Enabling Efficient Alias Speculation

Soumyadeep Ghosh[†]
Princeton University
deep@princeton.edu

Yongjun Park[†]
Hongik University
yongjun.park@hongik.ac.kr

Arun Raman[†]
Qualcomm Research
arraman@qti.qualcomm.com

Abstract

Microprocessors designed using HW/SW codesign principles, such as Transmeta™ Efficeon™ and the soon-to-ship NVIDIA 64-bit Tegra® K1, use dynamic binary optimization to extract instruction-level parallelism. Many code optimizations are made significantly more effective through the use of alias speculation. The state-of-the-art alias speculation system, SMARQ, provides 40% speedup on average over a system with no alias speculation. This performance, however, comes at the cost of introducing new *alias registers* and increased power consumption due to new checks for validating speculation. Consequently, improving the efficiency of alias speculation by reducing alias register requirements and rationalizing speculation validation checks is critical for the viability of SMARQ. This paper presents **alias coalescing**, a novel technique to significantly improve the efficiency of SMARQ through a synergistic combination of compiler and microarchitectural techniques. By using a more compact encoding for memory access ranges for memory instructions, alias coalescing simultaneously reduces the alias register pressure in SMARQ by a geomean of 26.09% and 39.96%, and the dynamic alias checks by 20.73% and 33.87%, across the entire SPEC CINT2006 and SPEC CFP2006 suites respectively.

Categories and Subject Descriptors D.3.4 [*Processors*]: Code generation, Incremental compilers

Keywords binary translator, alias speculation, code generation, hardware/software co-design

1. Introduction

The industry is actively exploring microprocessor design using HW/SW co-design through dynamic binary translation to get high performance at low power. Commercial examples of such microprocessors are Transmeta™'s x86 Efficeon™ and NVIDIA 64-bit ARM Tegra® K1. In these designs, hot code is identified at runtime and optimized by a dynamic binary translator, which deploys a suite of low-overhead compiler optimizations to extract instruction-level parallelism (ILP) in the code.

Memory disambiguation to discover *unaliased memory operations* and their dependent instructions is critical to extract maximum ILP. Memory disambiguation is usually done using alias analysis by a compiler. However, in a dynamic binary translator, the cost of a full suite of alias analysis may be prohibitively large. And in the general case, alias analysis is undecidable [12]; thus, even the best analyses conservatively report *may-alias* relationships between memory instructions. To overcome these limitations, optimization systems often use **alias speculation** to improve program performance.

Alias speculation assumes that some may-alias memory instructions never alias with each other at runtime and speculatively schedules these instructions out of order. At runtime, the speculated instructions write one or more "alias registers" with their memory access ranges; these registers are subsequently checked to ensure that speculation is valid. When speculation fails, misspeculation recovery mechanisms ensure correctness of execution. For efficiency, alias speculation systems often have hardware support to check speculation and recover swiftly from misspeculation.

Past alias speculation proposals [3, 4, 6, 7, 11, 13, 14, 18, 21] either (i) encode memory access ranges (base address and size of access) of instructions using a signature (e.g. a Bloom filter [15, 20]), or (ii) encode the memory access range of each instruction individually in an "alias register". While signature-based schemes compute signatures agnostic to input code, these schemes often result in too many false positives (leading to misspeculations) to be practically viable.

On the other hand, alias register based techniques incur few (or even no) false positives and achieve significant performance improvements. For instance, the state-of-the-art alias speculation system, SMARQ [21] provides 40% speedup on average over a system with no alias speculation. However, alias speculation systems like SMARQ use at least one alias register to encode memory access ranges for each memory instruction. This necessitates the introduction of a large number of alias registers to achieve any significant performance gains. Additionally, these systems also require each alias register to be individually checked to validate speculation. This results in a large number of dynamic alias checks, which in turn, increases the power requirements for these systems.

Thus, it is critical to reduce the alias hardware requirements of an alias speculation system like SMARQ while retaining its performance benefits. This work presents a novel HW/SW co-designed alias speculation scheme called **alias coalescing** that achieves this goal. Alias coalescing is premised on a key observation—*memory instructions in programs often access spatially proximate memory addresses.* This spatial locality often manifests in programs as memory instructions with the same base address but different immediate displacements. Often, these instructions refer to different fields of the same data structure.

Alias coalescing combines/coalesces the memory access ranges of such instructions into one "alias register", thus transforming concrete memory reference disambiguation into memory object disam-

[†] This work was carried out while the authors were working at Intel Labs in Santa Clara, CA.

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. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

LCTES '15, June 18–19, 2015, Portland, OR, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3257-6/...$15.00.
http://dx.doi.org/10.1145/2670529.2754964

biguation. This reduces the alias hardware requirement from one alias register per speculated memory instruction to one alias register per coalesced set of memory instructions. The number of checks required to validate speculation also reduces from one check per memory instruction to one check per coalesced set. Alias coalescing uses new algorithms to analyze code at run-time and identify *coalescent* memory operations whose memory access ranges can be effectively combined into a few alias registers and checked in a HW alias checking structure. Thus, coalescing hits the sweet spot between signature-based and one-register-per-speculated-access approaches by getting the alias checking efficiency benefits of the former and the finer checking granularity (and lower false positive rates) of the latter.

We evaluate alias coalescing on top of the state-of-the-art alias speculation system, SMARQ [21]. Alias coalescing reduces the alias register pressure for SMARQ (using 28 alias registers) by a geomean of 26.09% and 39.96% respectively for SPEC CINT2006 and SPEC CFP2006 benchmark suites. Further, it reduces the number of dynamic alias checks required by a geomean of 20.73% for SPEC CINT2006 and 33.87% for SPEC CFP2006 workloads.

In summary, the contributions of this paper are:

- Alias coalescing, a technique to more efficiently encode memory access ranges and enable efficient alias speculation;
- Methods to identify memory operations suitable for coalescing, interfaces to alias checking hardware for coalescing, methods to allocate alias registers for coalesced memory operations, and hardware design to perform coalesced alias checking; and
- A detailed evaluation of the effectiveness of alias coalescing in reducing alias hardware requirements for rotating register file based alias speculation.

2. Motivation

To illustrate the need for alias coalescing, consider how alias speculation in SMARQ [21] reorders memory instructions for the example in Figure 1(a). The `while` loop calculates the sum of all elements in a linked list, and stores it in memory. Figure 1(b) shows the sequence of memory instructions in this loop. Figure 1(c) shows a speculative reordering of the memory instructions designed to hide load latency and to allow the loads' dependent instructions to be scheduled earlier, thereby improving performance. This speculative reordering is valid if and only if the memory access ranges of the loads (LD_2 and LD_3) do not overlap with the memory access range of the store ST_3. To verify this assumption, SMARQ uses alias *protection* and *checking*.

```
val = node->value;
while (node) {
  *sum+= val
  node = node->next;
  val = node->value;
}
```

(a) High-level Program

```
ST1: st  r3, [r4]
LD2: r2 = ld  0x8[r2]
LD3: r1 = ld  0x0[r2]
```

(b) Original Sequence of Memory Instructions

```
LD2: r2 = ld  0x8[r2] // protect
LD3: r1 = ld  0x0[r2] // protect
ST1: st  r3, [r4]     // check
```

(c) Reordered Memory Instructions

```
[r4] represents sum,
0x8[r2] represents
node->next,
0x0[r2] represents
node->value
```

Figure 1. Hardware alias detection

The speculatively hoisted instructions first record their memory access ranges in their individual alias registers. This is called the *protect* step, as these speculated instructions are being protected from memory operations (typically stores) that occur earlier in original program order. When the instructions occurring earlier in the original program order are executed, they *check* their own memory access ranges against those of the speculatively hoisted instructions to verify that speculation is correct. An overlap detected by the check step raises an *alias fault* indicating misspeculation. In response to an alias fault, the system recovers by rolling back execution to an earlier, non-speculative state. Execution is then resumed without reordering the instructions which caused the alias fault to be raised. Figure 1(c) shows the protect and check operations associated with the different memory instructions in the example.

Note that each speculated memory instruction in SMARQ requires one alias register. Thus, the alias register pressure is directly proportional to the number of speculated instructions. Furthermore, during the check operation, each speculated instruction introduces one dynamic alias check. This directly affects the latency to detect the presence of an alias fault (more checks imply higher latency). Other alias speculation schemes such as Advanced Load Address Table (ALAT) in Itanium [3, 4, 13, 14] and static alias registers in the Transmeta™ processors [6, 11] exhibit the same properties. Thus, there is a need to introduce a more efficient and compact encoding of memory access ranges to improve the *efficiency* of alias speculation—by reducing the alias register pressure and by reducing the number of dynamic alias checks. The proposed encoding must strike a good balance between hardware requirements for alias protection and checking, and precision and efficiency to identify individual aliasing memory operations. Alias coalescing aims to present such an efficient encoding.

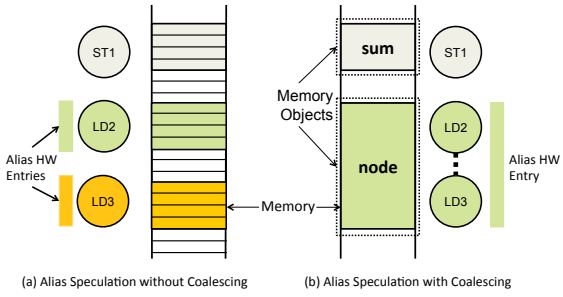

Figure 2. Transformation of concrete memory reference disambiguation to memory object disambiguation due to alias coalescing for the code in Figure 1. Each color represents a separate alias hardware entry.

3. Alias Coalescing

To understand the idea behind alias coalescing, consider again, the code example in Figure 1(a) and the corresponding sequence of memory instructions shown in Figure 1(b). LD_2 and LD_3 access the fields $next$ and $value$ of the same memory object, $node$. In existing alias speculation proposals, both these loads would occupy their own entries in the alias hardware. However, one could coalesce the alias hardware entries for these loads into a single entry. As shown in Figure 2, *coalescing the alias hardware entries for these loads increases the granularity of memory disambiguation from concrete memory references to memory objects*.

While object-based representation of alias hardware entries encodes memory access ranges more compactly, it requires knowing object layouts in memory, which is often difficult to determine at the binary level. Thus, we need a simpler way to identify speculated instructions whose alias hardware entries could be coalesced. One option is to use memory profiling [2, 10, 19, 22] to determine spatially proximate memory locations, which often lend themselves to better encoding schemes. Further, one may use a Bloom filter [15, 20] to summarize these accesses. However, profiling in-

troduces its own set of overheads and requires representative input sets. Similarly, using a Bloom filter may introduce less efficient alias checking and insufficient precision to identify individual aliasing operations.

To address the above problems, we make the following key observation: *programs often contain a number of memory instructions accessing spatially proximate addresses. This often manifests in code as memory instructions with the same base address but different immediate displacements*. Often, these memory instructions access different fields of the same memory object. Alias coalescing *coalesces* the alias hardware entries for each of these speculated memory instructions (with same base address but different immediate displacements) into a single entry. Thus, the alias hardware requirement falls from one entry per memory instruction to one entry per coalesced set. Similarly, the number of dynamic alias checks required also reduces from one check per memory instruction to one check per coalesced set. These reductions enable alias coalescing to improve the efficiency of alias speculation.

3.1 Identifying Memory Operations for Coalescing

The first step for alias coalescing is to identify the memory operations whose memory access ranges can be coalesced. This is done by formally capturing the observation that *spatially proximate memory accesses are often present in programs as memory instructions with the same base address but differing in their displacements*. Two memory operations M_i and M_j are considered candidates for coalescing, if their:

- base addresses, indexes, and scales are the same, and
- immediate displacements differ by not more than $\epsilon > 0$.

After memory operations are identified as candidates for coalescing, the memory access ranges for the coalesced set is obtained as follows: Assuming that each memory operation M_i has the address range $[b_i, e_i]$, the memory access range of the coalesced set is the less precise $[B, E]$, where

$$B = min(b_i), \text{and } E = max(e_i); \forall i$$

3.2 Static vs Dynamic Coalescing

Once the coalesced sets of memory instructions are determined, the next step is to decide when to coalesce the memory access ranges of these instructions. Coalescing of memory access ranges may be performed either by a representative memory operation in the coalesced set, called *static coalescing*, or each memory operation individually, called *dynamic coalescing*.

Static Coalescing: Consider the static coalescing example shown in Figure 3(a). Subscripts indicate the order of memory operations in the original program. Suppose ld_4 is selected as the representative of the coalesced set $\{ld_2, ld_4, ld_5\}$. Also, let $imm1$ and $imm3$ be the minimum and maximum displacements respectively for the loads in the coalesced set. For the reordered sequence shown in Figure 3(a), the first memory operation from the coalesced set is marked for alias protection. The annotation `COALESCE imm1 imm3` indicates the range of offsets to be added to the base register of the associated memory operation. The range of offsets comprises the minimum immediate displacement (`-imm1`) and the maximum immediate displacement (`imm3`) in the candidate coalesced set. It must be noted that static coalescing would require further changes to the ISA (over the base alias speculation scheme) because we would need to encode the range of offsets within the instruction itself.

Dynamic Coalescing: In contrast to static coalescing which summarizes the memory address range of a coalesced set using a representative memory operation, dynamic coalescing coalesces an alias register as and when each member of the coalesced set executes. Figure 3(b) shows an example of alias speculation with dynamic coalescing. In this example, ld_4 initially protects its alias hardware

```
ld4 imm1[r2]   // protect COALESCE imm1 imm3
ld2 imm2[r2]
st1 [r1]       // check
ld5 imm3[r2]
st3 [r3]       // check
```

(a) Static Coalescing

```
ld4 imm1[r2]   // protect
ld2 imm2[r2]   // protect
st1 [r1]       // check
ld5 imm3[r2]   // protect
st3 [r3]       // check
```

(b) Dynamic Coalescing

Figure 3. Examples of Static and Dynamic Alias Coalescing

entry by writing its own memory access range. The subsequent loads ld_2 and ld_5 update the same entry when they are executed, each extending the entry to include its own memory access range. Thus, when st_1 executes, it checks the memory access range of the coalesced set not including ld_5, as the latter updates the alias hardware entry dynamically after st_1 has finished its checks. Since dynamic coalescing extends memory access ranges incrementally, fewer spurious alias faults (false positives) may occur, compared to static coalescing. Another important distinction from static coalescing is that dynamic coalescing does not require any further changes to the ISA. Due to these advantages of dynamic coalescing over static coalescing, this work presents an evaluation of a dynamic coalescing system.

3.3 Interfering Instructions

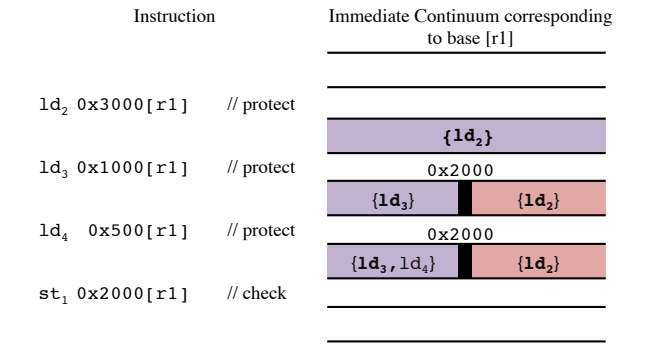

Figure 4. Coalescing with interfering instructions

The memory access range of a coalesced instruction is the same as the memory access range for the entire coalesced set. However, this loss in precision may result in spurious alias faults. Consider the instruction sequence in Figure 4. The subscript of a memory operation indicates its position in the original, unoptimized, program order. Loads are hoisted speculatively and must be checked by stores past which they have been reordered. In the example, assuming single-byte memory accesses and $r1 = 0$, the memory access ranges of ld_2 and ld_3 are `[0x3000, 0x3000]` and `[0x1000, 0x1000]` respectively. If the two loads were to be coalesced, the memory access range of the coalesced set is `[0x1000, 0x3000]`.

When st_1 is subsequently executed, the memory access range of the store `[0x2000, 0x2000]` overlaps with the alias register written by the speculated loads, i.e. `[0x1000, 0x3000]`, raising a spurious alias fault. The fault would not have arisen had ld_3 not been coalesced with ld_2 and they had each written a separate alias register. We call st_1 an *interfering instruction* since it interferes with alias coalescing.

To avoid such spurious faults, the system must not coalesce loads that may be analyzed to have interfering instructions. We employ hole-based alias register allocation to eliminate such spurious faults. Alias coalescing may be viewed as operations on an "immediate continuum" (see Figure 4) induced by instructions sharing the same base, index, and scale but different immediates. At the

beginning, the immediate continuum corresponding to base $r1$ is empty/unallocated. After ld_2, the continuum consists of just ld_2 and an alias register is assigned to ld_2. When ld_3 is considered for coalescing, it adds any interfering instructions that have been determined via analysis by the binary translator (described later in Section 4.2) as holes in the immediate continuum. In this case, st_1 is an interfering instruction and introduces a hole (shown in black in Figure 4) effectively splitting the continuum, and thereby preventing ld_3 from being coalesced with ld_2 since they fall on either side of the hole. This causes ld_2 to be assigned a different alias register. Subsequently, ld_4 is coalesced with ld_3 since they fall on the same side of the hole and ld_4 writes the same alias register as ld_3. After st_1, the hole is removed; since st_1 is the youngest instruction past which the loads were hoisted, the two allocated alias registers may be freed, and the continuum returns to its initial empty state.

4. Implementation

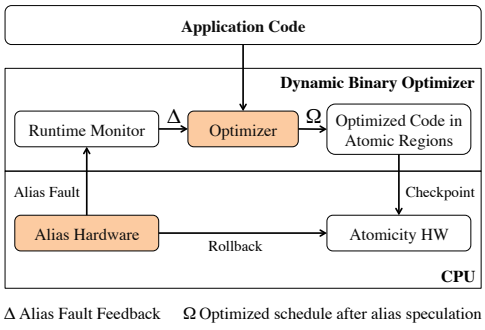

Figure 5. Dynamic optimization system with HW/SW co-designed alias speculation with coalescing

Our target architecture is a HW/SW co-designed research VLIW processor, where the alias hardware consists of a rotating alias register file. The base alias speculation system is SMARQ [21] and we implemented dynamic coalescing on top of SMARQ. During execution, the application code is first optimized by a dynamic binary optimizer. Since it is expensive and difficult to perform traditional alias analysis during execution [5], the dynamic optimizer performs simple alias analysis and relies on alias speculation to improve the effectiveness of speculative optimizations.

The optimized code is organized into atomic regions for speculative execution [17]. These atomic regions are single-entry, multi-exit superblocks with arbitrary control flow inside including conditionals, loops, and function calls. During execution, the atomicity hardware in the CPU creates a checkpoint at the entry point of each speculative region. These checkpoints are used to rollback execution in case of an alias fault. The runtime monitor in the dynamic optimizer catches any alias faults and subsequently invokes the optimizer to re-optimize the speculative region. The reoptimization step is more conservative and is based on the assumption that the two memory instructions that triggered the alias fault always alias with each other. All the memory consistency violations, hardware interrupts and exceptions are also caught by the runtime monitor to trigger rollbacks of speculative regions [8].

4.1 Alias Checking Hardware

The alias checking hardware consists of a bank of alias registers (ARs), organized as a rotating alias register file (Figure 6(a)). It has a pointer to the oldest memory operation in a separate `AHPTR` register. Figure 6(b) shows the format of a single alias register, which consists of five fields: (1) a *valid* bit to indicate if the entry in the alias register is in use; (2) a *load/store* bit to indicate if the memory instruction using the alias register is a load or a store instruction; (3) the *physical page number (PPN)* for the memory locations accessed; (4) *begin* offset for storing the start of the memory access range; and (5) *end* offset for storing the upper end of the memory access range. Both *begin* and *end* are encoded relative to the physical page number. For a 48-bit address space, this scheme reserves 36 bits for the *PPN* and 13 bits each for the *begin* and *end* entries. Thus, the size of each alias register is 64 bits.

To handle cases where instructions in a coalesced set may span non-consecutive pages, we introduce a small set of auxiliary registers called *Page Crossing Structures (PCS)*. Each PCS register (Figure 6(c)) has a six-bit entry for alias register number (AR#) in addition to the fields of an alias register in the rotating alias register file. Whenever page crossing cannot be encoded using the 13 bits of the *begin* and *end* fields of an alias register A in the main rotating register file, the hardware allocates a PCS register and sets the AR# field to be A. The other fields in the PCS are set using the memory access ranges for the new physical page that has been accessed. During a check operation, the hardware now checks the registers in the rotating alias register file, as well as the corresponding PCS registers (indexed using the AR# values). When a PCS register cannot be allocated because all are in use, the hardware raises an exception in response to which the code is retranslated without alias coalescing. Table 1 shows the various actions performed by the alias hardware in response to a protect, check, or rotate operation.

4.2 Changes to the Dynamic Binary Optimizer

For correct alias speculation, the dynamic binary optimizer must implement the desired checking relationship among program loads and stores by assigning them *protect* (P) and *check* (C) bits along with the appropriate *order*. For efficient alias speculation, the assignment must minimize the number of checks and the number of spurious alias faults. Finally, the algorithms employed must be fast. We employ a fast, topological sorting based register allocation algorithm, built on top of the SMARQ alias register allocation algorithm. Unlike general-purpose register allocation, there is no hardware support for alias register spilling. Consequently, alias register allocation is integrated with the instruction scheduler. Alias speculation is throttled appropriately to prevent memory operation reordering (and therefore use of alias registers) when there is a paucity of alias registers.

4.2.1 Preliminaries

Alias register allocation is done when instructions are considered for scheduling, by building a *checking graph* that precisely specifies the alias checks to be performed by the hardware. Figure 7 specifies the various types of edges added to instruction nodes. A load A may be reordered past multiple stores A_i, such that MAY_ALIAS(A, A_i) and therefore CHECK(A_i, A) are true. All such stores must check (C) the alias register written for protection (P) by the load—for such stores, C(A_i) is $true$, and for the load, P(A) is $true$. The liverange of the alias register begins at the load and ends at the *last store in the reordered program order*. The order associated with the load A, and consequently the alias register allocated to it, is determined by the last store A_i that checks it. The alias register associated with the load A may be deallocated and used for a different load after the last store A_i has checked A.

In addition, anti-checking edges (as defined in Figure 7) must be added to the checking graph. Section 4.2 in [21] details the need for anti-checking edges. ANTI-CHECK(A_j, A_i) prevents A_i from checking A_j by ensuring that A_j is assigned a higher order than A_i. This avoids alias faults that are unnecessary for correctness of optimization; in the absence of anti-checking edges, spurious alias faults may be raised due to multiple choices in the topological sorting of the checking graph. Note that from the perspective of the

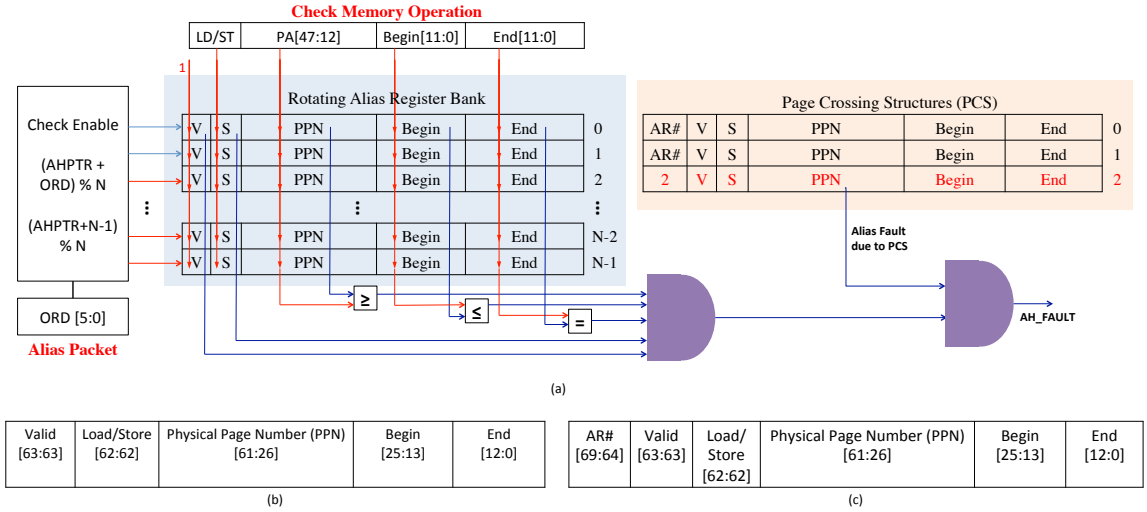

Figure 6. Alias Checking Hardware. (a) Rotating alias registers, PCS, and the alias fault checking mechanism (b) A rotating alias register (c) A page crossing structure (PCS) register

Operation	Intent	Actions performed by Alias Hardware
protect `ord`	Update alias register with memory access range	x = (AHPTR + `ord`) % N if V(AR_x) = 0: V(AR_x) = 1, PPN(AR_x) = p, begin(AR_x) = b, end(AR_x) = e if V(AR_x) = 1 and PPN(AR_x) = p: begin(AR_x) = min(begin(AR_x), b), end(AR_x) = max(end(AR_x), e) if V(AR_x) = 1 and PPN(AR_x) != p: Allocate PCS_i; V(PCS_i) = 1, PPN(PCS_i) = p, begin(PCS_i) = b, end(PCS_i) = e
check `ord`	Compare valid alias registers with memory access range	for all alias registers AR_i with i ∈ [(AHPTR+`ord`) % N, (AHPTR+N-1) % N]: if V(AR_i) = 0: no alias fault if V(AR_i) = 1: if PPN(AR_i) != p, no alias fault if PPN(AR_i) = p: if b < end(AR_i) and e > begin(AR_i), raise alias fault else no alias fault repeat above checks for all PCS_j with AR#(PCS_j) = i where i ∈ [(AHPTR+`ord`) % N, (AHPTR+N-1) % N]
rotate `n`	Invalidate `n` alias registers starting at AHPTR	V(AR_i) = 0, for i ∈ [AHPTR, AHPTR + `n` - 1] V(PCS_j) = 0, for all PCS_j with AR#(PCS_j) = i, where i ∈ [AHPTR, AHPTR + `n` - 1] AHPTR = AHPTR + `n`

Table 1. Actions performed by the alias hardware on protect, check and rotate operations. AHPTR points to the oldest live alias register. N is the total number of alias registers. The memory operation in question accesses addresses [b,e] within the physical page p.

MAY_ALIAS (A_j, A_i) if:
- A_i precedes A_j in program order
- A_i and A_j may access the same memory location
- A_i or A_j is a store

CHECK (A_i, A_j) if:
- MAY_ALIAS (A_j, A_i)
- A_j precedes A_i after scheduling

ANTI-CHECK (A_j, A_i) if:
- MAY_ALIAS (A_j, A_i)
- A_j precedes A_i after scheduling
- not CHECK (A_j, A_i)
- P(A_i), and
- C(A_j)

COALESCENT (A_i, A_j) if:
- scales, indexes and base addresses of A_i and A_j are the same [SIB(A_i) = SIB(A_j)], and
- Immediate displacements of A_i and A_j differ

INTERFERE (A_i, A_j) if:
- COALESCENT (A_i, A_j)
- A_i precedes A_j in original program order
- A_j precedes A_i in optimized program order
- A_i is a store and A_j is a load

Figure 7. Definitions of types of edges used to construct the checking graph for alias register allocation

checking graph and the alias register allocation algorithm, an anti-checking edge ANTI-CHECK(A_j, A_i) is equivalent to a checking edge CHECK(A_j, A_i).

Interference edges (also defined in Figure 7) are used to perform hole-based alias register allocation to eliminate the effect of interfering stores as discussed in Section 3.3. Each memory operation M is associated with a tuple <scale, index, base> or SIB(M) containing value numbers for each register, obtained via a global value numbering pass [16]. As shown in Figure 7, COALESCENT(A_i, A_j) if SIB(A_i) equals SIB(A_j). The algorithm maintains a continuum for each SIB, CONT(SIB), which is initialized to [-∞, ∞]. The continuum consists of fills and holes, with holes introduced by interfering stores, and fills having a representative load that may be coalesced with by other loads. REP(CONT(SIB(M)), RANGE(M)) retrieves the representative load, if any, of the fill in the continuum in which memory operation M falls, where RANGE(M)=[imm(M), imm(M)+size(M)].

Note that INTERFERE(A_i, A_j) needs to exist only if C(A_i) ∧ P(A_j). However, the decision to coalesce A_j with an earlier load A_k may potentially have to be made before A_i acquires checking. Consequently, the algorithm conservatively adds interference edges a priori before scheduling/alias register allocation. While this may result in unnecessary splitting of a continuum and lost coalescing opportunities, in practice, most such stores A_i acquired checking.

4.2.2 Alias Register Allocation Algorithm

Algorithms 1-5 explain the procedure for allocating alias registers to speculated memory operations. The input to the alias coalescing and allocation algorithm is the list of instructions in the superblock, since the algorithm is embedded inside the instruction scheduling compiler pass. The instructions are annotated with metadata relevant for building the may-alias and interference graphs in the alias coalescing algorithms; the metadata is accumulated over the dynamic compilation steps prior to scheduling.

Algorithm 1 schedule()

```
1:  MAY_ALIAS = compute_may_alias()
2:  INTERFERE = compute_interference()
3:  num_alloc_regs = 0, num_delay_regs = 0
4:  delayed = {}, AHPTR = 0
5:  repeat
6:     A_i = Pick unscheduled instruction
7:     A_ic = try_coalescing(A_i) // refer Algorithm 2
8:     for A_j ∈ MAY_ALIAS(*, A_i) do
9:        if ¬ SCHEDULED(A_j) then
10:          C(A_j) = true
11:          if ¬ P(A_i) then
12:             P(A_i) = true
13:             delay_alias_reg (A_i) // refer Algorithm 3
14:          end if
15:          CHECK(A_j, A_ic) = true
16:       else
17:          if P(A_j) ∧ C(A_i) ∧ AR(A_j) is NONE ∧
             ¬CHECK(A_j, A_ic) ∧ ¬COALESCED(A_j) then
18:             CHECK(A_j, A_ic) = true // anti-checking edge
19:          end if
20:       end if
21:    end for
22:    if (C(A_i) ∨ P(A_i)) then
23:       alloc_alias_reg (A_i) // refer Algorithm 4
24:    end if
25:    SCHEDULED(A_i) = true
26:    Remove A_i from holes in CONT(SIB(A_i))
27: until all instructions are scheduled
28: allocate_coalesced_alias_reg() // refer Algorithm 5
```

Initialization (Algorithm 1, lines 1-3): Compute the may-alias relationship among memory operations. Initialize the total number of loads that have been assigned alias protection ($P(A_i)$ is true) and assigned order (num_alloc_regs) or not assigned order as yet (num_delay_regs) at any point in the algorithm.

Scheduling loop (Algorithm 1, lines 4-27): On each iteration of the loop, the algorithm picks an instruction to schedule based on some scheduling algorithm, e.g. list scheduling. If the instruction is a memory operation, then it attempts to assign an alias register if alias protection or checking is needed. It reduces the number of alias registers simultaneously in use by coalescing when possible.

Alias Register Coalescing (Algorithm 2): All stores A_i which interfere with the load attempting to be coalesced A, are processed to add holes in the continuum CONT(SIB(A)). When a hole is added, it splits the range represented by a load in two, with the load originally representing the whole range now representing only a subrange. Holes and fills are merged when possible—this detail is elided from the algorithm for simplicity. After holes relevant to load A have been added, the continuum is checked to see if there is a valid representative with which A may be coalesced. If so, then the algorithm coalesces A with the representative A_j, and returns A_j; else, the algorithm returns A. With coalescing, Algorithm 1 constructs the checking graph with A_j acting as a proxy for A; all checking and anti-checking edges which would have been incident on A are instead incident on A_j.

Incremental Construction of Checking Graph (Algorithm 1, lines 8-21): Checking edges go backwards in the schedule while anti-checking edges go forward. Therefore, unscheduled instructions are inspected for checking edges while instructions already scheduled are inspected for anti-checking edges. Note that an anti-checking edge is not added if load A_j has been coalesced; this is because A_j protects *itself and later loads* from intervening stores which may alias, and the precedence relationship (A_j precedes A_i after scheduling) no longer holds.

Alias Register Delay (Algorithm 3): When a memory operation

Algorithm 2 try_coalescing (A), called by schedule()

```
1:  for all A_i ∈ INTERFERE(*,A) do
2:     rep = REP(CONT(SIB(A)), RANGE(A))
3:     if rep is not NONE then
4:        if [l, r] = range of CONT(SIB(A)) represented by rep
          then
5:           if RANGE(REP(rep)) ¡ RANGE(REP(A)) then
6:              REP(l, imm(A)) = rep
7:              REP(imm(A)+size(A), r) = REP(r, ...)
8:           else
9:              REP(imm(A)+size(A), r) = rep
10:             REP(l, imm(A)) = REP(..., l)
11:          end if
12:       end if
13:    end if
14:    CONT(SIB(A)) = CONT(SIB(A)) \ RANGE(A)
15: end for
16: A_j = REP(CONT(SIB(A)), RANGE(A))
17: if A_j is not NONE then
18:    COALESCED_WITH(A) = A_j
19:    COALESCED(A_j) = true
20:    return A_j
21: end if
22: return A
```

A_i is marked as requiring alias protection, $P(A_i)$ is true, it is added to the $delayed$ list. An alias register cannot be assigned to A_i since its order will be known only after later checking stores have been allocated. If there is no representative of the range in the continuum accessed by A_i, it is marked as the potential representative of that range.

Alias Register Allocation (Algorithm 4): The algorithm assigns alias registers to memory operations by topologically sorting the checking graph—when a memory operation has no incoming checking edges, it is allocated an alias register. The alias register assigned (line 6) is an offset with respect to the oldest memory operation in the machine which has order `AHPTR`. Note that if a memory operation W has been coalesced, it is assigned an alias register in a post-pass because it should be assigned the same alias register as the operation with which it was coalesced. When a memory operation is assigned an alias register, all outgoing checking edges are deleted. This could make other instructions in the $delayed$ list become ready for allocation and they are then allocated alias registers. Since an alias register is assigned only when the last instruction which checks it has been scheduled (i.e. the last checking edge has been removed), the alias register's liverange is closed at that point and it may be reclaimed. Register reclamation is performed via rotation of `AHPTR`. Finally, a memory operation W that has been allocated an alias register is removed from the $delayed$ list, and if it was a representative of a range in the continuum, it is marked as not representing the range anymore since later memory operations cannot be coalesced with it.

Alias Register Allocation Post-pass (Algorithm 5): A memory

Algorithm 3 delay_alias_reg (A), called by schedule()

```
1: num_delay_regs += 1
2: delayed = delayed ∪ A
3: AR(A) = NONE
4: if REP(CONT(SIB(A)), RANGE(A)) is NONE then
5:    REP(CONT(SIB(A)), RANGE(A)) = A
6: end if
```

operation A that has been coalesced with an earlier one W writes the same alias register as W. However, between W and A, there may have been rotation of `AHPTR` to reclaim other alias registers; consequently, AR(A) is offset by an amount equal to the difference

Algorithm 4 alloc_alias_reg (A), called by schedule()

```
1: AHPTR_AT(A) = AHPTR
2: if |CHECK(*, A)| == 0 then
3:    work = work ∪ A
4:    for all W in work do
5:       if P(W) ∧ COALESCED_WITH(W) is NONE then
6:          AR(W) = AHPTR + num_alloc_regs - AHPTR_AT(W)
7:          num_alloc_regs += 1
8:          num_delay_regs -= 1
9:       else
10:         if C(W) then
11:            AR(W) = AHPTR + num_alloc_regs - AHPTR_AT(W)
12:         end if
13:      end if
14:      work = work \ W
15:      for all A_i ∈ CHECK(W, A_i) do
16:         CHECK(W, *) = CHECK(W, *) \ CHECK(W, A_i)
17:         if (C(A_i) ∪ P(A_i)) ∧ (|CHECK(*, A_i)| == 0) then
18:            work = work ∪ A
19:         end if
20:      end for
21:   end for
22: end if
23: if num_alloc_regs > AHPTR then
24:    ROT(A_i) = num_alloc_regs - AHPTR
25:    AHPTR = num_alloc_regs
26:    num_alloc_regs -= 1
27: end if
28: for all W ∈ delayed do
29:    if AR(W) is not NONE then
30:       delayed = delayed \ W
31:       Remove W if it is REP(CONT(SIB(W)), RANGE(W))
32:    end if
33: end for
```

in AHPTR values at the time W and A were respectively scheduled. This ensures that W and A access the same physical alias register.

Algorithm 5 allocate_coalesced_alias_reg (A), called by schedule()

```
1: for all A ∈ memory operations do
2:    if P(A) then
3:       W = COALESCED_WITH(A)
4:       if W is not NONE then
5:          AR(A) = AR(W) - (AHPTR_AT(A) - AHPTR_AT(W))
6:       end if
7:    end if
8: end for
```

4.2.3 Example

Figure 8 illustrates the alias register allocation algorithm. Load ld_5 has been speculatively eliminated with forwarding from ld_2. Loads ld_6, ld_3, and ld_2 have all been speculatively hoisted above one or more stores. After scheduling ld_6, the algorithm adds the checking edge CHECK(st_4, ld_6), assigns P(ld_6) and C(st_4), and delays alias register allocation for ld_6. It also opens a continuum for $r0$ and marks ld_6 as the representative of that continuum. The algorithm handles the interference edge INTERFERE(st_1, ld_2) by splitting the $r0$ continuum into [-∞, $0x0$) and [$0x4$, +∞], with the former having ld_6 as its representative and the latter not having any representative. After scheduling ld_3, the algorithm adds the checking edge CHECK(st_1, ld_3), assigns P(ld_3) and C(st_1), and delays alias register allocation. It also opens a continuum for $r1$ with ld_3 as the representative. After scheduling ld_2, the algorithm fails to coalesce

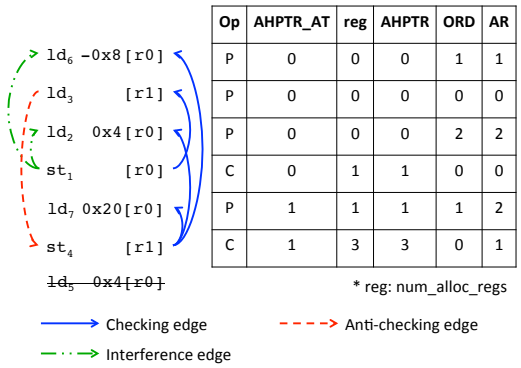

Figure 8. Alias register allocation example

Architectural Features	Parameter
8-wide VLIW	2 INT units, 2 FP units, 2 MEM units 1 BRANCH unit, 1 ALIAS unit
L1 I-Cache	8-way 32 KB, 1 cycle latency
L1 D-Cache	6-way 24KB, HW prefetch
L2 Cache	8-way 256KB, 3 cycle latency, HW prefetch
L3 Cache	16-way 8MB, 25 cycle latency
Memory	1GB, 104 cycle latency

Table 2. Architectural Parameters

ld_2 with ld_6 since even though COALESCENT(ld_2, ld_6), ld_6 is not REP(CONT(SIB(ld_2)), RANGE(ld_2))—this is the effect of the interfering store st_1. ld_2 is now marked as REP(CONT(SIB(ld_2)), RANGE(ld_2)). When st_1 is scheduled, it has no incoming edges and is therefore assigned order 0 (meaning it will check all valid alias registers 0 and above). Outgoing edges are removed, resulting in ld_3 being assigned order 0 after which num_alloc_regs is incremented to 1. After st_1, the algorithm rotates out the alias register with order 0, and increments AHPTR to 1. ld_7 is then successfully coalesced with ld_2, with its alias register allocation delayed. After st_4 is scheduled, it obtains an order 0 but with AHPTR 1, st_4 will check alias registers 1 and above. All outgoing edges are deleted and loads ld_6 and ld_2 are now assigned alias registers. ld_7 must update the same register as ld_2, however, its order is assigned in a post-pass to account for any freeing rotations in between ld_2 and ld_7 (a rotation of 1 in the example shown after st_1).

5. Evaluation

5.1 Methodology

Our target architecture is a VLIW processor similar to Transmeta Efficeon [11]. The dynamic binary translation framework shown in Figure 5 translates and optimizes x86 binaries into internal VLIW instructions. The CPU is modeled by a production-quality cycle-accurate simulator, which supports atomic region execution [8, 17] with up to 56 rotating alias registers (number of registers limited by encoding constraints in the ISA). Various significant architectural parameters of the VLIW processor are listed in Table 2.

Programs are first executed via interpretation. The system simultaneously identifies hot basic blocks. Once the basic blocks cross a hotness threshold, the dynamic optimizer forms a superblock region [11] along hot paths. Once a region is formed, the x86 code is translated into an internal representation (IR). At this stage, the optimizer performs alias analysis as well as the following optimizations: if-conversion, copy propagation, dead code elimination, loop invariant code motion, constant propagation, and common subexpression elimination. Finally, the optimizer performs register allocation and instruction scheduling.

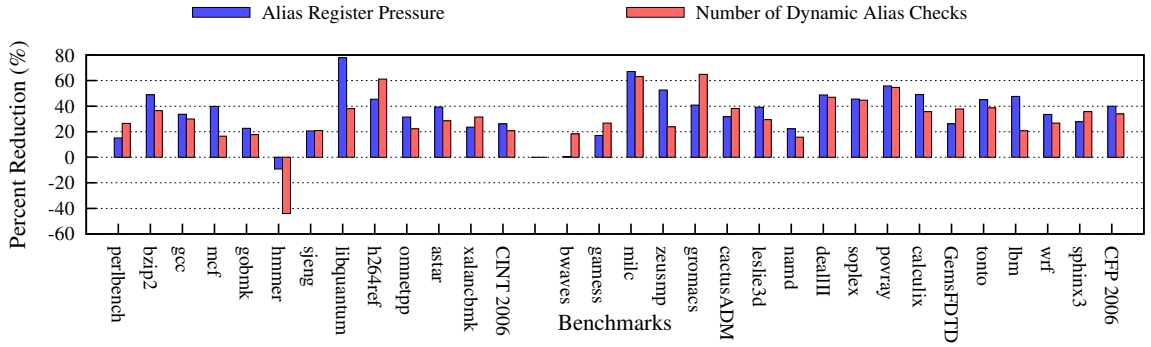

Figure 9. Reduction in alias register pressure and number of dynamic alias checks due to coalescing for 28 alias registers

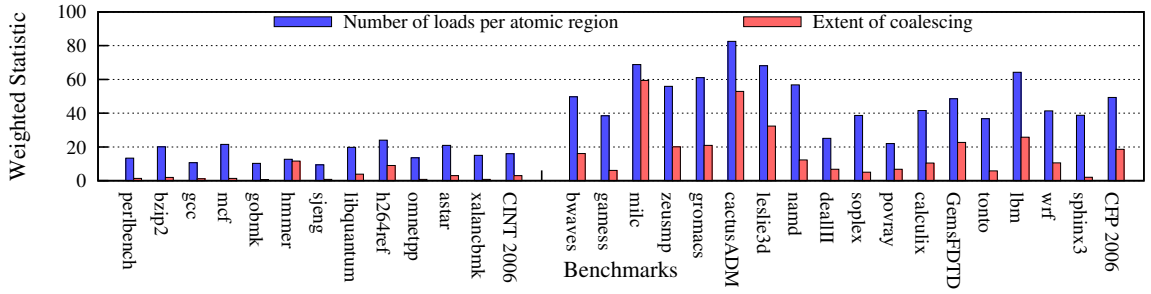

Figure 10. Average number of loads and number of coalesced loads per atomic region for 28 alias registers

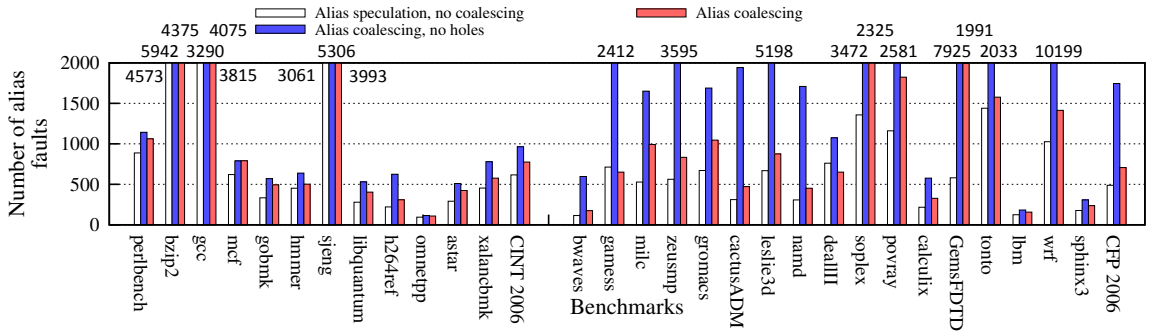

Figure 11. With 28 alias registers, alias coalescing results in very few spurious alias faults due to imprecise access information.

We evaluate alias coalescing on the SPEC CPU2006 suite (both INT and FP), compiled using icc at `-O3` enabling aggressive static optimizations that use data dependency and alias analyses. A benchmark is automatically divided into representative checkpoints for simulation. From each checkpoint, a benchmark is functionally simulated for 4 billion x86 instructions to warm up the dynamic optimizer, for 5 million x86 instructions to warm up the microarchitectural state, and simulated cycle-accurately for 25 million x86 instructions to gather performance data. Please note that the baseline for evaluation is SMARQ without alias coalescing.

5.2 Alias Register Pressure

Alias register pressure is measured by calculating the maximum number of valid alias registers required per atomic region, weighted by the *hotness* of the atomic region. Figure 9 shows that when 28 alias registers are available for allocation, alias coalescing reduces the average alias register pressure by a geomean of 26.09% and 39.96% respectively for the SPEC CINT2006 and SPEC CFP2006 benchmark suites. This reduction can be better placed into context when considering the extent of coalescing, defined as the weighted average of the number of memory operations coalesced into a single alias register, weighted by the hotness of the atomic region. The greater the extent of coalescing, the greater is the expected reduction in alias register pressure. Figure 10 shows the extent of coalescing for each of these benchmarks. The extent of coalescing is much larger for SPEC CFP benchmarks than SPEC CINT benchmarks, which explains why SPEC CFP benchmarks showed greater reduction in alias register pressure.

The only exception was 456.hmmer, where we observed a 9.26% increase in the alias register pressure. For this benchmark, the hot region consists of a loop, which the dynamic binary translator failed to capture as a single region due to paucity of alias registers in the baseline. As a result of the loop being split into multiple regions, the amount of alias speculation in each region was limited. However, with alias coalescing to the extent shown in Figure 10, the effectively increased alias register file size allowed the binary translator to capture the entire loop in a single region and unroll it multiple times – atomic region size in 456.hmmer increased on average by 51.54% compared to the baseline. This greater scope allowed the binary translator to perform increased alias speculative load hoisting, resulting in greater alias pressure visible in Figure 9.

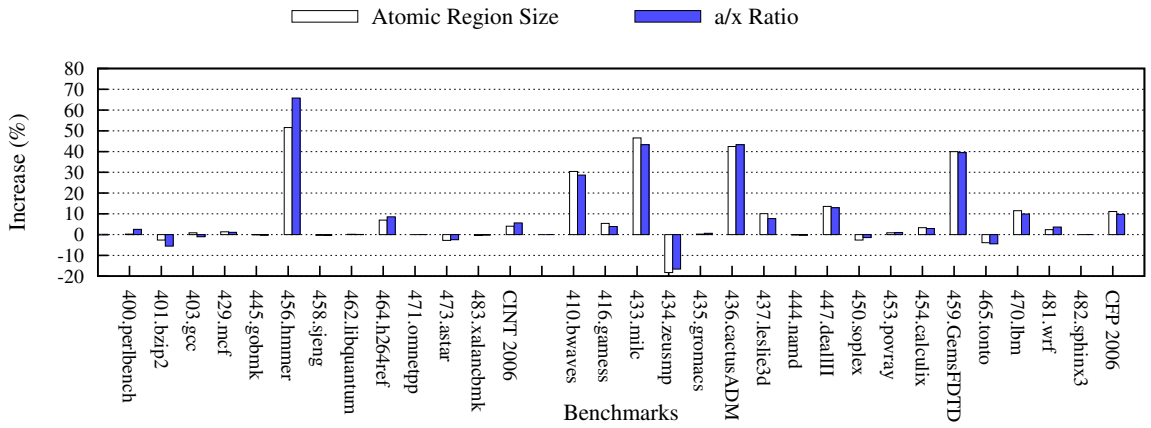

Figure 12. Increase in scheduling freedom in terms of average number of x86 instructions in each atomic region (Atomic Region Size) and average number of VLIW sub-instructions per x86 instruction (a/x Ratio) for the configuration with 28 rotating alias registers.

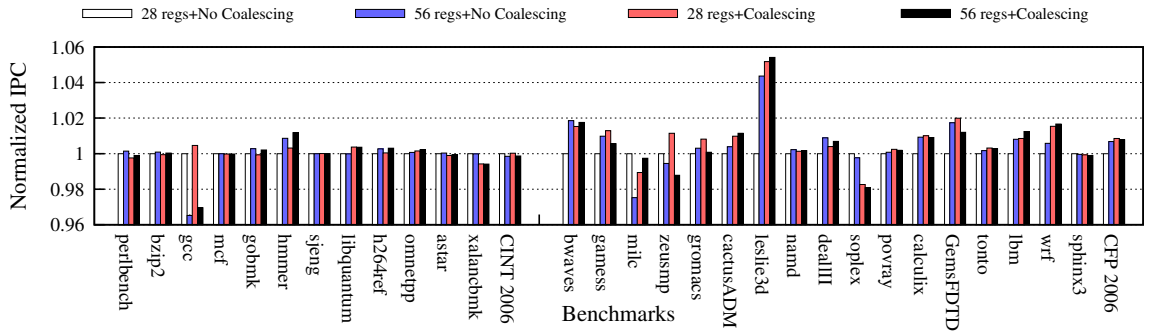

Figure 13. IPC gains due to coalescing with 28 and 56 alias registers, normalized with respect to IPC with 28 alias registers without coalescing. Graph shows how alias coalescing effectively makes the 28 alias register file appear twice as large.

5.3 Number of Dynamic Alias Checks

Reducing the number of dynamic alias checks per checking instruction reduces the latency to detect the presence of an alias fault which is necessary to retire an atomic region. As described previously, coalescing reduces the number of dynamic alias checks required from one check per speculated memory instruction to one check per coalesced set. This translates to a geomean of 20.73% and 33.87% reduction for the SPEC CINT2006 and SPEC CFP2006 benchmarks suites respectively (Figure 9). Note that while alias register pressure tracks the *maximum* number of simultaneously live alias registers in an atomic region, the number of dynamic alias checks tracks the *average* number of alias registers live and checked at each checking point in the program. The graph shows that alias coalescing effectively reduces both.

5.4 Reduction in Spurious Alias Faults

Alias coalescing may introduce spurious alias faults due to the imprecision of stored memory access information. Figure 11 shows the number of alias faults in the baseline without alias coalescing, and the increase due to alias coalescing, over the entire duration of benchmark execution (both warmup and performance measurement phases). The figure shows how highly effective hole-based alias register allocation (Section 3.3) is in reducing the number of spurious alias faults due to interfering instructions.

5.5 Instruction Scheduling Freedom

As described in Section 4, the optimized code is organized into speculative atomic regions by the dynamic optimizer. All optimizations listed in Section 5.1 are applied within an atomic region. Thus, the size of the atomic regions is critical to the effectiveness of optimizations. One factor affecting atomic region size is the availability of alias registers. If an atomic region required more alias registers than the total number available, it is split by the optimizer into smaller regions which are then reoptimized individually in the hope that they have reduced alias register pressure. As coalescing reduces alias register pressure, the effective size of the alias register file is now increased, allowing larger atomic regions to be successfully optimized speculatively. Figure 12 shows the increase in the average sizes of the atomic regions (in terms of number of x86 instructions) for each benchmark, weighted by the hotness of the atomic region. Mostly FP benchmarks benefit; 410.bwaves, 433.milc, 436.cactusADM, and 459.GemsFDTD show a considerable increase to the tune of 40% in the average atomic region size. As discussed in Section 5.2, 456.hmmer is the one INT benchmark to benefit greatly. These were precisely the benchmarks where the size of the hot atomic regions was constrained primarily a paucity of alias registers. In some of these benchmarks, loops which were earlier split into multiple regions are now successfully translated and optimized in a single region.

Another useful statistic to consider is the number of packets in a VLIW instruction (sub-instructions within a large VLIW instruction) per x86 instruction (we call this the *a/x* ratio). This ratio conveys the efficiency of the VLIW schedule generated for the processor. The higher the number, the better is the utilization of the processor resources. As can be seen in Figure 12, coalescing improves the *a/x* ratio for most benchmarks. There is a direct correlation between the increase in region sizes for benchmarks and the observed *a/x* ratio. This validates the intuition that the processor has greater scheduling freedom if more instructions are available to schedule.

5.6 Performance

Figure 13 shows performance of alias coalescing for two alias register file sizes (28 and 56 alias registers). The baseline is SMARQ with 28 alias registers (i.e. without coalescing). Recall that in our study speculative load chain hoisting is the only optimization using alias speculation. Figure 13 shows that coalescing provides a modest 0.85% IPC geomean improvement for FP benchmarks with 28 rotating alias registers. Note that this performance improvement is an additional gain over product-quality dynamically optimized code generated by SMARQ (which claims an average speedup of 40% over systems with no alias speculation). Some benchmarks see 2%-5% gains with coalescing owing to the increased scope of optimization due to an increased effective alias register file size. Two benchmarks exhibit slight performance drop because of spurious faults that suppress the hoisting of critical loads. Notably, the performance using 28 alias registers with coalescing is at par with the performance of 56 alias registers without coalescing. Thus, alias coalescing enables the system to retain the performance gains obtained due to SMARQ, while using a smaller register file (half in size) and much fewer dynamic alias checks. The number of dynamic alias checks for coalescing with 28 registers is 28.90% and 45.41% less for CINT and CFP benchmarks respectively than SMARQ without coalescing for 56 registers.

6. Related Work

Prior proposals make use of an alias register mechanism for alias detection in hardware [3, 6, 9, 11, 18, 21]. For instance, Itanium uses the Advanced Load Address Table (ALAT) [3] for memory alias detection. It requires stores to automatically check memory access ranges for all alias registers set by reordered loads.

In the context of dynamic optimization systems, Transmeta Efficeon [11] implements hardware alias detection through the use of static alias registers and uses a *bit-mask* in instructions to specify the individual alias registers whose memory access ranges need to be checked. Due to the limited space available for encoding instructions, this scheme restricts the size of the alias register file to 16 registers. This, in turn, limits the size of the regions for applying speculative optimizations.

Wang et al. [21] adopt the concept of order-based memory alias detection [9] to propose SMARQ. Rong et al. [18] propose an efficient register allocation scheme for such a rotating alias register file-based memory alias detection technique that formulates register allocation as a software pipelining problem. This work implements alias coalescing on top of the SMARQ alias register allocation algorithm. However, alias coalescing can also be used to improve the efficiency of most other alias speculation systems.

DeAliaser [1] enables alias speculation using the speculative bits available in modern hardware systems that implement transactional memory. Compared to alias coalescing and other rotating alias register based proposals, DeAliaser requires greater hardware cost, since all cache lines are now extended by a few bits. Also, DeAliaser requires checking against all speculative memory operations. These extra checks increase the energy consumption requirement for the DeAliaser memory alias speculation system.

7. Conclusion

Alias coalescing is an effective technique to simultaneously reduce alias register pressure and the number of dynamic comparisons performed per alias check, without decreasing performance. This paper contributed methods to identify memory operations suitable for coalescing, interfaces to alias checking hardware for coalescing, methods to allocate alias registers for coalesced memory operations, and hardware design to perform coalesced alias checking. As research in microprocessors using dynamic binary translation gains steam, we expect the methods and insights presented in this paper to be of great interest to system designers.

8. Acknowledgements

We would like to thank the many outstanding engineers at Intel Labs for their comments on this work. We thank the anonymous reviewers for their insightful comments. We also thank Nick Johnson, Feng Liu, Taewook Oh, Jordan Fix, Harshad Deshmukh, and Sergiy Popovych for their feedback on various drafts of this paper.

References

[1] W. Ahn, Y. Duan, and J. Torrellas. DeAliaser: Alias Speculation Using Atomic Region Support. In *ASPLOS*, 2013.

[2] D. A. Connors. Memory Profiling for Directing Data Speculative Optimizations and Scheduling. Master's thesis, University of Illinois, Urbana, IL, 1997.

[3] J. Crawford. Guest Editor's Introduction: Introducing the Itanium Processors. *IEEE Micro*, 20(5):9–11, Sept. 2000.

[4] X. Dai, A. Zhai, W.-C. Hsu, and P.-C. Yew. A General Compiler Framework for Speculative Optimizations Using Data Speculative Code Motion. In *CGO*, 2005.

[5] S. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. In *POPL*, 1998.

[6] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, and J. Mattson. The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-life Challenges. In *CGO*, 2003.

[7] M. Franklin and G. S. Sohi. ARB: A Hardware Mechanism for Dynamic Reordering of Memory References. *IEEE Transactions on Computers*, 45(5), May 1996.

[8] M. Herlihy and J. E. B. Moss. Transactional Memory: Arch. Support for Lock-free Data Structures. In *ISCA*, 1993.

[9] B. Holscher, G. Rozas, J. Van Zoeren, and D. Dunn. Systems and methods for reordering processor instructions. *US Patent*.

[10] M. Itzkowitz, B. J. N. Wylie, C. Aoki, and N. Kosche. Memory Profiling using Hardware Counters. In *SC*, 2003.

[11] K. Krewell. Transmeta Gets More Efficeon. *Microprocessor report. v.17*, October 2003.

[12] W. Landi. Undecidability of static analysis. *LOPLAS*, 1992.

[13] J. Lin, T. Chen, W.-C. Hsu, and P.-C. Yew. Speculative Register Promotion Using Advanced Load Address Table (ALAT). In *CGO*, 2003.

[14] 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 *PLDI*, 2003.

[15] M. Mehrara, J. Hao, P.-C. Hsu, and S. Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *PLDI*, 2009.

[16] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., 1997.

[17] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles. Hardware Atomicity for Reliable Software Speculation. In *ISCA*, 2007.

[18] H. Rong, H. Park, C. Wang, and Y. Wu. Allocating Rotating Registers by Scheduling. In *MICRO*, 2013.

[19] S. Rubin, R. Bodík, and T. Chilimbi. An Efficient Profile-analysis Framework for Data-Layout Optimizations. In *POPL*, 2002.

[20] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler. Scalable hardware memory disambiguation for high ilp processors. In *MICRO*, 2003.

[21] C. Wang, Y. Wu, H. Rong, and H. Park. SMARQ: Software-Managed Alias Register Queue for Dynamic Optimizations. In *MICRO*, 2012.

[22] Q. Wu, A. Pyatakov, A. Spiridonov, E. Raman, D. W. Clark, and D. I. August. Exposing Memory Access Regularities Using Object-Relative Memory Profiling. In *CGO*, 2004.