and not add significant complexity to the hardware.

A key design decision is where to place the atomic regions, since it dictates which pairs of aliased accesses can be speculated upon. Note that placing atomic regions carelessly results in performance degradation due to frequent rollbacks or the overhead of the atomic region instructions themselves. Hence, we decide to place atomic regions around loops. Wrapping loops in atomic regions provides ample range for optimizations such as LICM and PRE to perform large-scale redundancy elimination. Also, the benefit from optimizations performed inside loops accumulates at each iteration, and can easily offset the overhead of atomic region instrumentation. This is hard to attain in straight-line code.

Once we place an atomic region around a given loop, we want to enable as many code motions as possible. The code motions might span the entire loop, as in the case of LICM, or they might span a subset of instructions in the loop, as in the case of GVN and PRE. For each code motion, we need to make sure that the moved expression does not alias with any intervening loads or stores that can compromise the correctness rules laid out in Section 2.1.1. Specifically, for each code motion span M, ideally we need to know the read (R_M) and write (W_M) sets, to check that they do not alias illegally with the R_E and W_E of the moved expression E.

In reality, existing transactional memory hardware keeps track of just two sets of addresses for an atomic region: the set of addresses read (R_{AR}) and those written (W_{AR}) in the atomic region (AR). Alias checking against R_{AR} and W_{AR} instead of against R_M and W_M always guarantees correctness, since $R_M \subseteq R_{AR}$ and $W_M \subseteq W_{AR}$ always holds. However, the imprecision of the check can result in extraneous failures at runtime and frequent rollbacks of the atomic region. In the end, the cost of rollbacks can prevent some optimizations from being performed.

We propose to mitigate the problem of imprecision by manipulating the Speculative Read (SR) bits of a transaction at will, so that they constitute read address sets other than R_{AR}. However, we must be prepared to give up on isolation guarantees for the atomic region. R_{AR} is only needed for conflict detection and isolation of the atomic region. Since we are not using the atomic region for synchronization purposes like in transactional memory, we can give up on isolation and repurpose the SR bits to mark only read addresses that we want to monitor for alias checking.

Note that we still need the transaction's atomicity guarantees, since we must be able to roll back the atomic region on alias speculation failure. Hence, we are forbidden from manipulating the Speculative Written (SW) bits because we still need to buffer writes for checkpointing and rollback. The implications of forfeiting isolation on the rest of the system are discussed later in Section 6.2.

In the next sections, we discuss how to reduce conservatism by manipulating the SR bits and using special check instructions.

3.2 DeAliaser ISA Extensions

Table 1 shows the new Instruction Set Architecture (ISA) extensions that we add for DeAliaser. The instructions are categorized into three types: those that control the placement of atomic regions, those that control what memory locations are monitored for alias checking, and those that check the monitored locations.

Controlling Atomic Regions. The instructions *begin_atomic_opt* and *end_atomic_opt* begin and end an atomic region for optimization. The atomic region created by these instructions is exactly the same as a regular atomic region in transactional memory, except that the operation of the SR bits is changed. Now, regular loads do not set SR bits; only the special loads described below do so.

A rollback of the atomic region triggers a jump to a program counter (PC) given as an operand to *begin_atomic_opt*. It is the PC of the safe version: a conservative replica of the code in the atomic region that does not assume any speculation, and is always safe to execute. This safe version support is standard in all transactional systems.

Controlling Monitored Locations. The set of cache lines with the SR bit set and the set of cache lines with the SW bit set are the set of monitored read and write locations, respectively. The *load.r* instruction loads data into a register and sets the SR bit of the line in the cache containing the data. The *clear.r* instruction takes an address and clears the SR bit of the line in the cache corresponding to that address. Therefore, the *load.r* and *clear.r* instructions start and end the monitoring of a loaded location. With them, we can flexibly expand and shrink the set of read-monitored locations as the need arises for alias checking. This gives us the freedom to alias check multiple spans of code motion.

We do not have the corresponding operations for SW bits. DeAliaser cannot manipulate them because they are used for determining what to roll back if the atomic region fails. Thus, we cannot control the set of write-monitored locations, which is always the entire set of locations written by the atomic region so far. This can be a source of imprecision when doing alias checks.

Checking Monitored Locations. The *storechk.(r/w/rw)* and *loadchk.(r/w/rw)* instructions, in addition to storing to a location or loading from it, perform a check of the speculative bits in the cache line containing the location. The instruction postfix indicates which speculative bit is checked: *r* checks the SR, *w* checks the SW, and *rw* checks both. If the checked bit is set (or at least one of the two checked bits is), the hardware signals an alias check failure. A check failure means that this store or load access does alias with a monitored location, violating a *no alias* assumption made by the compiler. This results in a rollback of the atomic region and a jump to the safe version.

The *storechk.(r/w/rw)* instructions set the SW bit as is required of all store instructions, and they do so after performing the checks. Notably, the *loadchk.r* and *loadchk.rw* instructions also set the SR bit after the checks, and this is by design. They are designed in this way because the SR bit only needs to be checked by a load instruction when it *also* wants to set it immediately afterwards, as will be seen later.

Instruction	Description
begin_atomic_opt PC	Begins an atomic region for optimization. It creates a new register checkpoint and starts buffering memory accesses. *PC* is the program counter of the beginning of the safe version.
end_atomic_opt	Ends an atomic region for optimization. It commits all buffered accesses.

(a)

Instruction	Description
load.r r1, addr	Loads location *addr* into register *r1* just like a regular load. In addition, it sets the SR bit in the cache line containing *addr* for monitoring.
clear.r addr	Clears the SR bit in the cache line containing *addr*, hence ending monitoring.

(b)

Instruction	Description
storechk.(r/w/rw) r1, addr	Stores register *r1* into location *addr* just like a regular store. In addition, it checks the SR bit, the SW bit, or both in the cache line containing *addr*, depending on whether the postfix of the opcode is *r*, *w*, or *rw*. If the checked bit (or at least one of the two checked) is set, the atomic region is rolled back and execution jumps to the safe version.
loadchk.(r/w/rw) r1, addr	Loads location *addr* into register *r1* just like a regular load. In addition, it checks the SR bit, the SW bit, or both in the cache line containing *addr*, depending on the postfix of the opcode. If the checked bit (or at least one of the two checked) is set, the atomic region is rolled back and execution jumps to the safe version. Lastly, *loadchk.r* and *loadchk.rw* set the SR bit after it is checked.

(c)

Table 1. Extensions to the ISA to control atomic regions (a), control monitored locations (b), and check monitored locations (c).

3.3 Alias Speculation and Code Motion

The way a conventional compiler applies the code motion rules of Section 2.1.1 is necessarily conservative because of two reasons. First, even if the rules hold "most of the time", the compiler must guarantee correct execution for the corner cases. Secondly, even if the rules hold all the time, static analyses are fundamentally limited in what they can prove.

We use DeAliaser to circumvent having to prove the rules. Instead, we check them at runtime, thereby enabling more code motion at compile time. DeAliaser checks a modified version of the set of rules which, while sometimes more conservative, is still sufficient to guarantee the original set of rules. We explain how the rules are modified and checked for four types of code motion optimizations that are allowed by DeAliaser: hoisting loads, sinking loads, hoisting stores, and sinking stores. We also describe how to sink the *clear.r* instructions introduced by DeAliaser itself, in order to minimize overhead. Refer to Figure 1 while we go over each case.

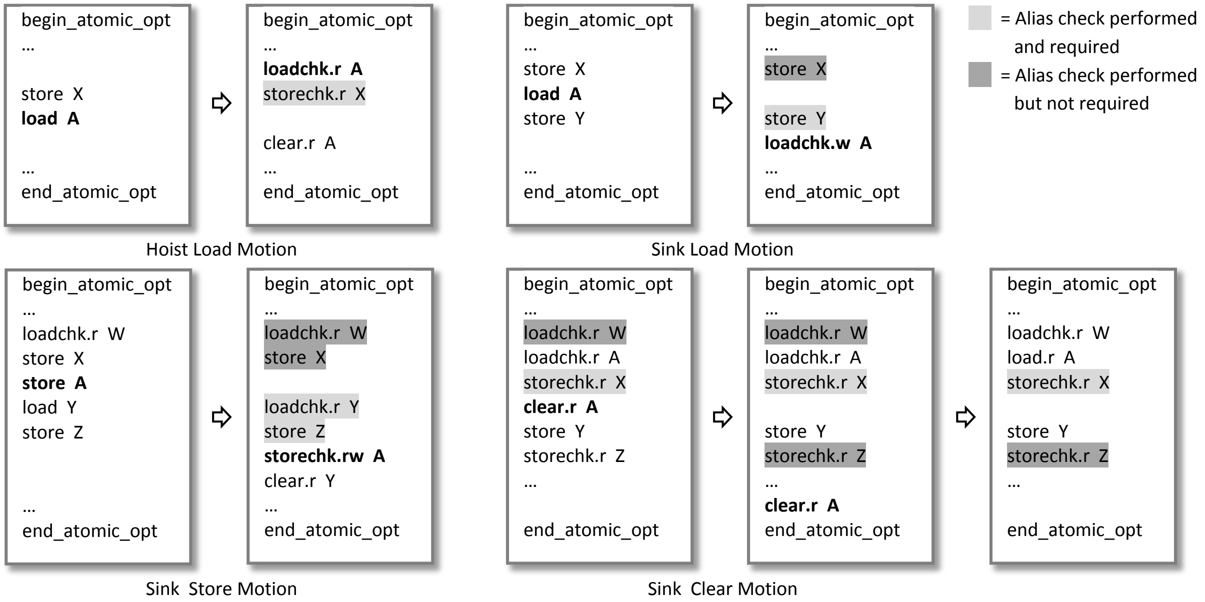
Figure 1(a) illustrates how each code motion optimization is performed by example. For each case, the left hand side shows the code in its original state and the right hand side shows it after transformation. The moved instruction itself is marked in bold, and the register operands for all instructions are omitted for brevity. In each example, the loads and stores in grey denote the instructions that are checked by DeAliaser for aliasing against the instruction moved. Of these, the light grey ones are those checks that are performed and are necessary for correctness, while the dark grey ones are those checks that are performed as a side effect because of the imprecision in the DeAliaser hardware, and are not necessary for correctness. The table in Figure 1(b) describes each motion optimization in more general terms. Column 2 in the table lists the compiler actions that need to be performed after each motion. Column 3 describes how the alias checks guarantee the correctness of the motion. Finally, Column 4 states whether the alias checks are precise.

For the purposes of explanation, we adopt the following terminology. As before, M is the set of instructions in the span of the code motion performed on expression E, which are loads and stores that *may alias* with E. R_E is the address read by the moved load and W_E is the address written by the moved store. R_M and W_M are the set of addresses read and written, respectively, by accesses in M. $R_{Monitored}$ and $W_{Monitored}$ are the set of read and written addresses, respectively, currently being monitored in the atomic region. As mentioned before, $R_{Monitored}$ can be expanded or shrunk using the *load.r*, *loadchk.r*, *loadchk.rw*, and *clear.r* instructions. However, $W_{Monitored}$ is always the entire set of addresses written in the atomic region so far. Finally, we refer to a load or a store that sets the SR or SW bits as a *monitored* load or store; a load or a store that checks the bits is a *checking* load or store.

3.3.1 Hoisting Loads

In the top left example of Figure 1(a), we hoist *load A* above *store X*. After hoisting a load, we must monitor all the stores that may-alias in the span M of the motion. Our transformation involves: changing the hoisted load to *loadchk.r*, changing all the store instructions in M that may alias with the load to *storechk.r*, and placing a *clear.r* at the end of M to stop monitoring the load. We use *loadchk.r* instead of *load.r* to detect any interference with another unrelated code motion optimization. Specifically, if that other optimization had a monitored load that aliased with our load, both loads would set the same bit. Then, the first *clear.r* operation found would clear the SR bit, incorrectly turning off monitoring for both loads. With *loadchk.r*, we ensure that we detect this problem when we start monitoring, and roll back the atomic region.

The **Read Motion Rule** for moving loads ($R_E \cap W_M$) is trivially guaranteed, as shown in Figure 1(b). This is because checking against the set of addresses written by M using *storechk.r* instructions, referred to as $W(chk.r)_M$ in the table, is equivalent by construction to checking against W_M. The checks are not only correct but precise, since a *storechk.r* check failure always means that the rule was violated. The only potential source of extraneous rollbacks comes from the use of the *loadchk.r* instruction. We will see that


= Alias check performed and required
= Alias check performed but not required
Hoist Load Motion
Sink Load Motion
Sink Store Motion
Sink Clear Motion

```
begin_atomic_opt
…

store  X
load  A


…
end_atomic_opt
```

```
begin_atomic_opt
…
loadchk.r  A
storechk.r  X

clear.r  A
…
end_atomic_opt
```

```
begin_atomic_opt
…
store  X
load  A
store  Y

…
end_atomic_opt
```

```
begin_atomic_opt
…
store  X

store  Y
loadchk.w  A
…
end_atomic_opt
```

```
begin_atomic_opt
…
loadchk.r  W
store  X
store  A
load  Y
store  Z


…
end_atomic_opt
```

```
begin_atomic_opt
…
loadchk.r  W
store  X

loadchk.r  Y
store  Z
storechk.rw  A
clear.r  Y
…
end_atomic_opt
```

```
begin_atomic_opt
…
loadchk.r  W
loadchk.r  A
storechk.r  X
clear.r  A
store  Y
storechk.r  Z
…

end_atomic_opt
```

```
begin_atomic_opt
…
loadchk.r  W
loadchk.r  A
storechk.r  X

store  Y
storechk.r  Z
…
clear.r  A
end_atomic_opt
```

```
begin_atomic_opt
…
loadchk.r  W
load.r  A
storechk.r  X

store  Y
storechk.r  Z
…

end_atomic_opt
```

(a)

Code Motion	Action	Correctness Guarantee	Precise?
Hoist Load	Change *load* to *loadchk.r*. Change all may-alias stores in M to *storechk.r*. Insert *clear.r* at end of span to stop monitoring.	$R_E \cap W_M = R_E \cap W(chk.r)_M = \emptyset$	Yes
Sink Load	Change *load* to *loadchk.w*.	$R_E \cap W_{Monitored} = \emptyset, W_{Monitored} \supseteq W_M$	No
Hoist Store	Change all may-alias loads in M to *loadchk.w*. Change all may-alias stores in M to *storechk.w*.	$W_{Monitored} \cap R(chk.w)_{AR} = \emptyset \wedge W_{Monitored} \cap W(chk.w)_{AR} = \emptyset$, $W_{Monitored} \supseteq W_E, R(chk.w)_{AR} \supseteq R_M$, $W(chk.w)_{AR} \supseteq W_M$	No
Sink Store	Change *store* to *storechk.r/w/rw* depending on whether you are speculating on load aliases, store aliases, or both. If speculating on load aliases, change all may-alias loads in M to *loadchk.r* (unless they are already *loadchk.r/rw*). For all newly monitored loads, insert *clear.r* to stop monitoring after the store.	$W_E \cap R_{Monitored} = \emptyset \wedge W_E \cap W_{Monitored} = \emptyset$, $R_{Monitored} \supseteq R_M, W_{Monitored} \supseteq W_M$	No
Sink Clear	If *clear.r* is sunk to end of atomic region, remove it. Change *loadchk.r* to *load.r* or vice versa for affected monitored loads.	Expanding a monitoring span never compromises correctness.	No

(b)

Figure 1. Code motions supported by DeAliaser shown through examples (a) and explained in general terms (b).

the *loadchk.r* can often be converted to a check-less *load.r* when sinking *clear* instructions.

3.3.2 Sinking Loads

In the top right example of Figure 1(a), we sink *load A* below *store Y*. The initial code also includes an earlier *store X*. Our transformation involves changing the sunk load to *loadchk.w*. This load now checks that it does not alias with any of the writes executed in the atomic region so far. This includes the *store Y*, but it also includes all the previous stores in the atomic region such as *store X*. As shown in Figure 1(b), since M covers a subset of the instructions in the atomic region, the **Read Motion Rule** ($R_E \cap W_M$) is trivially guaranteed to be enforced. However, the check is not precise, potentially leading to extraneous rollbacks.

3.3.3 Hoisting Stores

We do not show an example of hoisting stores because of space limitations, but we still describe the actions performed and correctness guarantees in Figure 1(b). After hoisting the store, all may-alias loads in M are changed to *loadchk.w* instructions and all may-alias stores in M are changed to *storechk.w* instructions.

These transformations guarantee the **Write Motion Rule** ($W_E \cap R_M \wedge W_E \cap W_M$) for stores, but with potentially significant imprecision. In Section 3.3.1, we could selectively monitor hoisted loads by changing the load to *load.r* or *loadchk.r* and setting the SR bit. The same cannot be done with stores and SW bits, however, and we have no choice but to check all written locations since the beginning of the atomic region ($W_{Monitored}$) — instead of just W_E. Also, since SW bits cannot be cleared, we cannot stop monitoring at the end of M. If there are any *loadchk.w*, *loadchk.rw*, *storechk.w*, or *storechk.rw* instructions after the end of M, DeAliaser still performs the checks. Hence, we check against the entire set of locations accessed with *loadchk.w/rw* (called $R(chk.w)_{AR}$ in the table), or *storechk.w/rw* (called $W(chk.w)_{AR}$ in the table) in the atomic region following the hoisted store.

Hoisting stores is the code motion that suffers the most from imprecision. Fortunately, most compiler optimizations in use today do not use it.

3.3.4 Sinking Stores

In the bottom left example of Figure 1(a), we sink *store A* below *load Y* and *store Z*. The initial code also includes two earlier instructions: a *loadchk.r* and a store. Our transformation involves three parts. First, we change the sunk store to the appropriate flavor of *storechk.r/w/rw* instruction, depending on whether the motion is speculating on load aliases, store aliases, or both. Second, if we are speculating on load aliases, we change all the may-alias loads in M to *loadchk.r*. If some of these loads are already *loadchk.r/rw*, we do not change them. Third, we place *clear.r* instructions immediately after the *storechk.r/w/rw* for all the newly monitored loads, to stop monitoring after the store.

Figure 1(b) shows the correctness guarantee for this motion speculating on both load and store aliases. The ones for speculating on either only loads or only stores are obtained by simply removing one term. We see that the **Write Motion Rule** ($W_E \cap R_M \wedge W_E \cap W_M$) is trivially guaranteed because the set of addresses accessed in M is a subset of monitored addresses.

This check is not precise. The monitored stores are all the writes before the sunk store. The monitored loads are all the *loadchk.r/rw* ones before the sunk store, including those that were in the original code before the transformation. In Figure 1(a), we see that there are unnecessary checks against *loadchk.r W* and *store X*.

3.3.5 Sinking Clears

After all the movement of loads and stores is done, the last step is to remove overhead by sinking the *clear.r* instructions that have been added in the process. They are either sunk to a less frequently executed location or to the end of the atomic region, where they can be trivially eliminated. As we sink a *clear.r*, we are expanding the monitoring span of a *load.r* or *loadchk.r/rw* and this is always correct.

As the *clear.r* instructions are moved and eliminated, some *loadchk.r* instructions that used to interfere with each other no longer do. In this case, the *loadchk.r* instructions can be safely converted to *load.r* instructions. For example, consider the bottom right example of Figure 1(a). Initially, *loadchk.r A* had to be checked against *loadchk.r W* to make sure *clear.r A* does not clear the SR bit set by the latter erroneously. Once *clear.r A* is eliminated, *loadchk.r A* can safely be converted to *load.r A*. Now *load.r A* no longer checks against *loadchk.r W* and, therefore, can no longer cause extraneous rollbacks.

Vice versa, movement of *clear.r* instructions can cause *load.r* instructions that used to not interfere with each other to do so and in these cases, they have to be reverted back to *loadchk.r* instructions.

3.3.6 Discussion

Even though having only a single set of SR and SW bits in the atomic region leads to some imprecision, the imprecision is controlled by manipulating the SR bits and also by checking only relevant memory accesses. Importantly, correctness is still guaranteed even when there are multiple code motions. For the optimizations and applications that we studied, our experience is that this imprecision does not significantly restrict code motion or cause extraneous failures.

Also, having just one SR and SW bit per cache line leads to another type of imprecision — that due to false sharing. But false sharing does not compromise the correctness of our transformations; at worst, it causes extraneous rollbacks. The only case that needs special attention is when a *loadchk.r* instruction is converted to a *load.r* instruction because it does not interfere anymore with other *loadchk.r/rw* or *load.r* instructions. One might mistakenly think that proving a *no alias* relationship between two loads is sufficient to guarantee non-interference. But even if two loads do not alias per se, they might alias through false sharing. Hence, converting *loadchk.r* instructions based on the no alias assumption may still cause a problem when the *clear.r* is executed. Therefore, the compiler needs to either take false sharing into account when performing the conversion or not perform the conversion.

The only case where instructions are added for instrumentation purposes (other than *begin_atomic_opt* and *end_atomic_opt*) is when *clear.r* instructions are needed. In addition, even these can often be eliminated by sinking them to the end of the atomic region. Consequently, DeAliaser introduces negligible instrumentation overhead.

Overall, the DeAliaser design successfully allows multiple code motions inside an atomic region while minimizing instrumentation overhead. At the same time, the added hardware complexity over transactions is kept very small.

3.4 Application of Code Motion to Optimizations

While there are numerous compiler optimizations that can benefit from DeAliaser, we focus on LICM, GVN, and PRE to demonstrate its usefulness. These optimizations are already implemented as optimization passes in the LLVM compiler that we use for evaluating our scheme. GVN and PRE are performed simultaneously, as part of a single GVN/PRE pass. Hence, from here on, we use GVN to refer to the GVN/PRE pass with both optimizations.

LICM involves the hoisting and sinking of loads, and the sinking of stores, as described in Section 2.1.2. Hoists are moved to the preheader of the loop and sinks to the exit block of the loop. Since atomic regions are instrumented around loops, the span of code motion and the span of the atomic region are often identical, namely the loop. In this case, R_M and W_M are simply the entire set of memory locations read or written in the atomic region, and $R_{Monitored}$ and $W_{Monitored}$ are equivalent to these. According to the Hoist Load, Sink Load, and Sink Store rows of Figure 1(b), this means that no imprecision is introduced due to the alias checks. Also, since monitoring always ends at the end of the atomic region, *clear.r* instructions can always be eliminated.

GVN/PRE involves the hoisting and sinking of loads in order to eliminate computation of identical expressions, or to find the location where an expression can be computed less frequently. Unlike LICM, the code motion done by GVN might not span the entire loop if the redundancy is found within the body of the loop. Hence, imprecision in the alias checks may lead to some conservatism. However, in the applications studied, we find that this conservatism does not lead to any significant restriction in optimizations. Also, like in LICM, we are almost always able to eliminate the *clear.r*.

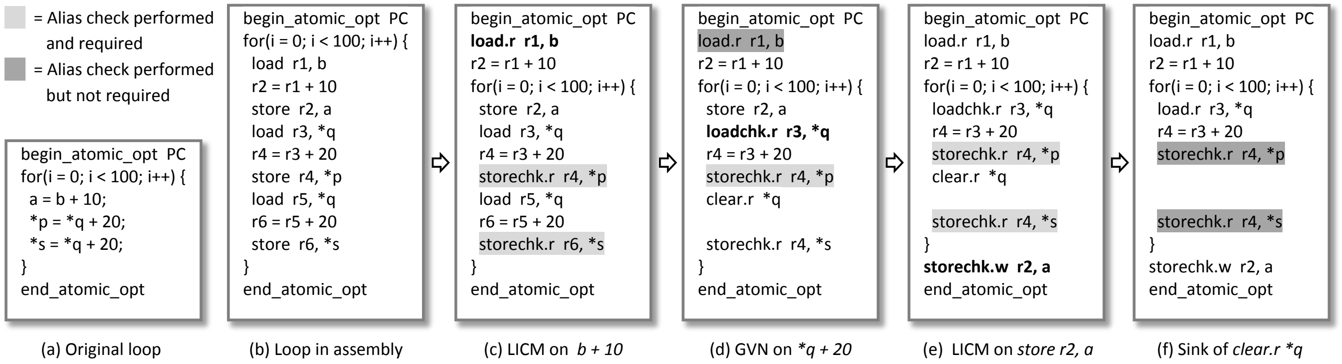
3.5 Transforming a Loop for Optimization

Before a loop can be optimized using DeAliaser, it needs to be instrumented with an atomic region. For this, *begin_atomic_opt* is inserted at the preheader of the loop and an *end_atomic_opt* is inserted at the exit block of the loop, so that the loop is enwrapped completely in an atomic region. Also, a safe version replica of the loop is created before any speculative optimizations are done. The safe version has no speculation, and no atomic regions are required. The failure of alias checks at any point in the atomic region triggers a rollback and a jump to the safe version. Once the safe version completes, execution jumps to the code after the loop.

The failure of an atomic region can also come from the overflow of the hardware resources that buffer the speculative writes and the read-monitored locations. To prevent overflows, we perform loop blocking, which reduces the footprint of an atomic region. After blocking, an atomic region is placed around the inner blocked loop. In the rare case where an atomic region still overflows, it can roll back and execute the safe version just like in the case of check failure.

3.6 Illustrative Example of LICM and GVN

We finish with an example of DeAliaser using LICM and GVN to optimize a loop. Figure 2(a) shows the initial loop. To see the capabilities of DeAliaser, assume that (i) *a* and *b* may alias with **p* and **s*; and (ii) **p*, **q*, and **s* may alias with each other. Ideally, we would like to perform three code motion optimizations: 1) use LICM to hoist $b + 10$ to the beginning of the loop; 2) use


▨ = Alias check performed and required
■ = Alias check performed but not required

```
begin_atomic_opt  PC
for(i = 0; i < 100; i++) {
  a = b + 10;
  *p = *q + 20;
  *s = *q + 20;
}
end_atomic_opt
```

(a) Original loop

```
begin_atomic_opt  PC
for(i = 0; i < 100; i++) {
  load  r1, b
  r2 = r1 + 10
  store  r2, a
  load  r3, *q
  r4 = r3 + 20
  store  r4, *p
  load  r5, *q
  r6 = r5 + 20
  store  r6, *s
}
end_atomic_opt
```

(b) Loop in assembly

```
begin_atomic_opt  PC
load.r  r1, b
r2 = r1 + 10
for(i = 0; i < 100; i++) {
  store  r2, a
  load  r3, *q
  r4 = r3 + 20
  storechk.r  r4, *p
  load  r5, *q
  r6 = r5 + 20
  storechk.r  r6, *s
}
end_atomic_opt
```

(c) LICM on *b + 10*

```
begin_atomic_opt  PC
load.r  r1, b
r2 = r1 + 10
for(i = 0; i < 100; i++) {
  store  r2, a
  loadchk.r  r3, *q
  r4 = r3 + 20
  storechk.r  r4, *p
  clear.r  *q

  storechk.r  r4, *s
}
end_atomic_opt
```

(d) GVN on **q + 20*

```
begin_atomic_opt  PC
load.r  r1, b
r2 = r1 + 10
for(i = 0; i < 100; i++) {
  loadchk.r  r3, *q
  r4 = r3 + 20
  storechk.r  r4, *p
  clear.r  *q

  storechk.r  r4, *s
}
storechk.w  r2, a
end_atomic_opt
```

(e) LICM on *store r2, a*

```
begin_atomic_opt  PC
load.r  r1, b
r2 = r1 + 10
for(i = 0; i < 100; i++) {
  load.r  r3, *q
  r4 = r3 + 20
  storechk.r  r4, *p


  storechk.r  r4, *s
}
storechk.w  r2, a
end_atomic_opt
```

(f) Sink of *clear.r *q*

Figure 2. Example of a loop optimized with LICM and GVN using alias speculation through DeAliaser. Assume that (i) *a* and *b* may alias with **p* and **s*; and (ii) **p*, **q*, and **s* may alias with each other.

GVN to remove the second redundant computation of $*q + 20$; and 3) use LICM to sink the store to a to the end of the loop. Unfortunately, without DeAliaser, the compiler cannot perform any of these optimizations because of potential aliasing.

With DeAliaser, the compiler can do them all. To see how, we start with the original loop in assembly, as shown in Figure 2(b). The next few charts show each of the steps of the transformation in sequence. At each step, we show the moved memory access in bold, and the memory accesses checked by DeAliaser against the moved access in gray, using the same conventions as in Figure 1(a).

Figure 2(c) uses LICM to hoist $b + 10$ to the beginning of the loop. As per Section 3.3.1, we change all the store instructions in the loop that may alias to *load b* to *storechk.r*. This includes *store *p* and *store *s*. We change *load b* to *load.r b* rather than *loadchk.r b* because there is no other code motion optimization that can interfere. Also, we do not place a *clear.r* because it would go to the end of the loop.

Figure 2(d) uses GVN to remove the second, redundant $*q + 20$. We hoist the second *load *q* and its computation above the *storechk.r *p*. The two hoisted instructions are combined with the ones that generated the same expression. In this load hoist, we do not change any stores because the one in the code motion span was already *storechk.r *p*. However, we need to replace the hoisted, combined *load *q* with *loadchk.r *q*. We use *loadchk.r* instead of *load.r* since it may alias with *load.r b* through false sharing (even if the compiler alias analysis thinks otherwise). In addition, we place the *clear.r *q* at the end of the code motion span.

Figure 2(e) uses LICM to sink *store a* to the end of the loop. Based on our assumptions in the example, we only need to speculate on store aliases. Hence, as per Section 3.3.4, the *store a* is sunk and replaced with *storechk.w a*. This instruction will be checked against all the stores in the loop.

Finally, Figure 2(f) reduces overhead by sinking and then eliminating the *clear.r *q* instruction. As per Section 3.3.5, we need to consider all the relevant new instructions added to the monitoring span of *loadchk.r *q*. We are including *storechk.r *s*, which may now produce extraneous failures. We are also adding *storechk.r *p* from *other* iterations, which may also create extraneous failures. Finally, because we have removed the *clear.r*, we can again use *load.r *q* rather than *loadchk.r *q*. The reason we had used *loadchk.r *q* was to ensure that the *clear.r* did not reset the SR bit set by *load.r b*. Retaining *loadchk.r *q* can only generate extraneous failures.

Beyond the false sharing issue with *load.r b*, the sinking and removal of *clear.r* is the only motion that adds imprecision to the checks. This motion is performed in the hope that the two stores will not alias with the load of $*q$ at runtime. However, if this does not turn out to be so, the *clear.r* must stay where it is.

Overall, the example shows how the compiler can leverage DeAliaser to perform multiple code motions in a single atomic region. A loop iteration started with 9 instructions and ended up with 4. Not a single instrumentation instruction was added.

4. Compiler Implementation

4.1 The Cost-Benefit Model

Until now, we have focused on how DeAliaser enables optimizations. But an equally important question is when are these optimizations beneficial. One can use heuristics for this purpose [12], but a more precise method is to profile the application. Fortunately, we can leverage the same hardware support that we use for DeAliaser to collect a profile with high efficiency. To decide whether to optimize a particular loop using DeAliaser, we estimate the following measures, in number of dynamic instructions for the average execution of the AR:

D: Number of instructions in the AR after applying DeAliaser.
I: Number of DeAliaser instrumentation instructions in the AR.
r: AR rollback rate.
R: Reduction in AR instructions through DeAliaser application.

For DeAliaser to be desirable, it should be that $R > I + D \times r$. It means that the benefit of the optimizations enabled by DeAliaser should outweigh the overhead due to instrumentation and rollbacks due to failures.

The number of dynamic instructions in an atomic region and the number of dynamic instructions reduced by optimizations can be obtained by branch profiling [43] or by taking measurements using hardware performance counters while profiling. DeAliaser instruction overhead is the two instructions *begin_atomic_opt* and *end_atomic_opt* at the beginning and end of the loop, plus any *clear.r* instructions that could not be eliminated.

There are two reasons why an atomic region can roll back: buffer overflows and alias speculation failures. We can eliminate virtually all of the buffer overflows through loop blocking. We measure the rate of remaining rollbacks due to speculation failures by maintaining two counters per atomic region while profiling: a success counter that is incremented at each successful commit and a failure counter that is incremented at every jump to the safe version.

We perform the optimizations incrementally for each code motion that requires speculation, and take profiling measurements to decide whether the motion was beneficial. All loops in an application can be profiled simultaneously in a single run. Hence, the number of profiling runs needed is the maximum number of speculative optimizations performed for a single loop.

4.2 Compiler Support

We modified the LLVM compiler [23] release 2.8 to implement DeAliaser. Considering how much effort it takes to correctly implement a complex alias analysis, the modifications that we did are quite straightforward. We performed four tasks in the compiler. One was to write a blocking pass for loops. We also wrote a pass that inserts loops in atomic regions, and creates the safe versions for the loops. Next, we modified the optimization passes to perform speculative code motions within the atomic regions and insert DeAliaser alias checking instructions appropriately. Lastly, we wrote a simple profiler pass that adds success and failure counters for each atomic region and generates a profile.

5. Evaluation

5.1 Experimental Setup

For our experiments, we use the applications in the SPEC INT2006 and SPEC FP2006 suites, except for *416.gamess* and *481.wrf*, which are in Fortran and our infrastructure does not handle well. We compile the applications using the modified LLVM compiler and, for each of the applications, produce three types of binaries: *BaselineAA*, *DSAA*, and *DeAliaser*.

BaselineAA is built by running LICM and GVN in sequence, using the default *Basic Alias Analysis* provided with the LLVM compiler, after running the standard -O3 set of optimizations. Basic Alias Analysis is what most applications compiled by LLVM use today. It is an aggressive, yet scalable alias analysis that uses knowledge about heap, stack, and global memory allocations, and structure field information.

DSAA is built by running LICM and GVN using the *Data Structure Alias Analysis*, which was first proposed by Lattner et al. [24]. Data Structure Alias Analysis is the most advanced alias analysis that has been implemented on LLVM and is shown to be more precise [24] than either Steensgaard's [37] or Andersen's [6] alias analyses. It is a heap-cloning algorithm that is flow-insensitive but context-sensitive and field-sensitive. It does unification to reduce memory usage, but we find that it is still not able to compile certain programs due to memory overflows. In fact, it fails to compile four benchmarks, namely *400.perlbench*, *403.gcc*, *483.xalancbmk*, and *445.gobmk*, with 4 GB of memory (the limit on our systems). Hence, we are not able to show results using *DSAA* for these four codes.

DeAliaser is built by running LICM and GVN using DeAliaser support to speculate on aliases. The Basic Alias Analysis is run before speculating using DeAliaser, to filter out the easily-provable alias relationships. Only the remaining may-aliases after running the Basic Alias Analysis are candidates for speculation. Also, loops with many iterations are blocked appropriately so that they do not overflow the speculative cache.

We run the applications on a cycle-level architecture simulator based on SESC [35], and built on top of Pin [27]. We model a uniprocessor desktop with an out-of-order core extended with DeAliaser instructions, and two levels of caches. The simulator simulates all latencies in detail, including memory latencies. It simulates the speculation resources needed to enable atomic regions. We use SR and SW bits at the granularity of cache lines, which is 64 bytes. Table 2 shows the parameters of the hardware.

The speculative L1 cache buffers all the speculatively-written data within an atomic region, as well as all the read-monitored cache lines (those with SR=1). The SR bit and SW bit in each cache line can be accessed and controlled by the DeAliaser ISA extensions. If the L1 cache overflows or an alias check fails, the atomic region is rolled back and the safe version is executed. Speculatively written cache lines are either marked non-speculative when the atomic region commits or are invalidated if the atomic region aborts. Read-monitored cache lines are unmarked when the atomic region either commits or aborts, or when a *clear.r* instruction is executed. In one of the experiments, we also use word-granularity SR and SW bits.

```
Processor:
  2-issue wide, out-of-order superscalar
  Functional units: 2 Int, 1 Ld/St, 1 Br, 2 FP
  I-window: 44 entries; Ld/St queue: 26 entries
Speculative L1 Cache:
  Round trip: 2 cycles
  Size, line, assoc.: 32KB, 64B, 8-way
  Speculative Rd/Wr bits: line granularity
L2 Cache:
  Round trip: 10 cycles
  Size, line, assoc.: 512KB, 64B, 8-way
Memory round trip: 300 cycles; Bus with: 32B
```

Table 2. Parameters of the hardware simulated.

We also use the simulator mentioned above to run the alias profiling phase for *DeAliaser*. We use the *train* input set provided in SPEC CPU2006 for profiling, and the *ref* input set for performance measurement. For the programs we tested, we find that the results of alias analysis profiling are not very sensitive to the input set.

5.2 Optimization Examples

We start the evaluation by showing two examples of real code where *DeAliaser* produces more efficient code than *BaselineAA* or *DSAA*. Figure 3 shows one loop in *433.milc* optimized using LICM, and Figure 4 shows another loop in *410.bzip2* optimized using a combination of LICM and GVN. In both cases, we show the loop before and after applying *DeAliaser*.

```
for(i=0;i<3;i++) {
  for(j=0;j<3;j++) {
    ...
    br=b->e[j][0].real;
    ...
    c->e[i][j].real=cr;
  }
}
```

(a)

```
for(i=0;i<3;i++) {
  ...
  br0=b->e[0][0].real;
  br1=b->e[1][0].real;
  br2=b->e[2][0].real;
  ...
  c->e[i][0].real=cr0;
  c->e[i][1].real=cr1;
  c->e[i][2].real=cr2;
}
```

(b)

```
begin_atomic_opt(Safe_Loop_PC)
load.r  real0 , b->e[0][0].real;
load.r  real1 , b->e[1][0].real;
load.r  real2 , b->e[2][0].real;
for(i=0;i<3;i++) {
  ...
  br0 = real0;
  br1 = real1;
  br2 = real2;
  ...
  storechk.r  cr0 , c->e[i][0].real
  store       cr1 , c->e[i][1].real
  store       cr2 , c->e[i][2].real
}
end_atomic_opt
```

(c)

Figure 3. Loop in function mult_su3_na(...) of *433.milc* before transformation (a), after loop unrolling (b), and after applying *DeAliaser* with LICM (c).

BaselineAA, *DSAA*, and *DeAliaser* unroll the loop in Figure 3(a) three times, to produce the singly-nested loop in Figure 3(b). After

Benchmark	Dynamic Instructions				Code Optimized Inside Loop	
	BaselineAA or *DSAA*	DeAliaser		Reduction (%)	BaselineAA or *DSAA*	DeAliaser
		Total	Overhead			
433.milc	448	343	2	23	Calculations of addresses for field $real$ in $b{\rightarrow}e[0][0]$, $b{\rightarrow}e[1][0]$, $b{\rightarrow}e[2][0]$ hoisted.	All hoists in BaselineAA. Loads of field $real$ in $b{\rightarrow}e[0][0]$, $b{\rightarrow}e[1][0]$, $b{\rightarrow}e[2][0]$ hoisted.
401.bzip2	950	905	5	5	None.	Load of address for $s{\rightarrow}tt$ hoisted. Redundant load of field $s{\rightarrow}cftab[uc]$ removed.

Table 3. Analysis of the examples shown in Figure 3 and Figure 4.

this, *BaselineAA* and *DSAA* are able to hoist out the address computations for the fields in $b{\rightarrow}e[0][0]$, $b{\rightarrow}e[1][0]$, and $b{\rightarrow}e[2][0]$, but not the loads of the fields themselves. The address computations can be hoisted because the dimensions of b are statically defined, but the loads cannot be hoisted because *BaselineAA* and *DSAA* cannot prove that they do not alias with the updates to the fields in $c{\rightarrow}e[i][0]$. *DeAliaser* places an atomic region around the entire loop and speculatively hoists the loads. We show the code in Figure 3(c). There is a single *storechk.r* to $c{\rightarrow}e[i][0]$. It turns out that b and c are always distinct locations.

```
for (i = 0; i < nblock; i++) {
  uc = (UChar)(s->tt[i] & 0xff);
  s->tt[s->cftab[uc]] |= (i << 8);
  s->cftab[uc]++;
}
```

(a)

```
for (i = 0; i < nblock; i+=50) {
  begin_atomic_opt(Safe_Loop_PC)
  load.r r1, &(s->tt[0]);
  for (ii = i; ii < i+50 && ii < nblock; ii++) {
    uc = (UChar)(r1[ii] & 0xff);
    load.r r2, s->cftab[uc];
    r3 = r1[r2];
    r3 |= (ii << 8);
    storechk.r r3, r1[r2];
    r2++;
    store r2, s->cftab[uc];
  }
  end_atomic_opt
}
```

(b)

Figure 4. Loop in subroutine BZ2_decompress() of *401.bzip2* before (a) and after (b) applying *DeAliaser* with LICM and GVN.

For *401.bzip2* in Figure 4(a), there is not much that *BaselineAA* or *DSAA* can do in terms of LICM or GVN, in spite of the visibly-redundant computations $s{\rightarrow}tt$ and $s{\rightarrow}cftab[uc]$. Neither analysis can prove that the store to $s{\rightarrow}tt[s{\rightarrow}cftab[uc]]$ does not alias with the loads for either of these expressions. However, in Figure 4(b), *DeAliaser* can perform both LICM and GVN after blocking the loop and placing an atomic region around the inner loop.

Through LICM, *DeAliaser* hoists the load of $s{\rightarrow}tt$ out of the loop and stores the value into r1, which gets reused across iterations. Also, through GVN, it stores the value of the first load of $s{\rightarrow}cftab[uc]$ into r2 and reuses it in place of the second redundant load. In the process, the aliasing store to $s{\rightarrow}tt[s{\rightarrow}cftab[uc]]$ is replaced with a *storechk.r* and the moved loads are replaced with *load.r* instructions. The store to $s{\rightarrow}cftab[uc]$, which is in the span of LICM for the load of $s{\rightarrow}tt$, is proven not to alias with that load by the Basic Alias Analysis. Therefore, it is left as is. Also, *clear.r* instructions are introduced for each monitored load to mark when the monitoring ends, and they are eventually eliminated.

Specifically, the *clear.r* for $s{\rightarrow}tt$ is placed at the end of the loop, where the monitoring ends, and is trivially removed. The *clear.r* for $s{\rightarrow}cftab[uc]$ is originally placed at the location where the redundant load used to be, which is right before *r2++*, but is also sunk out of the loop and eliminated. This is possible because even though the monitored load always aliases with $store\ r2, s{\rightarrow}cftab[uc]$, the latter instruction is a regular store and does not check the SR bit. As mentioned, the instruction could remain a regular store and not get converted to a *storechk.r* thanks to the Basic Alias Analysis. This is a good example of cooperation between runtime alias speculation and static alias analysis to achieve the best outcome.

Table 3 presents an analysis of the optimizations performed on these two loops. Columns 2-5 show what effect optimizations have on the average dynamic instruction count of each atomic region. Column 2 refers to the dynamic instruction count when compiled using the *BaselineAA* or *DSAA* configurations (they do not actually contain atomic regions, but we count the corresponding region of code). Column 3 refers to the dynamic instruction count when compiled using the *DeAliaser* configuration. Column 4 shows how many instructions in Column 3 are for the purposes of enabling the *DeAliaser* optimization, such as *begin_atomic_opt*, *end_atomic_opt*, *clear.r*, and instructions needed to block the loop. Column 5 shows the percentage of instruction reduction for *DeAliaser* when compared to *BaselineAA*/*DSAA*. For both loops, applying *DeAliaser* results in a reduction in the number of instructions, namely 23% for *433.milc* and 5% for *401.bzip2*. Columns 6-7 show the actual code movements that occur for the *BaselineAA*/*DSAA* and *DeAliaser* configurations.

5.3 Alias Analysis Results

Figure 6 measures the quality of alias analysis responses generated by each alias analysis configuration, when paired with the LICM pass. The results shown are the percentage of queries that return a *may alias*, *no alias*, or *must alias* response among the total set of queries thrown by the LICM client pass. In the figure, *B*, *D*, and *A* refer to the *BaselineAA*, *DSAA*, and *DeAliaser* configurations, respectively. For *DeAliaser*, *no alias* means that the profiler recommended a speculation be performed, although during the execution, we may suffer occasional aborts.

We are unable to show bars for *400.perlbench*, *403.gcc*, *443.gobmk*, and *483.xalancbmk* for *DSAA* due to reasons mentioned in Section 5.1. For these benchmarks, we use the *BaselineAA* results when calculating the averages. This methodology is used throughout the evaluation.

Alias analyses return a *may alias* response when they can neither prove that a pair of references must alias nor not alias. Hence, the fraction of *may alias* responses is a measure of their imprecision. In the figure, we see that *DeAliaser* leads to a drastic reduction in the fraction of *may alias* responses in all of the benchmarks. All of the *may alias* responses eliminated are translated into *no alias* responses, thereby recommending code motion for LICM. We also see that *DeAliaser* is not able to eliminate all of the *may alias* responses. This is because some pairs of references actually do *may alias* during execution — that is, they reference the same location a noticeable number of times. Finally, recall that DeAliaser is un-

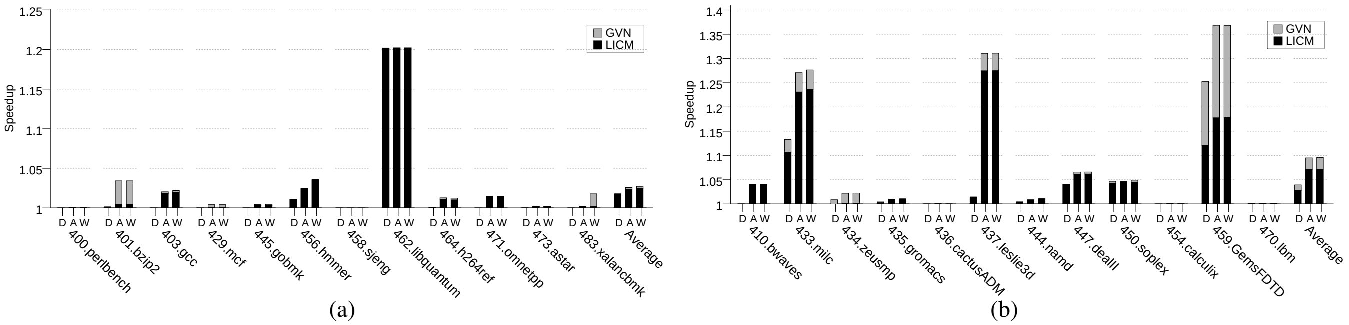


Figure 5. Speedups of *DSAA* (D), *DeAliaser* (A), and word-granularity *DeAliaser* (W) over *BaselineAA* for SPEC INT2006 (a) and SPEC FP2006 (b).

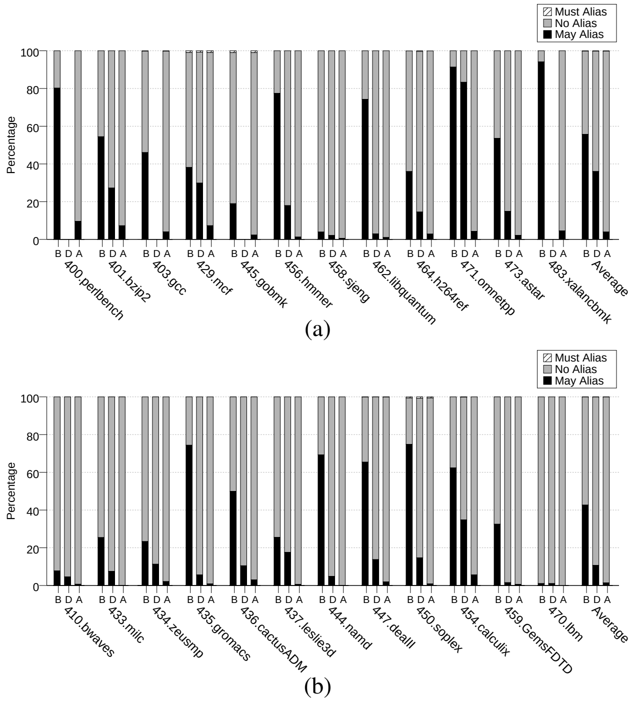


Figure 6. Fraction of alias queries performed by LICM that return a *may alias*, *no alias*, or *must alias* response for SPEC INT2006 (a) and SPEC FP2006 (b). In the figure, B, D, and A stand for *BaselineAA*, *DSAA*, and *DeAliaser*.

able to conclude for sure that two references alias with each other. Hence, some of the remaining *may aliases* in the *DeAliaser* bars may correspond to pairs of references that *must alias*.

The results for GVN are omitted in the interest of space, but follow the same trend.

5.4 Performance

To assess the impact of DeAliaser, we report the execution speedups (Figure 5) and the reduction in dynamic instruction count (Figure 7). Figure 5 shows the benchmark execution speedups measured using our architectural simulator. We show the speedups of *DSAA* (D), *DeAliaser* (A), and word-granularity *DeAliaser* (W) over *BaselineAA*. Word-granularity *DeAliaser* is like *DeAliaser* except that, to detect aliased accesses, caches keep speculative read and write bits *per word*. The speedups are broken down into the contribution of LICM and GVN.

The figure shows that *DeAliaser* enables significant speedups in multiple benchmarks, beyond what can be provided using *DSAA*. On average, *DeAliaser* speeds-up SPEC FP2006 benchmarks by 9% and SPEC INT2006 benchmarks by about 3% over *BaselineAA*. On the other hand, *DSAA* manages an average of 4% for SPEC FP2006 and 2% for SPEC INT2006. In both configurations, most of the speedups come from LICM, although GVN also has a significant impact in benchmarks such as *401.bzip2* and *459.GemsFDTD*.

There is barely any difference between the speedups achieved using word-granularity SR and SW bits and those using line-granularity bits. The granularity of access bits determines the accuracy of alias checks performed by hardware; larger granularity can cause them to return an aliased result due to false sharing. This can potentially stifle optimizations by returning less accurate information. However, the results show that having one SR and SW bit per word is unnecessary, when using 64-byte cache lines.

In general, better alias analysis may not necessarily translate into better performance. This is because even a single alias inside a loop can prevent code optimization. In addition, even if the analysis can prove that none of the references alias with each other, there simply might not be opportunities for optimization in that loop.

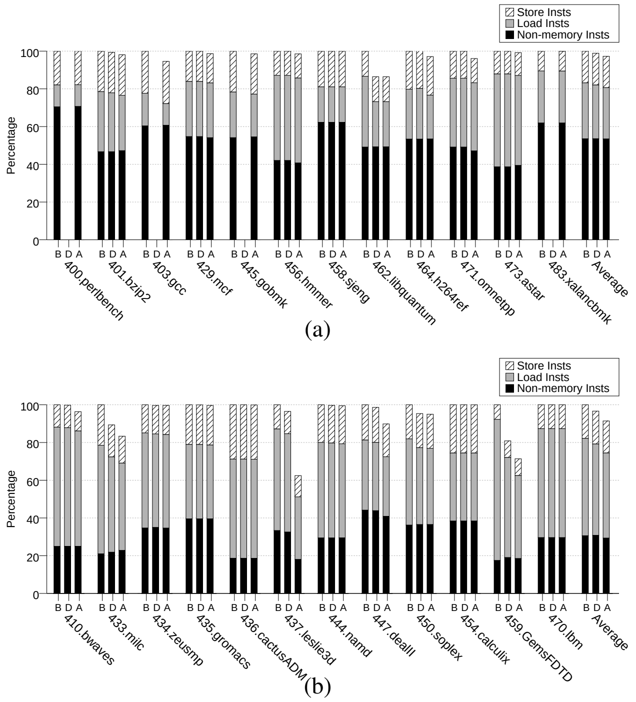
Figure 7 shows the reduction in the dynamic instruction count of the benchmarks categorized by non-memory, load, and store instructions. *B*, *D*, and *A* again refer to the *BaselineAA*, *DSAA*, and *DeAliaser* configurations. We can see that, across the benchmarks, the reduction in the instruction count is loosely correlated with the speedups. On average, *DeAliaser* reduces the instruction count by 9% in the SPEC FP2006 benchmarks and by about 3% in the SPEC INT2006 benchmarks over *BaselineAA*.

The most marked reduction is in the number of load instructions. The hoisting of load instructions that read loop-invariant values from memory is responsible for the majority of the reduction. There is also some reduction in non-memory instructions, especially in *437.leslie3d*. This is due to the movement of the address calculation for matrix element accesses.

Store instructions show some reduction in *437.leslie3d* and *433.milc*. This is due to two reasons. The first one is the sinking of stores to the exit of the loop. The second one is the reduction in register pressure due to having computation moved out of the loop, leading to fewer spills. However, register spills can also be increased with LICM, by increasing the live ranges of the registers that contain the values of hoisted loads. This was the case in *459.GemsFDTD*, but the reduction in load instructions more than made up for it.

5.5 Atomic Region Aborts

In our *DeAliaser* experiments so far, the use of the cost-benefit model of Section 4.1 has resulted in very few atomic region aborts. To study a more aggressive environment, we scrap the model and



(a)

(b)

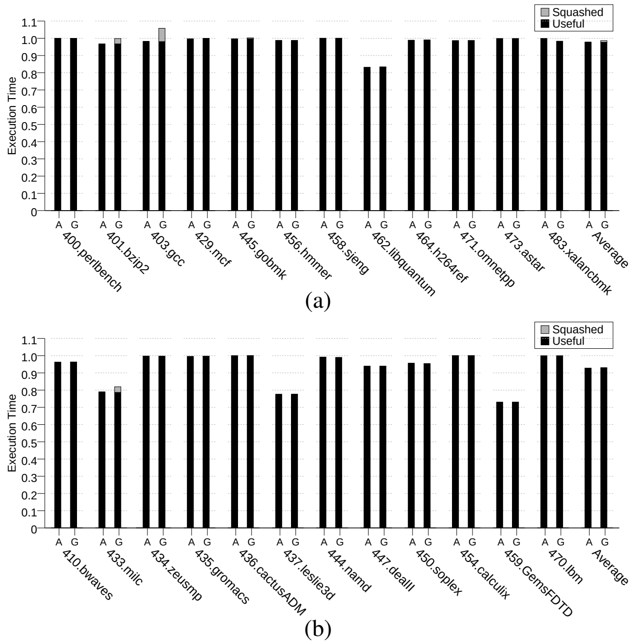
Figure 7. Dynamic instruction count of *BaselineAA* (B), *DSAA* (D), and *DeAliaser* (A) after LICM and GVN for SPEC INT2006 (a) and SPEC FP2006 (b). The bars are broken down by category and normalized to *BaselineAA*.

speculate on all alias pairs that *may alias*, regardless of the cost. We call this configuration *DeAliaser_Aggressive*.

The resulting execution time is shown in Figure 8, alongside that of the original *DeAliaser*, normalized to *BaselineAA*. The *Useful* component of each bar is the time spent making forward progress, while the *Squashed* component is the time spent executing instructions that are aborted due to aliasing. We see that the *Useful* component does not show any change from *DeAliaser* to *DeAliaser_Aggressive*, meaning that barely any new optimizations were enabled. This was expected, given that there was preciously little remaining *may alias* results to speculate upon (Figure 6). However, the few new optimizations that are enabled sometimes do cause abort overhead in *401.bzip2*, *403.gcc*, and *433.milc*. For some benchmarks, it is important for the compiler to have a cost model in order to avoid slowdowns.

5.6 Commit Latency Sensitivity

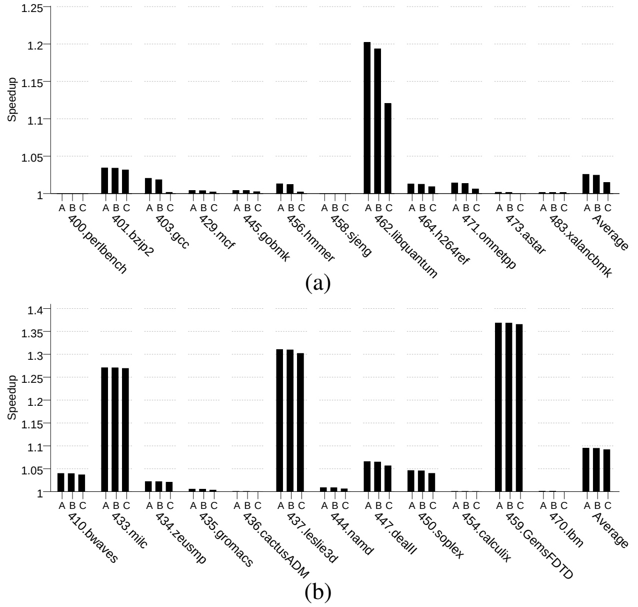
For our performance results in Section 5.4, we assumed that both *begin_atomic_opt* and *end_atomic_opt* take just a single cycle to complete. In different systems, these instructions may take different times and, in particular, the *end_atomic_opt* that commits an atomic region, could take significantly more time. In Figure 9, we do a sensitivity study on how much our speedups depend on commit latency. In the figure, (A), (B), and (C) show the speedups of *DeAliaser* over *BaselineAA* when assuming commit latencies of 1, 10, and 100 cycles, respectively. The figure shows that the speedups for SPEC FP2006 benchmarks barely change even as we increase the commit latency to 100 cycles, but SPEC INT2006 benchmarks are more sensitive. At 100 cycles, the average speedup for SPEC INT2006 is halved. This is because SPEC FP2006 benchmarks have much larger atomic regions, which are better able to amortize commit overhead, as can be seen next.



(a)

(b)

Figure 8. Execution time of *DeAliaser* (A) and *DeAliaser_Aggressive* (G) for SPEC INT2006 (a) and SPEC FP2006 (b). The bars are broken down by category and normalized to *BaselineAA*.



(a)

(b)

Figure 9. Speedups of *DeAliaser* with commit latencies 1 cycle (A), 10 cycles (B), and 100 cycles (C) over *BaselineAA* for SPEC INT2006 (a) and SPEC FP2006 (b).

Benchmark	Coverage(%)	Dynamic Instructions per Atomic Region							L1 Occ. (%)
		BaselineAA	DSAA		DeAliaser		ALAT		
			Insts	Reduc.(%)	Insts	Reduc.(%)	Insts	Reduc.(%)	
401.bzip2	9	933	933	0	734	21	930	0	2
403.gcc	23	439	439	0	337	23	440	0	2
429.mcf	9	1228	1228	0	1053	14	1207	2	0
445.gobmk	12	1229	1229	0	1082	12	1218	1	2
456.hmmer	8	8580	8535	1	6935	19	7787	9	1
462.libquantum	69	1401	1122	20	1124	20	1405	0	2
464.h264ref	8	848	843	1	549	35	811	4	1
471.omnetpp	73	3185	3185	0	3014	5	3028	5	5
473.astar	4	528	528	0	426	19	623	-18	19
483.xalancbmk	1	1171	1171	0	1073	8	1171	0	4
INT Average	18	1702	1674	2	1434	16	1628	4	4
410.bwaves	72	6261	6259	0	5941	5	6101	3	1
433.milc	54	16027	13150	18	11113	31	14620	9	6
434.zeusmp	5	9005	8811	2	8415	7	8879	1	7
435.gromacs	3	2592	2514	3	2257	13	2566	1	4
436.cactusADM	0	3072	3072	0	1228	60	2282	26	1
437.leslie3d	58	5331	5113	4	1853	65	3739	30	4
444.namd	38	24907	24757	1	24574	1	24915	0	14
447.dealII	36	2713	2607	4	1957	28	2362	13	2
450.soplex	25	1377	1121	19	1104	20	1395	-1	5
454.calculix	0	1002	995	1	882	12	995	1	3
459.GemsFDTD	64	28270	20054	29	15571	45	26776	5	21
FP Average	29	8379	7371	12	6241	26	7885	6	6

Table 4. Average dynamic instruction statistics per atomic region for each benchmark.

5.7 Characterization

In this section, we characterize the runtime behavior of the atomic regions instrumented by *DeAliaser*. Dynamic atomic regions are characterized in terms of coverage, instruction count, and hardware speculation and conflict detection resources needed. The data is shown in Table 4. In the table, there are three benchmarks missing (*400.perlbench*, *458.sjeng*, and *470.lbm*) because *DeAliaser* does not instrument any atomic regions.

Column 2 of the table shows the percentage of dynamic instructions that were covered with atomic regions. We can see that the average is 18% for SPEC INT2006 and 29% for SPEC FP2006. Column 3 shows the dynamic instruction count per atomic region for *BaselineAA* after performing LICM and GVN. As mentioned before, *BaselineAA* does not actually contain atomic regions, but we count the number of instructions in the corresponding loop. We can see that we start with sizable atomic regions. The average size is 1702 instructions for SPEC INT2006 and 8379 instructions for SPEC FP2006. This is desirable, to amortize atomic region instrumentation overhead.

Columns 4-5 show information pertaining to the average atomic region in *DSAA*: Column 4 shows the dynamic instruction count, and Column 5 shows the percentage of instruction reduction compared to *BaselineAA*. Columns 6-7 shows the same information for *DeAliaser*. The dynamic instruction count for *DeAliaser* includes overhead for DeAliaser instrumentation and loop blocking. For the vast majority of cases, the only DeAliaser instrumentation required was the *begin_atomic_opt* and *end_atomic_opt* instructions. With *DeAliaser*, we obtain average instruction reductions of 16% for SPEC INT2006 and 26% for SPEC FP2006. For *DeAliaser* to show good speedups, both the instruction reduction percentage and the coverage have to be high. This occurs in *462.libquantum*, *433.milc*, *437.leslie3d*, and *459.GemsFDTD*.

Columns 8-9 show, for comparison, dynamic instruction count and percentage of instruction reduction had we used an ALAT. Although the ALAT can perform all the load hoisting optimizations done by *DeAliaser*, it also requires lots of extra alias check instructions such as ld.c or chk.a to be performed, which cancel out much of the instruction reduction from optimization. This results from the fact that a ld.c or chk.a check instruction must be placed at the original location of every hoisted load. Sometimes, this even leads to a net increase in the total number of instructions. It occurs because the hoisting induces register spills.

Column 10 shows the average fraction of the 32KB speculative L1 cache that is used per atomic region to buffer speculatively written lines and read-monitored lines. In many cases, the L1 occupancy rate is very small compared to the size of the atomic region. For example, *456.hmmer* has 6935 instructions per atomic region, but the occupancy rate is just 1%. In a typical transaction used in transactional memory, most of the lines that need to be buffered are speculatively read lines, since usually loads are more prevalent than stores. In the case of *DeAliaser*, read-monitored lines are only a small fraction of the lines read in the atomic region, which helps reduce the footprint.

6. Discussion

6.1 Limitations due to Imprecision

In Section 3.3, we discussed the potential limitations of DeAliaser due to having just a single set of SR and SW bits to monitor memory locations, and not being able to manipulate SW bits due to rollback requirements. These limitations make the alias checks more imprecise by stifling optimizations.

Adding extra hardware to the upcoming transactional memory systems can help in easing some of these limitations. One addition with significant impact would be to add support for one extra monitored write set, decoupled from rollback requirements, that the compiler can freely manipulate. This can be easily done by adding one extra SW bit per cache line to mark monitored write locations. With this, the precision of alias checks for load sinking, store hoisting, and store sinking that use the $W_{Monitored}$ set would be dramatically improved (refer to Figure 1(b)).

Beyond this, adding more hardware to increase the number of monitored read or write sets can progressively increase the precision of alias checks. Ideally, we would like as many monitored sets as the number of code motions. This can be accomplished with one SR and SW bit-pair in cache lines for each code motion. However, a more efficient implementation may be to use hardware signatures that encode addresses (e.g., [28, 38, 39]).

6.2 Isolation Issues

Repurposing the SR bits for load monitoring means forfeiting the isolation semantics of atomic regions. Let us refer to the atomic regions created using DeAliaser's *begin_atomic_opt* instruction as *opt* atomic regions, and those created using the conventional Transactional Memory (TM) *begin_atomic* instruction as *sync* atomic regions. The issue arises when a remote processor's write conflicts with an address read by a local *opt* atomic region with a non-monitored load. Since the local SR bit is not set, the local *opt* atomic region would still commit, compromising isolation.

The simple solution to this problem is not to use *opt* atomic regions for the synchronization purposes that *sync* atomic regions serve in TM. An atomic region should be used for either optimization or synchronization, but not both.

But what if an *opt* atomic region is nested inside a *sync* atomic region? If nested atomic regions are flattened, as is done in many current systems, manipulation of the SR bits in the inner atomic region would compromise the isolation of the outer *sync* atomic region. To avoid this problem, nested *opt* atomic regions should automatically trigger a jump to the safe version with no DeAliaser instrumentation.

What if a *sync* atomic region is nested inside an *opt* atomic region? To avoid compromising the isolation of the inner *sync* atomic region, all the loads in the inner atomic region should set the SR bit. This would make sure that any read memory conflict in the inner atomic region would trigger a rollback. While this may increase the chances of alias speculation failure in the outer *opt* atomic region, correctness is guaranteed. If the increase in failure rate is too high, it would be detected in the profiling phase and the *opt* atomic region would not be instrumented.

Adding hardware support for maintaining a monitored read set separate from the speculative read set (e.g., by adding one extra bit in each cache line separate from the SR bit) solves this issue.

6.3 Applying to Dynamic Compilation

Although our discussion has focused on the LLVM static compiler, we believe that there is fertile ground for applying alias speculation to dynamic compilers. Dynamic compilers have largely avoided complex alias analyses such as Data Structure Alias Analysis because of the unacceptable compilation overhead. However, a dynamic analysis such as the one in DeAliaser that leverages profiling information is a perfect fit for application to dynamic compilers. Moreover, as discussed in Section 4.1, DeAliaser profiling imposes only a very small overhead.

7. Related Work

7.1 Optimizations using Atomic Regions

Speculative code optimizations in atomic regions have been applied to control speculation over a hot trace of code [15, 29, 30, 32]. When these traces of code are generated to enable on-the-fly optimization of uops translated from machine instructions [7, 8, 15, 29], the atomic regions are also used to support precise exceptions at the machine level. In addition, atomic regions are a good way of hiding code movements, which might violate the machine memory model, from other processors. With this support, a compiler can enforce a strict memory model like sequential consistency on a relaxed-consistency machine, and still attain high performance [4]. Continued interest in atomic regions has also led to work that proposes ways to deal with the limitation of hardware speculation resources, either through short rollbacks [8] or smart monitoring of the resources [7].

7.2 Alias Speculation

There is a significant body of work that attempts to perform alias analysis speculatively, relying on runtime checks to verify them in similar ways as DeAliaser. These checks can either be done purely in software [31], or with the assistance of special hardware such as in Itanium's Advanced Load Address Table (ALAT) [14, 25, 26] and in Transmeta's binary translator [15, 21].

The ALAT is a hardware table that monitors accesses to memory locations. It is used for hoisting load instructions across potentially aliasing stores. A check instruction is placed at the location of the original load that verifies in hardware that no intervening stores aliased with the hoisted load. Since the span of code motion is clearly demarcated by the hoisted load and the check, precise alias checks on multiple code motions are naturally supported. However, the need to perform recovery in software prevents certain optimizations from being performed and the engineering complexity discourages compiler writers from implementing aggressive optimizations. Also, the frequency of check instructions cost both performance and energy.

Perhaps the work that bears the most resemblance to ours is the work on Transmeta's Code Morphing Software (CMS) binary translator [15, 21]. Just like DeAliaser, CMS leverages atomic regions to perform alias speculation. Load hoisting is used to overlap load latencies with computation in their statically-scheduled VLIW hardware. However, it is unclear from the ISA extensions that the company published, how multiple spans of code motion can be supported inside a single atomic region. This support is crucial for aggressive optimization. Moreover, to our knowledge, the exact details of how CMS takes advantage of those extensions have never been published in the literature. In addition, note that neither CMS or the ALAT supports the movement of stores.

In this paper, we discuss many compiler as well as hardware aspects of the problem of doing alias speculation using atomic regions and propose a solution that can move both loads and stores in a general way. We also discuss ways to minimize imprecision in the face of multiple code motions inside an atomic region.

Another related approach has been to have the hardware monitor alias speculation violations and silently patch up the state when it happens, unbeknownst to software and even without having to rollback execution, such as in CRegs [13] and in the Store-Load Address Table (SLAT) [33]. While this is an attractive idea, the complexity of patching up the state is analogous to the job of generating recovery code in the compiler, and doing this in hardware clearly has its limits. Hence, the optimization that they support is limited to speculative register promotion.

8. Conclusions

This paper proposed, for the first time, to expose the alias checking capabilities present in upcoming transactional memory systems for the purpose of alias speculation. This is accomplished with a set of novel instructions and hardware extensions. We call our approach DeAliaser. The paper also presented how these instructions are used in a source-level compiler to enable (i) code motion on both loads and stores, and (ii) multiple code motions in a single atomic region with minimal loss of precision. The latter is crucial for aggressive optimization. DeAliaser's capabilities are attained by simply exposing the existing checkpointing and memory conflict checking facilities of atomic regions to the compiler. No significant hardware complexity was added to existing atomic regions.

We evaluated LICM, GVN, and PRE optimizations enhanced for DeAliaser in the LLVM compiler. Our work resulted in an average application speedup of 9% for SPEC FP2006 benchmarks and 3% for SPEC INT2006 benchmarks, over the application of the baseline LLVM alias analysis. We also showed a detailed characterization of the atomic regions formed by DeAliaser, and examples of real code where the optimizations happened.

Acknowledgements

This work was supported in part by NSF under grants CCF-1012759 and CNS-1116237, and by Intel under the Illinois-Intel Parallelism Center (I2PC).

References

[1] *IBM System Blue Gene Solution: Blue Gene/Q Application Development Manual*. IBM Corp., March 2012.

[2] *Intel Architecture Instruction Set Extensions Programmig Reference*. Intel Corporation, February 2012.

[3] *POWER ISA Transactional Memory*. IBM Corp., December 2012.

[4] W. Ahn, S. Qi, J.-W. Lee, M. Nicolaides, X. Fang, J. Torrellas, D. Wong, and S. Midkiff. BulkCompiler: High-Performance Sequential Consistency through Cooperative Compiler and Hardware Support. In *International Symposium on Microarchitecture*, December 2009.

[5] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting Equality of Variables in Programs. In *Symposium on Principles of Programming Languages*, January 1988.

[6] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, Univ. of Copenhagen, 1994.

[7] E. Borin, Y. Wu, M. Breternitz, and C. Wang. LAR-CC: Large Atomic Regions with Conditional Commits. In *International Symposium on Code Generation and Optimization*, April 2011.

[8] E. Borin, Y. Wu, C. Wang, W. Liu, M. Breternitz, Jr., S. Hu, E. Natanzon, S. Rotem, and R. Rosner. TAO: Two-level Atomicity for Dynamic Binary Optimizations. In *International Symposium on Code Generation and Optimization*, April 2010.

[9] V. T. Chakaravarthy. New Results on the Computability and Complexity of Points–to Analysis. In *International Symposium on Principles of Programming Languages*, January 2003.

[10] J. Chung, L. Yen, S. Diestelhorst, M. Pohlack, M. Hohmuth, D. Christie, and D. Grossman. ASF: AMD64 Extension for Lock-Free Data Structures and Transactional Memory. In *International Symposium on Microarchitecture*, December 2010.

[11] C. Click. Azul's Experiences with Hardware Transactional Memory. In *HP Labs Workshop on Transactional Memory*, January 2009.

[12] J. Da Silva and J. G. Steffan. A Probabilistic Pointer Analysis for Speculative Optimizations. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2006.

[13] P. Dahl and M. O'Keefe. Reducing Memory Traffic with CRegs. In *International Symposium on Microarchitecture*, December 1994.

[14] X. Dai, A. Zhai, W.-C. Hsu, and P.-C. Yew. A General Compiler Framework for Speculative Optimizations Using Data Speculative Code Motion. In *International Symposium on Code Generation and Optimization*, March 2005.

[15] J. Dehnert et al. The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges. In *International Symposium on Code Generation and Optimization*, March 2003.

[16] A. Diwan, K. S. McKinley, and J. E. B. Moss. Type-based Alias Analysis. In *International Conference on Programming Language Design and Implementation*, May 1998.

[17] R. Ghiya, D. Lavery, and D. Sehr. On the Importance of Points-to Analysis and Other Memory Disambiguation Methods for C Programs. In *International Conference on Programming Language Design and Implementation*, June 2001.

[18] M. Hind. Pointer Analysis: Haven't We Solved this Problem Yet? In *Workshop on Program Analysis for Software Tools and Engineering*, June 2001.

[19] S. Horwitz. Precise Flow-insensitive May-alias Analysis is NP-hard. *ACM Transactions on Programming Languages and Systems*, January 1997.

[20] R. Kennedy, S. Chan, S.-M. Liu, R. Lo, P. Tu, and F. Chow. Partial Redundancy Elimination in SSA Form. *ACM Transactions on Programming Languages and Systems*, May 1999.

[21] A. Klaiber. The Technology Behind Crusoe Processors. Technical report, Transmeta, January 2000.

[22] W. Landi. Undecidability of Static Analysis. *ACM Letters on Programming Language System*, December 1992.

[23] C. Lattner and V. Adve. LLVM: a Compilation Framework for Lifelong Program Analysis Transformation. In *International Symposium on Code Generation and Optimization*, March 2004.

[24] C. Lattner, A. Lenharth, and V. Adve. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World. In *International Conference on Programming Language Design and Implementation*, June 2007.

[25] J. Lin, T. Chen, W.-C. Hsu, and P.-C. Yew. Speculative Register Promotion Using Advanced Load Address Table (ALAT). In *International Symposium on Code Generation and Optimization*, March 2003.

[26] J. Lin, T. Chen, W.-C. Hsu, P.-C. Yew, R. D.-C. Ju, T.-F. Ngai, and S. Chan. A Compiler Framework for Speculative Analysis and Optimizations. In *International Conference on Programming Language Design and Implementation*, June 2003.

[27] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *International Conference on Programming Language Design and Implementation*, June 2005.

[28] C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In *International Symposium on Computer Architecture*, June 2007.

[29] N. Neelakantam, D. R. Ditzel, and C. Zilles. A Real System Evaluation of Hardware Atomicity for Software Speculation. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2010.

[30] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles. Hardware Atomicity for Reliable Software Speculation. In *International Symposium on Computer Architecture*, June 2007.

[31] A. Nicolau. Run-Time Disambiguation: Coping with Statically Unpredictable Dependencies. *IEEE Transactions on Computers*, May 1989.

[32] S. Patel and S. Lumetta. rePLay: A Hardware Framework for Dynamic Optimization. *IEEE Transactions on Computers*, June 2001.

[33] M. Postiff, D. Greene, and T. Mudge. The Store-load Address Table and Speculative Register Promotion. In *International Symposium on Microarchitecture*, December 2000.

[34] G. Ramalingam. The Undecidability of Aliasing. *ACM Transactions on Programming Language System*, September 1994.

[35] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC Simulator, January 2005. http://sesc.sourceforge.net.

[36] N. Rinetzky, G. Ramalingam, M. Sagiv, and E. Yahav. On the Complexity of Partially-flow-sensitive Alias Analysis. *ACM Transaction on Programming Language System*, May 2008.

[37] B. Steensgaard. Points-to Analysis in Almost Linear Time. In *International Symposium on Principles of Programming Languages*, January 1996.

[38] J. Torrellas, L. Ceze, J. Tuck, C. Cascaval, P. Montesinos, W. Ahn, and M. Prvulovic. The Bulk Multicore Architecture for Improved Programmability. *Communications of the ACM*, December 2009.

[39] J. Tuck, W. Ahn, L. Ceze, and J. Torrellas. SoftSig: Software-Exposed Hardware Signatures for Code Analysis and Optimization. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, January 2008.

[40] T. Vandrunen and A. L. Hosking. Value-based Partial Redundancy Elimination. In *Compiler Construction*, March 2004.

[41] K. R. Wadleigh and I. L. Crawford. *Software Optimization for High Performance Computing: Creating Faster Applications*. Prentice Hall, 2000.

[42] R. P. Wilson and M. S. Lam. Efficient Context-sensitive Pointer Analysis for C programs. In *International Conference on Programming Language Design and Implementation*, June 1995.

[43] Y. Wu and J. R. Larus. Static Branch Frequency and Program Profile Analysis. In *International Symposium on Microarchitecture*, November 1994.